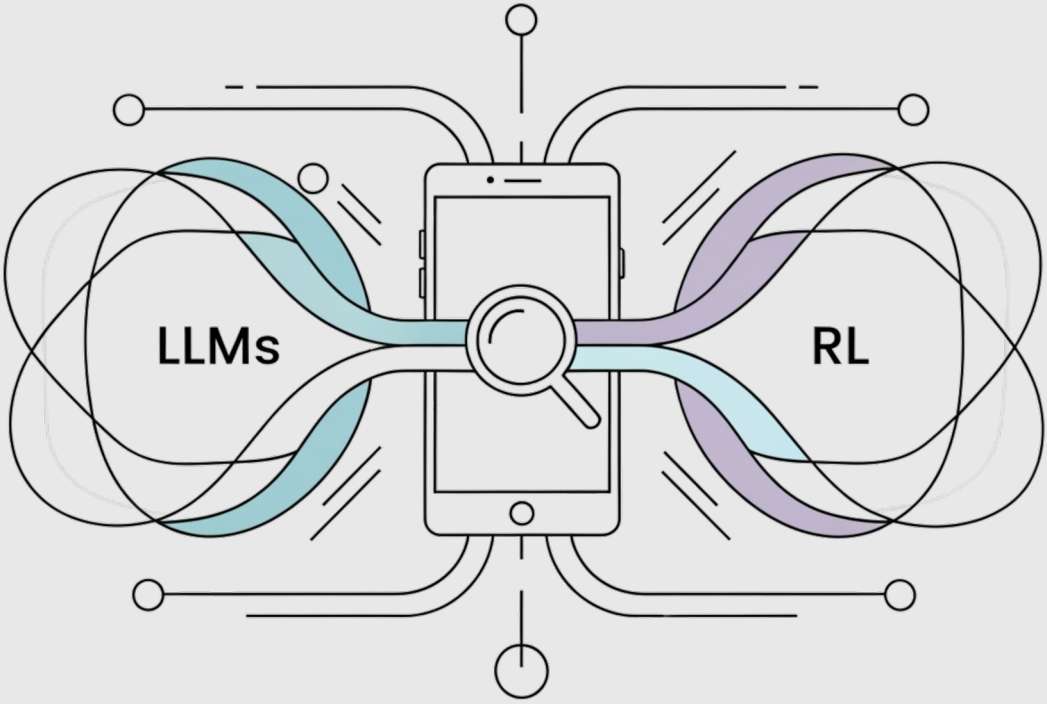



# Automatisiertes Testen mobiler Applikationen mit Hilfe von Large Language Models und Reinforcement Learning

Demian Frister

# Automatisiertes Testen mobiler Applikationen mit Hilfe von Large Language Models und Reinforcement Learning

von Demian Frister

# Automatisiertes Testen mobiler Applikationen mit Hilfe von Large Language Models und Reinforcement Learning

Demian Frister

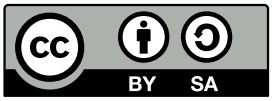
Dissertation, genehmigt von der KIT-Fakultät für Wirtschaftswissenschaften
des Karlsruher Instituts für Technologie (KIT), 2025

Tag der mündlichen Prüfung: 05.06.2025
Referent: Prof. Dr. Andreas Oberweis
Korreferent: Prof. Dr. Thomas Schuster

**Impressum**

Autor: Demian Frister

Stand: 26. Juni 2025






DOI: 10.5445/IR/1000182620

# Kurzfassung

Die zunehmende Bedeutung mobiler Applikationen bei gleichzeitig steigenden Qualitätsanforderungen erfordert effiziente Methoden zur automatisierten Softwarequalitätssicherung. Mit dem Ziel einer umfassenden Qualitätssicherung entwickelt diese Arbeit ein KI-gestütztes Testframework für mobile Applikationen, das erstmals Large Language Models (LLMs) zur Unit-Test-Generierung mit Reinforcement Learning (RL)-basierten Methoden zur GUI-Exploration kombiniert.
Der zweiphasige Ansatz umfasst eine Trainingsphase zur Modelloptimierung und Feinabstimmung von lokal lauffähigen Modellen sowie eine Operationsphase zur Testdurchführung.

Die LLM-basierte White-Box-Komponente wird auf einem spezialisierten und für diese Arbeit angepassten Datensatz trainiert und erreicht mit proprietären Modellen eine Anweisungsüberdeckung von bis zu 73%. Lokale Modelle unter 13 Milliarden Parametern eignen sich primär als Assistenzsysteme. Die implementierte Reparaturstrategie eliminiert syntaktische Fehler und ermöglicht die automatisierte Verbesserung fehlerhafter Tests.

Die RL-basierte Black-Box-Komponente realisiert durch einen trainierten Agenten die systematische Exploration der Benutzeroberfläche. Eine YOLO-basierte Feature-Extraktion beschleunigt das initiale Training. Die containerisierte Architektur gewährleistet reproduzierbare Testbedingungen und ermöglicht eine flexible Integration verschiedener Modelle.

Ein webbasiertes Frontend visualisiert die Testergebnisse und ermöglicht die Echtzeitüberwachung der Testausführung. Die systematische Evaluation nach den Standards der Systems and software product Quality Requirements and Evaluation (SQuaRE)-Normenreihe demonstriert die Effektivität des Frameworks für die automatisierte Qualitätssicherung mobiler Applikationen. Das System ermöglicht Entwicklern eine effiziente Generierung von Basistests und setzt damit Ressourcen für komplexere Testszenarien frei.

# Abstract

The increasing importance of mobile applications coupled with rising quality requirements necessitates efficient methods for automated software quality assurance. Aiming at comprehensive quality assurance, this thesis develops an AI-driven testing framework for mobile applications that for the first time combines Large Language Models (Große Sprachmodelle, LLMs) for unit test generation with Reinforcement Learning (Bestärkendes Lernen, RL)-based methods for GUI exploration.
The two-phase approach encompasses a training phase for model optimization and fine-tuning of locally executable models, followed by an operational phase for test execution.

The LLM-based white-box component is trained on a specially created dataset and achieves statement coverage of up to 73% using proprietary models. Local models below 13 billion parameters primarily serve as assistance systems. The implemented repair strategy eliminates syntactic errors and enables automated improvement and repair of faulty tests.

The RL-based black-box component enables systematic exploration of the user interface through an optimized agent. A YOLO-based feature extraction accelerates initial training. The containerized architecture ensures reproducible test conditions and allows flexible integration of various models.

A web-based frontend visualizes test results and enables real-time monitoring of test execution. The systematic evaluation according to SQuaRE standards demonstrates the effectiveness of the framework for automated quality assurance of mobile applications. The system enables developers to efficiently generate basic tests, freeing resources for more complex testing scenarios.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Promptverzeichnis

# Codeverzeichnis

# Abkürzungen

| | | |
|---|---|---|
| ADB | Android Debug Bridge | 21, 238, 241, 242, 293, 294, 330 |
| API | Application Programming Interface (Anwendungsprogrammier-schnittstelle) | 20, 93 |
| APK | Android Package | 20 |
| App | Mobile Applikation | 15–25, 27–30, 87, 147, 175, 177, 208, 210, 211, 213, 214, 337, V, VI, XIII |
| AST | Abstract Syntax Tree (Abstrakter Syntaxbaum) | 264, 273 |
| BPE | Byte-Pair Encoding (Bytepaar-Kodierung) | 138 |
| CI/CD | Continuous Integration and Continuous Delivery (Kontinuierliche Integration und Auslieferung) | 62, 87, 235, 332, 336, 338, 340 |
| CLM | Causal Language Model (Kausales Sprachmodell) | 142–144, 147, 154 |
| CNN | Convolutional Neural Network (Faltendes Neuronales Netz) | 125, 126, 149 |
| CoT | Chain-of-Thought | 163–166, 197, 267, 279, 312–314, 317, 321, XV |
| CRISP-DM | Cross-Industry Standard Process for Data Mining (Industrieübergreifender Standard-prozess für Data Mining) | 136, 137, 227, VII, XI |

| | | |
|---|---|---|
| CUT | Component under Test (Zu testende Komponente) | 53, 219 |
| DDPG | Deep Deterministic Policy Gradient | 181, 214 |
| DEX | Dalvik Executable (Ausführbare Dalvik-Datei) | 20 |
| DL | Deep Learning (Tiefes Lernen) | 107, 112, 123, 124, 126, 129, 141, 210, 214, 235, VI |
| DLC | Data Life Cycle (Datenlebenszyklus) | 227 |
| DUT | Device under Test (Zu testendes Gerät) | 53, 55, 70, 234, 285 |
| EP | Expert Parallelism (Expertenparallelisierung) | 134 |
| FCN | Fully Convolutional Network (Vollständiges Faltendes Neuronales Netz) | 132 |
| FFN | Feed-Forward-Network | 149 |
| Framework | Programmierumgebung bzw. Grundstruktur für Anwendungen | 20–22, 26, 27 |
| GUI | Graphical User Interface (Grafische Benutzeroberfläche) | 3, 21–23, 29, 45, 46, 53, 55, 64, 67, 68, 86–95, 122, 126, 129, 132, 174, 175, 177, 180, 182, 183, 186, 187, 204–207, 209, 210, 218–220, 283, 285, 324, 327, 328, 330, 331, VI, XI |

| | | |
|---|---|---|
| IDE | Integrierte Entwicklungsumgebung | 21 |
| KI | Künstliche Intelligenz | 2–4, 7, 21, 39, 40, 45–48, 72, 101, 129, 135, 187, 188, 297, 298, 335 |
| LLM | Large Language Model (Großes Sprachmodell) | 3–5, 7, 8, 54, 72, 73, 102, 113, 123, 129, 134–137, 139–141, 143, 145, 147, 149, 151–153, 155–157, 159–161, 163, 165, 167–169, 171, 172, 189–201, 203, 205, 207, 208, 214–216, 219, 222, 225–227, 229–233, 248, 250, 251, 253, 259, 262, 264–267, 270–278, 280, 281, 297, 302, 304–306, 311, 314, 319, 332, 334–337, 339, 340, I, III, VII, XII, XV |
| LoRA | Low-Rank Adaptation | 156–158, 230, 259, 260, 262 |
| LSTM | Long Short-Term Memory (Langzeit-Kurzzeit-Gedächtnis) | 145 |
| LUT | Language under Test (Zu testende Programmiersprache) | 3, 53, 218 |
| LvlUT | Level under Test (Zu testende Testebene) | 3, 53, 219 |

| | | |
|---|---|---|
| MDP | Markov Decision Process (Markow-Entscheidungsprozess) | 174, 175, 235, 294 |
| ML | Machine Learning (Maschinelles Lernen) | 101–104, 107, 112, 129, 147, 241, 248 |
| MLM | Masked Language Model (Maskiertes Sprachmodell) | 112, 140, 143, 144, 154, XII |
| MoE | Mixture of Experts | 123, 133–135, VI, XI |
| NLP | Natural Language Processing (Natürliche Sprachverarbeitung) | 102, 111, 150, 168 |
| NN | Neural Network (Neuronales Netz) | 113, 114, 118, XI |
| PEFT | Parameter-Efficient Fine-Tuning (Parameter-effizientes Feintuning) | 156, 227, 259 |
| PPL | Perplexity (Perplexität) | 168–170, 230, 261, 262, 304, 307, 308, 322 |
| PPO | Proximal Policy Optimization | 179, 180, 283, 288, 325 |
| PWA | Progressive Web App | 27 |
| QLoRA | Quantized Low-Rank Adaptation | 152, 158 |
| ReLU | Rectified Linear Unit | 116–118, 149 |

| | | |
|---|---|---|
| RL | Reinforcement Learning (Bestärkendes Lernen) | 3, 4, 6–8, 55, 102, 104, 129, 132, 173–175, 177–186, 189, 208, 209, 211, 213–215, 222–224, 234–237, 241, 248, 250, 282, 283, 285, 286, 288–290, 293, 294, 302, 324, 325, 328–330, 332–337, 339, I, III, VII, XII |
| RLHF | Reinforcement Learning from Human Feedback (Bestärkendes Lernen aus menschlichem Feedback) | 202, 339 |
| RNN | Recurrent Neural Network (Rekurrentes Neuronales Netz) | 118, 125, 126, 145, 147, 148, 151 |
| SAC | Soft Actor-Critic | 180, 182, 214, 283 |
| SFT | Supervised Fine-Tuning | 202 |
| SGD | Stochastic Gradient Descent (Stochastischer Gradientenabstieg) | 106, 121, 125, 154 |
| SMoE | Sparse Mixture of Experts | 134 |
| SQ | Softwarequalität: Erfüllungsgrad definierter Anforderungen einer Software | 1, 7, 17, 31–37, 39–43, 45–48, 52, 219, VI |
| SQuaRE | Systems and software product Quality Requirements and Evaluation (System- und Software-Qualitätsanforderungen und -bewertung) | 7, 36, 297–299, 336, II, IV, VI |

| | | |
|---|---|---|
| SUT | Software under Test (Zu testende Software) | 3, 53, 218 |
| TaS | Take a Step Back | 164, 165, 267, 268, 279, 314, 321, XV |
| TD3 | Twin Delayed Deep Deterministic Policy Gradient | 181, 182, 214 |
| TDD | Test Driven Development (Testgetriebene Entwicklung) | 72 |

# 1 Einleitung

Die zunehmende Digitalisierung und der technologische Fortschritt haben die Interaktion zwischen Mensch und Computersystemen grundlegend verändert. Bezogen auf die Internetnutzung übersteigt seit 2017 der Anteil mobiler Geräte mit Touch-Bedienung den Anteil klassischer, per Maus und Tastatur bedienbarer Computersysteme (StatCounter 2024b).

Diese sogenannten Mobilen Applikationen (Apps) unterscheiden sich von klassischen Desktop-Anwendungen durch ihre Heterogenität, variable Netzwerkbedingungen und touch-basierte Interaktionen (Linares-Vásquez u. a. 2017). Gleichzeitig steigen die Erwartungen an die Qualität und Zuverlässigkeit dieser Anwendungen, da sie zunehmend geschäftskritische Funktionen übernehmen (Pecorelli u. a. 2021).

Die Sicherstellung der Softwarequalität (SQ) dieser Applikationen kann durch White-Box-Tests zur Analyse der internen Programmstruktur sowie Black-Box-Tests zur Evaluation des Nutzerverhaltens (Liggesmeyer 2009) erfolgen.
Insbesondere im Bereich mobiler Anwendungen stellen die Testverfahren eine besondere Herausforderung dar, da die manuelle Durchführung einen erheblichen Zeit- und Ressourcenaufwand erfordert (Bertolino 2007) und die Vielzahl an Geräten und Betriebssystemversionen die Komplexität zusätzlich erhöht (Joorabchi, Mesbah und Kruchten 2013; Flora, X. Wang und Chande 2014).

## 1.1 Motivation

Die systematische Qualitätssicherung von Softwaresystemen gewinnt mit zunehmender Komplexität und Vernetzung an Bedeutung. Die Konsequenzen unzureichender Testverfahren manifestieren sich dabei in verschiedenen Formen und Ausmaßen. Ein prägnantes Beispiel hierfür stellt der Vorfall bei CrowdStrike im Juli 2024 dar, bei dem ein fehlerhaftes Software-Update zu einem globalen IT-Ausfall führte, der etwa 8,5 Millionen Windows-Systeme betraf und einen geschätzten Schaden von mindestens 5,4 Milliarden Dollar verursachte (itsecuritycoach 2024; Kunz 2024).
Dieser Vorfall unterstreicht die zentrale Bedeutung systematischer Softwaretests für die Qualitätssicherung.

Entwickler verbringen einen signifikanten Teil ihrer Arbeitszeit mit der Erstellung und Wartung von Tests (Pecorelli u. a. 2021). Insbesondere das manuelle Schreiben von Unit-Tests wird von Entwicklern als monoton empfunden und bindet wertvolle Ressourcen (Daka und G. Fraser 2014). Diese zeitliche Belastung führt häufig zu einer unzureichenden Testüberdeckung, wodurch die Softwarequalität direkt beeinträchtigt wird (Daka und G. Fraser 2014). Eine aktuelle Studie zeigt, dass die automatisierte Generierung von Tests durch KI-basierte Werkzeuge von Entwicklern als wichtigste potenzielle Unterstützung zur Verbesserung der Testentwicklung gesehen wird (Khemka und Houck 2024).

Bei mobilen Applikationen verstärken sich diese Herausforderungen durch die Anforderungen der Testumgebung (Muccini, Di Francesco und Esposito 2012; Flora, X. Wang und Chande 2014; Choudhary, Gorla und Orso 2015).

Somit erfordern die in der ISO/IEC/IEEE 29119-1:2022 definierten Testobjekte im mobilen Kontext eine differenzierte Betrachtung. Die Software under Test (SUT) muss auf einer Vielzahl von Geräten mit unterschiedlichen Eigenschaften funktionieren, wobei die Fragmentierung der Android-Plattform mit verschiedenen Versionen und Herstelleranpassungen diese Komplexität zusätzlich erhöht (Flora, X. Wang und Chande 2014). Für die Language under Test (LUT) werden spezifische Testframeworks für mobile Plattformen benötigt, da traditionelle Testansätze an die Besonderheiten mobiler Entwicklungsumgebungen angepasst werden müssen (Pecorelli u. a. 2021). Das Level under Test (LvlUT) umfasst sowohl die Code-Ebene als auch GUI-Interaktionen, wodurch unterschiedliche Teststrategien für Unit-Tests und Oberflächentests erforderlich werden (Muccini, Di Francesco und Esposito 2012; Pecorelli u. a. 2021).

Traditionelle automatisierte Testverfahren adressieren diese Herausforderungen nur unzureichend (Muccini, Di Francesco und Esposito 2012). Sie sind häufig nicht in der Lage, die Komplexität mobiler Anwendungen und ihrer Nutzungsszenarien vollständig abzudecken.
Neue KI-basierte Methoden, insbesondere Large Language Models (Große Sprachmodelle, LLMs), zeigen hingegen vielversprechende Ansätze. Diese können neben natürlicher Sprache auch Programmcode analysieren und Testcode generieren (Tufano, Drain u. a. 2020; Tufano, S. K. Deng u. a. 2022; Guilherme und Vincenzi 2023; Schäfer u. a. 2024). Die Integration von Reinforcement Learning (Bestärkendes Lernen, RL) ermöglicht zudem die systematische Exploration von Benutzeroberflächen unter realistischen Bedingungen (Toyama u. a. 2021).

Ein integrierter Ansatz, der LLM- und RL-Agenten als Basis für Testverfahren einsetzt, könnte die identifizierten Herausforderungen ganzheitlich adressieren.

Die lokale Ausführung der Künstliche Intelligenz (KI)-Modelle gewährleistet den Schutz sensibler Codeinformationen und die Einhaltung von Datenschutz- und Ressourcenanforderungen. Die Kombination von **White-Box-Tests** und **Black-Box-Tests** durch KI-gestützte Verfahren ermöglicht eine umfassendere und effizientere Qualitätssicherung mobiler Applikationen.

## 1.2 Ziel

Das übergeordnete Ziel dieser Arbeit ist die Entwicklung eines KI-gestützten Testframeworks für mobile Applikationen, das durch die Kombination von *White-Box-Tests* und *Black-Box-Tests* eine umfassende Qualitätssicherung ermöglicht.
Das Framework adressiert die in der Motivation beschriebenen Herausforderungen durch eine zweiphasige Pipeline mit Trainings- und Operationsphase für zwei zentrale Komponenten:

1. Eine LLM-basierte Komponente für White-Box-Tests zur automatisierten Generierung von Unit-Tests, die zur Identifikation von *Fehlern* und *Defekten* auf Code-Ebene dient.

2. Eine RL-basierte Komponente für Black-Box-Tests zur systematischen Exploration der Benutzeroberfläche, die *Störungen* und resultierende *Ausfälle* erkennt.

Daraus leiten sich folgende Unterziele ab:

**Konzeption der Framework-Architektur**

- Entwicklung einer zweiphasigen Pipeline mit Trainings- und Operationsphase für beide KI-Komponenten unter Berücksichtigung der in ISO/IEC/IEEE 29119-2:2021 definierten Testprozesse.
- Integration in eine containerisierte Testumgebung zur Gewährleistung reproduzierbarer Bedingungen gemäß ISO/IEC 25023:2016.
- Implementierung eines Demonstrators mit dazugehörigem Frontend zur Visualisierung der Testergebnisse und Testüberdeckung.

**LLM-Komponente für White-Box-Tests**

- Aufbau eines hochwertigen Trainingsdatensatzes für Java Unit-Tests unter Berücksichtigung der Datenqualitätsmetriken nach ISO/IEC 25024:2015.
- Implementierung eines effizienten Feinabstimmungsprozesses unter Verwendung von aktuellen Methoden zur Optimierung der Ressourcennutzung.
- Integration fortgeschrittener Prompt-Techniken zur Verbesserung der Testgenerierung.
- Entwicklung systematischer Validierungsstrategien für generierte Tests.
- Implementation iterativer Verbesserungsschleifen für fehlerhafte Tests durch Analyse der Compiler- und Testausführungsergebnisse.
- Erreichung einer Unit-Test-Überdeckung von annähernd 80% der Codezeilen, da dies als Richtwert für eine effektive Testüberdeckung gilt (Mockus, Nagappan und Dinh-Trong 2009).

**RL-Komponente für Black-Box-Tests**

- Implementation einer GUI-Interaktionsumgebung zur systematischen Exploration der Benutzeroberfläche.
- Entwicklung eines hybriden Belohnungssystems zur Förderung systematischer Exploration durch Kombination extrinsischer und intrinsischer Belohnungen.
- Integration von Ähnlichkeitsmessungen zur effektiven Zustandserkennung und der Erkennung von erfolgreichen Zuständsänderungen.
- Implementation spezialisierter Feature-Extraktionskomponenten für die Analyse von GUI-Elementen.

**Framework-Integration und Evaluation**

- Implementation einer einheitlichen Testausführungsumgebung mit reproduzierbaren Testbedingungen.
- Evaluation der generierten Unit-Tests anhand von Testüberdeckungsmetriken.
- Bewertung der GUI-Tests.
- Systematische Dokumentation identifizierter Fehler.

Ein besonderer Fokus liegt auf der lokalen Ausführbarkeit der KI-Modelle zur Gewährleistung von Datenschutz und Ressourceneffizienz. Durch die Verwendung quantisierter Modelle und effizienter Trainingsmethoden wird der Ressourcenbedarf minimiert. Das Framework soll dabei ohne tiefgreifendes KI-Fachwissen bedienbar sein und eine weitgehend automatisierte Testdurchführung ermöglichen.

Die technische Umsetzung orientiert sich an den etablierten Standards der SQuaRE-Normenreihe (ISO/IEC 24000:2014 ff.) und integriert moderne Ansätze der KI-gestützten Softwarequalitätssicherung.
Die Kombination aus LLM-basierter Unit-Test-Generierung und RL-gestützter GUI-Exploration ermöglicht eine umfassende und ressourcenschonende Testüberdeckung auf Code- und Benutzerebene.

## 1.3 Forschungsmethode und Gliederung der Arbeit

Das methodische Vorgehen dieser Arbeit orientiert sich an den in der ISO/IEC/IEEE 15288:2023 definierten Prozessen des Systems Engineering. Die Entwicklung des KI-basierten Testframeworks erfolgt dabei in fünf aufeinander aufbauenden Phasen, die systematisch die identifizierten Herausforderungen mobiler Applikationstests adressieren. Diese, in Tabelle 1.1 zusammengefassten Phasen, ermöglichen eine strukturierte Integration der LLM- und RL-Komponenten unter Berücksichtigung der definierten Qualitätsanforderungen.

In der **Problemdefinitionsphase** werden zunächst die spezifischen Herausforderungen beim Testen mobiler Applikationen identifiziert und die theoretischen Grundlagen zu SQ und Testverfahren erarbeitet. Darauf aufbauend erfolgt eine Beschreibung der Grundlagen von LLMs und RL für automatisierte Tests.

Die **Konzeptentwurfsphase** beginnt mit einer Analyse verwandter Arbeiten und bestehender Lösungsansätze. Basierend auf diesen Erkenntnissen wird das Gesamtkonzept für das integrierte Testframework entwickelt, wobei LLMs für White-Box-Tests zur Unit-Test-Generierung und RL für Black-Box-Tests zur GUI-Exploration ausgewählt werden.

Im **detaillierten Systementwurf** erfolgt die Ausarbeitung der Trainings- und Operationsphasen für beide KI-Komponenten sowie die Definition der Systemschnittstellen. Zusätzlich wird eine containerisierte Testumgebung entworfen, die eine reproduzierbare Ausführung ermöglicht.

Die **Implementierungs- und Integrationsphase** umfasst die konkrete Umsetzung der LLM-basierten Unit-Test-Generierung und der RL-basierten GUI-Tests. Die einzelnen Komponenten werden in das Gesamtframework integriert und durch ein Frontend zur Visualisierung ergänzt.

In der abschließenden **Verifizierungs- und Validierungsphase** werden sowohl die einzelnen Testkomponenten als auch das Gesamtsystem evaluiert.

| Phase | Ergebnisse | Kapitel |
|---|---|---|
| Problemdefinition | Herausforderungen beim Testen mobiler Applikationen | 2 & 3.3 |
| | Grundlagen Softwarequalität & Tests | 3 |
| | Grundlagen von LLMs & RL | 4 |
| Konzeptentwurf | Analyse verwandter Arbeiten | 5 |
| | Gesamtkonzept des Frameworks & Auswahl der KI-Komponenten | 6 |
| Detaillierter Systementwurf | Trainings- & Operationsphasen | 6.2 & 6.3 |
| | Systemschnittstellen | |
| Implementierung & Integration | Containerisierte Testumgebung | 7.1 |
| | Integration & Frontend | 7.2 |
| | LLM-basierte Unit-Tests | 7.3 |
| | RL-basierte GUI-Tests | 7.4 |
| Verifizierung & Validierung | Evaluation der Testkomponenten | 8 |
| | Validierung des Gesamtsystems | |

**Tabelle 1.1:** Übersicht der Projektphasen und deren Ergebnisse

## 1.4 Publikationen

Die folgenden Listen stellen eine Übersicht über Publikationen, die in die vorliegende Arbeit eingeflossen sind, und studentische Abschlussarbeiten dar. Die studentischen Abschlussarbeiten haben insbesondere zur Implementierung einzelner Komponenten des Testframeworks und zur Evaluation von Zwischenergebnissen beigetragen.

### 1.4.1 Eigene Publikationen

Demian Frister geb. Hartmann (2019). „Sensor Integration with ZigBee Inside a Connected Home with a Local and Open Sourced Framework: Use Cases and Example Implementation“. Las Vegas, NV, USA

Demian Frister, Andreas Oberweis und Aleksandar Goranov (2020). „Automated Testing of Mobile Applications Using a Robotic Arm“. In: *CSCI*. 2020 International Conference on Computational Science and Computational Intelligence (CSCI). IEEE, S. 1729–1735.

Demian Frister, Aleksandar Goranov und Andreas Oberweis (2021). „Industrieroboter testet Apps“. In: *German Testing Magazin* (Janaur 2021)

Jacob Hoffmann und Demian Frister (2024). „Generating Software Tests for Mobile Applications Using Fine-Tuned Large Language Models“. In: *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test*. AST '24. Lisbon, Portugal: Association for Computing Machinery, S. 76–77.

Jacob Hoffman und Demian Frister (2025). „Automatische Testgenerierung mit generativer KI“. In: *German Testing Magazin* (Januar 2025)

### 1.4.2 Abschlussarbeiten

Im Zusammenhang mit dem Promotionsvorhaben wurden verschiedene Teilziele und verwandte Aspekte auch im Rahmen studentischer Abschlussarbeiten aufgegriffen und vertieft. Die Arbeiten wurden vom Autor der vorliegenden Arbeit fachlich betreut.

#### 1.4.2.1 White-Box-Testen

Yicheng Feng (2020). „Conception and comparison of test options for mobile applications“. Masterarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Michael Mayer (2022). „Automatische Testgenerierung für mobile Anwendungen mit Hilfe von dem Transfomer Modell GPT-J“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Khalil Sakly (2022). „Generating automated test cases using the transformer model GPT“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Jacob Hoffmann (2023). „Generating Software Tests Using Transformer Networks“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Nicolas Kaum (2024). „Automatische Generierung von Testfällen für gestengesteuerte Applikationen mit Hilfe lokaler Open-Source Large Language Models“. Masterarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Zhi Wang (2024). „Evaluating automated generated software tests for Mobile Application using LLM“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Tristan Moritz Dohmen (2024). „Steigerung der Softwarequalität mobiler Anwendungen mittels automatisierter Testgenerierung durch den Einsatz maschinellen Lernens“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

David Diemand (2024). „Testen mit GPT – automatische Testgenerierung mit Transformernetzwerken“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Tamino Ludwig (2024). „Evaluation von LLM-generierten App-Tests“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Davide Alpino (2024). „Konzeption und Implementierung einer Metrik zur Evaluierung von LLM-generierten Softwaretests“. Masterarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Bjarne Koll (2025). „Testgenerierung mittels großer Sprachmodelle und Feedbackschleifen“. Masterarbeit. Karlsruhe: Karlsruher Institut für Technologie

#### 1.4.2.2 Black-Box-Testen

Tarek Sellami (2021). „Detecting UI-related Bugs through Automated Black-Box Testing“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Tobias Winter (2022). „Automatisches Testverfahren für Flutter Applikationen mittels Roboterarm“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Kozsir Norbert (2022). „Developer Centric Cross-Platform E2E Mobile Testing“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Can Pigur (2024). „Entwicklung eines Reinforcement Learning Agenten zum automatisierten Testen von Android Anwendungen“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Vito Pepe Jaromir Völker (2022). „Erkennung von UI-Elementen in mobilen Applikationen mit YOLOv5“. B.Sc. Karlsruhe Institute of Technology

Benjamin Meyjohann (2022). „Detection and Segmentation of Interactive Elements in Mobile Applications using Deep Learning Trained on an Automatically Generated and Labeled Dataset“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

#### 1.4.2.3 Abschlussarbeiten mit indirektem Bezug zur vorliegenden Arbeit

Marvin Ruchay (2019). „Intuitive Fernsteuerung eines Roboterarms durch allgegenwärtige Smartphone Sensoren“. Masterarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Davinny Sou (2019). „Testmanagement mit XML-Netz - basierten Unternehmenssoftware - Referenzmodellen“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Cynthia Diedrich (2021). „Konzeptionierung und Umsetzung eines modularen Konfigurators für eine einheitliche Robotersteuerung“. Masterarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Valentin Scheiger (2021). „Approximation der inversen Kinematik eines 7-DOF Roboterarms mittels Deep Reinforcement Learning“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Jan Söder (2021). „Anforderungsanalyse an einen nichtindustriellen Staubsaugroboter für den Einsatz im Büroumfeld“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Annika Greif (2021). „Analyse der Privatsphäreneinschränkungen bei Haushaltssaugrobotern“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Marc Streckfuß (2022). „Dynamische regelbasierte teilautonome Aktionsplanung für Roboter“. Masterarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Julian Müller (2022). „Fernsteuerung eines Roboterarms mit sieben Freiheitsgraden durch kamerabasierte Human Pose Estimation“. Masterarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Benjamin Volkert (2022). „Mensch-Roboter-Interaktion: Erkennen einer Benutzerabsicht mit anschließender Objektübergabe“. Masterarbeit. Karlsruher Institut für Technologie.

Ilia Tanev Bagov (2022). „Learning Implicit Preferences of Calendar Users“. Masterarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Vincent Müller (2022). „Greifen von Objekten mit einem 7-DoF-Roboterarm mithilfe von Deep Reinforcement Learning“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Barkin Karatay (2022). „Entwicklung eines formalen Modells zur Umsetzung eines automatisierten Ablaufs in einem Smart Home“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

Felix Marschall (2023). „Process Management in a Smart Home context“. Bachelorarbeit. Karlsruhe: Karlsruher Institut für Technologie.

### 1.4.3 Seminararbeiten mit Bezug zur vorliegenden Arbeit

Franziska Thompson (2022). „Fehlerklassifizierung bei mobilen Applikationen“. Seminararbeit. Karlsruhe: Karlsruher Institut für Technologie.

Can Pigur und Benjamin Meyjohann (2024). „Enhanced AI Feedback for Fine-Tuning on Preferences for Java Unit Test Code Generation“. Seminararbeit. Karlsruhe: Karlsruher Institut für Technologie.

Lars Bissinger, Robert Alexander Müller, Kaloyan Kalinov Naydenov, Lorenz Neugebauer, Daniel Maximilian Riffel und Philip Wagner (2024). „Entwicklung einer Testplattform für mobile Applikationen“. Seminararbeit. Karlsruhe: Karlsruher Institut für Technologie

# 2 Grundlagen Mobiler Applikationen

Mobile Applikationen (Apps) haben sich zu einem wesentlichen Element der digitalen Infrastruktur entwickelt. Der Begriff „App“ leitet sich von „*Applikation*“ bzw. „Anwendungssoftware“ ab und bezeichnet speziell für mobile Endgeräte entwickelte Softwareanwendungen.

Der Begriff „*Software*“ bezeichnet im Allgemeinen Programme für Computersysteme. Die ISO/IEC/IEEE 24765:2017 unterscheidet dabei zwischen Anwendungssoftware, die spezifische Aufgaben für Benutzer erfüllt, und System-Software, die die Lauffähigkeit der Anwendungssoftware gewährleistet.

Die IEEE definiert Anwendungssoftware (engl. application software) als Software, die für die Erfüllung von Benutzeranforderungen entwickelt wurde, wie zum Beispiel Software für Navigation, Buchhaltung oder Prozesskontrolle ISO/IEC/IEEE 24765:2017.

Die ISO, IEC und IEEE 2022b, S. 43 charakterisiert mobile Systeme als:

> *„ [. . . ] typically portable computing devices without physical connections for communications or power (e.g. they are often battery-powered). Due to their need for portability, mobile devices are typically small enough to be hand-held. Examples of mobile devices are smartphones, media players, smart watches, and smart cards. “*

Apps sind Anwendungssoftware, die insbesondere für portable, oft batteriebetriebene Geräte entwickelt wurden. Die kontinuierliche Weiterentwicklung der mobilen Technologien hat zu einer Expansion der Anwendungsmöglichkeiten geführt, die über die ursprünglichen Funktionen mobiler Systeme hinausgehen (Wasserman 2010).
Apps unterliegen dadurch besonderen Anforderungen und Einschränkungen, die sich aus den charakteristischen Eigenschaften mobiler Geräte ergeben: begrenzte Ressourcen, Touch-basierte Eingabe, variable Netzwerkbedingungen und diverse Sensoren zur Umgebungserfassung.

Diese Anforderungen unterscheiden mobile Applikationen von klassischer Desktop-Software und erfordern damit auch angepasste Entwicklungs- und Testmethoden (Flora, X. Wang und Chande 2014; Muccini, Di Francesco und Esposito 2012).

Flora, X. Wang und Chande (2014) kategorisieren die Unterschiede in drei Hauptbereiche: *Hardware*, *Software* und *Kommunikation*.

Um eine einfache Handhabung für den Benutzer zu gewährleisten, müssen Apps die begrenzten Ressourcen der *Hardware* mobiler Geräte, wie Prozessorleistung, Arbeitsspeicher und Batteriekapazität, effizient nutzen. Dies stellt Anforderungen an die Effizienz und das Ressourcenmanagement (Wasserman 2010; D. Zhang und Adipat 2005). Die zusätzlich zu berücksichtigende Anforderungen beeinflussen die Entwicklung und Funktionalität von Apps maßgeblich.

Die Eingabemechanismen divergieren ebenfalls von denen klassischer Desktop-Systeme, wobei Touchscreens und die Bedienung mittels Gesten eine zentrale Rolle einnehmen. Die Variabilität der Bildschirmgrößen und Bauformen unter den verschiedenen mobilen Systemen und Geräten stellt eine zusätzliche Herausforderung dar, da das Layout für unterschiedliche Seitenverhältnisse und Auflösungen implementiert werden muss (D. Zhang und Adipat 2005; Dehlinger und Dixon 2011).

*Softwareseitig* zeichnen sich Apps durch weitere Anforderungen an die Benutzeroberfläche und gestengesteuerte Interaktionsmuster aus. Die Notwendigkeit einer kurzen Startzeit, einer intuitiven Bedienung und einer effizienten Ressourcennutzung sind kritische Faktoren (Flora, X. Wang und Chande 2014; Wasserman 2010). Zudem müssen Aspekte wie Personalisierung, Lokalisierung und die Interaktion mit anderen Apps auf dem Gerät und dem Betriebssystem berücksichtigt werden. Die Fähigkeit von Apps, durch Sensoren kontextabhängige Aktionen zu ermöglichen, erhöht die Komplexität der Entwicklung (Dehlinger und Dixon 2011; Muccini, Di Francesco und Esposito 2012).

Im Bereich der *Kommunikation* müssen Apps mit variablen Netzwerkbedingungen umgehen können. Sie müssen auch bei instabilen oder langsamen Verbindungen zuverlässig funktionieren (Flora, X. Wang und Chande 2014; Muccini, Di Francesco und Esposito 2012). Dies erfordert eine robuste Programmierung, die auf Verbindungsabbrüche und wechselnde Netzwerkqualitäten adäquat reagieren kann (Muccini, Di Francesco und Esposito 2012; D. Zhang und Adipat 2005).

Zusätzlich stellt die Heterogenität der Geräte, Betriebssystemversionen und Nutzungsszenarien hohe Anforderungen an die SQ von Apps. Insbesondere die Gewährleistung von Zuverlässigkeit, Leistung und Benutzerfreundlichkeit gewinnt im Kontext mobiler Applikationen eine erhöhte Bedeutung (Muccini, Di Francesco und Esposito 2012; Linares-Vásquez u. a. 2017).

Um die Anforderungen an die Entwicklung und Qualitätssicherung von Apps vollständig zu erfassen, ist eine detaillierte Betrachtung der grundlegenden Arten mobiler Betriebssysteme und deren Entwicklungsumgebungen erforderlich. Die folgenden Abschnitte widmen sich dieser Analyse unter Berücksichtigung der Komplexität und Diversität der Plattformen (Linares-Vásquez u. a. 2017; Muccini, Di Francesco und Esposito 2012).

## 2.1 Betriebssysteme

Mobile Betriebssysteme bilden das Fundament für die Funktionalität und Leistungsfähigkeit mobiler Geräte. Die Wahl des Betriebssystems und dessen Umgebung bestimmt maßgeblich die Anforderungen an die Entwicklung und Qualitätssicherung mobiler Applikationen für Entwickler und Tester. Zudem beeinflusst sie die verfügbaren Werkzeuge und Teststrategien (Linares-Vásquez u. a. 2017; Muccini, Di Francesco und Esposito 2012).

Apps für mobile Geräte sind im Gegensatz zu traditioneller Desktop-Software speziell für die Anforderungen und Limitationen mobiler Geräte konzipiert (Flora, X. Wang und Chande 2014). Diese Spezialisierung manifestiert sich in Aspekten wie effizientem Energiemanagement, touchbasierten Benutzerschnittstellen, integrierter Sensorik und angepassten Sicherheitsmodellen (Wasserman 2010).

Der Markt für mobile Betriebssysteme wird von zwei dominanten Akteuren bestimmt: *Android*, entwickelt von Google, und *iOS*, entwickelt von Apple. Diese Duopolsituation prägt die Entwicklungslandschaft mobiler Applikationen. Aktuelle Statistiken zeigen, dass diese beiden Plattformen zusammen über 99% des globalen Marktanteils ausmachen, wobei Android etwa 72% und iOS rund 27% hält (StatCounter 2024a). Um eine breite Benutzerbasis zu erreichen, sollte eine entwickelte App typischerweise mindestens eine dieser Plattformen unterstützen. Die Wahl des Betriebssystems beeinflusst maßgeblich den Entwicklungsprozess, die verfügbaren Werkzeuge und die Teststrategien für Apps.

Android, als Open-Source-Plattform, ermöglicht eine breitere Palette von Entwicklungsansätzen und bietet mehr Flexibilität für Entwickler und Tester. Im Gegensatz dazu präsentiert iOS ein stärker kontrolliertes Ökosystem mit eigenen Entwicklungsstandards und -werkzeugen (Goadrich und Rogers 2011). Ein Beispiel hierfür sind die Human Interface Guidelines[1] von Apple, die detaillierte Anforderungen und „Best Practices“ für die Gestaltung und Entwicklung von iOS-Apps festlegen, was zu einem einheitlicheren Benutzererlebnis nach ISO 9241-210:2019 auf der Plattform beiträgt.

In den folgenden Unterabschnitten werden die Charakteristika von Android und iOS näher betrachtet, mit Fokus auf ihren Architekturen und Entwicklungsumgebungen. Ein besonderes Augenmerk wird auf Android gelegt, da es als quelloffenes System eine größere Flexibilität bietet und somit besonders relevant für die Entwicklung neuer Testansätze ist (Linares-Vásquez u. a. 2017).

### 2.1.1 Android

Android ist ein Open-Source-Betriebssystem für mobile Geräte, das von Google[2] entwickelt und 2008 erstmals veröffentlicht wurde (Goadrich und Rogers 2011). Es basiert auf einem modifizierten Linux-Kernel[3] und wurde speziell für Touchscreen-Mobilgeräte wie Smartphones und Tablets konzipiert.

Die Architektur von Android ist in mehrere Schichten unterteilt, die eine klare Trennung von Verantwortlichkeiten ermöglichen (Google 2024c):

1. *Linux-Kernel*: Bildet die Grundlage des Systems und steuert Hardware--Interaktionen.

[1] `https://developer.apple.com/design/human-interface-guidelines`
[2] Google ist ein weltweit agierendes Technologieunternehmen (`https://www.google.de/contact/impressum.html`).
[3] Siehe: `https://source.android.com/docs/core/architecture/kernel?hl=de`

2. *Hardware-Abstraktionsschicht (HAL)*: Stellt standardisierte Schnittstellen für Hardware-Komponenten bereit.
3. *Native Bibliotheken und Android Runtime*: Enthält C/C++-Bibliotheken und die Laufzeitumgebung für Java-Code.
4. *Application Framework (Framework)*: Bietet Application Programming Interface (API) für Entwickler zur Erstellung von Apps .
5. *System Apps*: Vorinstallierte Apps wie E-Mail, SMS, Kalender etc.

Android-Apps werden hauptsächlich in Java[4] oder Kotlin[5] entwickelt (Flora, X. Wang und Chande 2014). Andere Programmiersprachen können ebenfalls verwendet werden, sind aber weniger verbreitet. Der Quellcode wird in Dalvik Executable (DEX) Format kompiliert und in Android Package (APK) Dateien verpackt. Der kompilierte Code ist für die Ausführung auf Android-Geräten optimiert und enthält alle notwendigen Ressourcen und Abhängigkeiten.

Zur Auslieferung werden mehrere APKs in einem Android App Bundle[6] (*.aab) zusammengefasst. Diese können über den Google Play Store oder alternative Quellen auf Geräten installiert werden.

Die empfohlene App-Architektur für Android-Apps besteht aus mindestens zwei Schichten (Google 2024b):

1. *UI-Ebene*: Verantwortlich für die Darstellung der Anwendungsdaten auf dem Bildschirm.
2. *Daten-Ebene*: Enthält die Geschäftslogik der Anwendung und stellt Anwendungsdaten bereit.

[4] `https://www.java.com/`
[5] `https://kotlinlang.org/`
[6] Android App Bundle: Ein Veröffentlichungsformat für Android Apps (`https://developer.android.com/guide/app-bundle?hl=de`)

Optional kann eine dritte Schicht, die Domain-Ebene, hinzugefügt werden, um Interaktionen zwischen GUI und Daten-Ebene zu vereinfachen.

Android bietet verschiedene integrierte Werkzeuge für Entwicklung und Tests (Google 2024b):

- Android Studio[7]: Entwicklungsumgebung für Android.
- Android Debug Bridge (ADB)[8]: Tool zur Gerätekommunikation.
- Espresso[9]: Framework für GUI-Tests.
- JUnit[10]: Framework für Unit-Tests.
- Monkey[11]: Werkzeug für Stresstests.

Die Testwerkzeuge *Espresso* und *Monkey* werden im Kapitel 3.3 basierend auf der Erläuterung von Softwaretests im Allgemeinen weiter ausgeführt. Die Offenheit und Flexibilität von Android ermöglicht experimentelle Testmethoden, einschließlich des Einsatzes von KI-basierten Methoden.

[7] Android Studio ist die offizielle integrierte Integrierte Entwicklungsumgebung (IDE) für Android-Entwicklung (siehe: `https://developer.android.com/studio?hl=de`).

[8] Android Debug Bridge ist ein vielseitiges Kommandozeilen-Tool zur Kommunikation mit Android-Geräten (siehe: `https://developer.android.com/tools/adb?hl=de`).

[9] Espresso ist ein Test-Framework für GUI-Tests von Android-Apps (siehe: `https://developer.android.com/training/testing/espresso?hl=de`).

[10] JUnit ist ein Framework für Unit-Tests auf Komponentenebene (siehe: `https://junit.org/junit5/docs/current/user-guide/`).

[11] Monkey ist ein Werkzeug zum Erzeugen von Pseudo-zufälligen GUI-Events für Stresstests (siehe: `https://developer.android.com/studio/test/other-testing-tools/monkey?hl=de`).

### 2.1.2 iOS

iOS ist das proprietäre mobile Betriebssystem von Apple, das 2007 mit der Einführung des ersten iPhones veröffentlicht wurde. Im Gegensatz zu Android ist iOS ein geschlossenes System, das ausschließlich auf Apple-Geräten wie *iPhone*, *iPad* und *iPod Touch* läuft.

Die Architektur von iOS besteht aus fünf Hauptschichten, die eine strukturierte Basis für App-Entwicklung bieten (Apple 2015):

1. Cocoa Touch: Enthält wichtige Frameworks für App-Entwicklung.
2. Media: Bietet Grafik-, Audio- und Videotechnologien.
3. Core Services: Stellt fundamentale Systemdienste für Apps bereit.
4. Core OS: Verwaltet die Low-Level-Funktionen des Systems.
5. Kernel and Device Drivers: Verwaltet die Hardware des Systems.

Diese Architektur weist starke Ähnlichkeiten mit der Android-Architektur auf, was auf gemeinsame Grundprinzipien im Design mobiler Betriebssysteme hindeutet.

iOS-Apps werden hauptsächlich in Swift[12] oder Objective-C[13] entwickelt und exklusiv über den Apple App Store vertrieben.

Apple stellt mit Xcode eine integrierte Entwicklungsumgebung bereit, die umfassende Tools für Entwicklung, Debugging und Testing umfasst (Apple 2024).

Für das Testen von iOS-Apps bietet Apple verschiedene Frameworks:

- XCTest: Hauptframework für Unit- und GUI-Tests.

[12] `https://swift.org/`
[13] `https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html`

- XCUITest: Erweiterung von XCTest für GUI-Automatisierung.
- TestFlight: Plattform für Beta-Tests mit externen Testern.

Im Vergleich zu Android bietet iOS eine homogenere Plattform mit geringerer Fragmentierung, was den Testprozess in gewisser Hinsicht vereinfacht. Allerdings beschränkt die Geschlossenheit des Systems die Möglichkeiten für tiefgreifende Systemtests und alternative Testmethoden (Muccini, Di Francesco und Esposito 2012).

Aufgrund dieser Einschränkungen und der größeren Verbreitung von Android konzentriert sich diese Arbeit primär auf die Entwicklung eines Testsystems für Android-Apps. Es wird jedoch angestrebt, die entwickelten Ansätze so zu gestalten, dass sie grundsätzlich auch auf iOS-Apps übertragbar sind. Die Ähnlichkeiten in der Architektur beider Systeme unterstützen diese Übertragbarkeit und ermöglichen somit eine breitere Anwendbarkeit der Forschungsergebnisse.

## 2.2 Applikationsentwicklung

Die Entwicklung mobiler Applikationen kann auf verschiedene Arten erfolgen, wobei jeder Ansatz Vor- und Nachteile bietet. Delia u. a. (2015) identifizieren drei Hauptansätze: *native Apps*, *mobile Web-Apps* und *hybride Apps*.

Die Wahl des Entwicklungsansatzes hängt von verschiedenen Faktoren ab, darunter die Anforderungen der App, der Zielmarkt, die verfügbaren Ressourcen und die angestrebte Benutzererlebnis.

### 2.2.1 Native Apps

Native Apps werden spezifisch für eine bestimmte Plattform entwickelt, unter Verwendung plattformeigener Programmiersprachen und Entwicklungstools (Flora, X. Wang und Chande 2014).

Native Apps werden direkt auf dem Betriebssystem des Geräts ausgeführt und haben vollen Zugriff auf alle Hardware-Komponenten und Systemfunktionen. Sie werden in plattformspezifischen Sprachen geschrieben und für die Prozessorarchitektur des Zielgeräts kompiliert. Dies ermöglicht eine angepasste Ausnutzung der Geräteressourcen und eine hohe Ausführungsgeschwindigkeit.

Für die Entwicklung nativer Android-Apps wird hauptsächlich Android Studio (siehe 2.1.1) verwendet, das auf der IntelliJ IDEA-Plattform basiert. Es bietet einen Code-Editor, Debugging-Tools und einen Layout-Editor für die Gestaltung der Benutzeroberfläche. Für iOS-Entwicklung wird Xcode (siehe 2.1.2) eingesetzt, das ähnliche Funktionen bietet und zusätzlich Werkzeuge für die Verwaltung von Zertifikaten und die Veröffentlichung im App Store enthält.

Native Apps nutzen plattformspezifische SDKs: Das Android SDK für Android-Apps und das iOS SDK für iOS-Apps. Diese SDKs bieten umfangreiche Bibliotheken und Tools für die App-Entwicklung (Google 2024b; Apple 2015). Der kompilierte Code wird direkt auf der CPU des Geräts ausgeführt, was zu einer hohen Leistung führt.

| Native Apps | |
|---|---|
| **Vorteile** | • Höchste Leistung und Effizienz<br>• Voller Zugriff auf Gerätefunktionen und APIs<br>• Beste Benutzererfahrung durch Einhaltung plattformspezifischer Design-Richtlinien<br>• Möglichkeit der Offline-Nutzung<br>• Umfangreiche Testmöglichkeiten auf Systemebene |
| **Nachteile** | • Höhere Entwicklungskosten bei Mehrplattform-Entwicklung<br>• Separate Codebases für jede Plattform<br>• Aufwendigere Wartung und Updates<br>• Plattformspezifisches Fachwissen erforderlich |

**Tabelle 2.1:** Vor- und Nachteile der Nativen Apps

### Build Tools

Build Tools sind essenzielle Komponenten im Entwicklungsprozess mobiler Applikationen, insbesondere für native Android-Apps.
Sie automatisieren Kompilierungs-, Test- und Packaging-Prozesse sowie das Dependency Management (Abhängigkeiten zwischen verwendeten Bibliotheken) (Pei Liu u. a. 2024).

Zwei prominente Build Tools in der Android-Entwicklung sind Maven und Gradle (Pei Liu u. a. 2024).

Maven[14] verwendet ein XML-basiertes Projektmodell und folgt dem Prinzip „Convention over Configuration“. Dies bedeutet, dass Maven vordefinierte Standardeinstellungen und Strukturen für Projekte verwendet, wodurch der Konfigurationsaufwand reduziert wird. Entwickler müssen nur Abweichungen von diesen Standards explizit definieren.

Das XML-basierte Projektmodell, bekannt als Project Object Model (POM), beschreibt die Projektstruktur, Abhängigkeiten und Build-Prozesse in einer standardisierten Form, was die Projektverwaltung und -verständlichkeit erhöht. Der in diesser Arbeit umgesetzte Demonstrator verwendet Maven, weswegen nicht weiter auf Gradle eingegangen wird.

Sowohl Maven als auch Gradle unterstützen:

- Automatisierung des Build-Prozesses.
- Abhängigkeitsmanagement (Verwaltung und Integration von externen Bibliotheken und Modulen).
- Test-Framework-Integration.

Build Tools spielen eine wichtige Rolle im Entwicklungs- und Testprozess. Im Kontext dieser Arbeit können diese Tools genutzt werden, um den Testprozess zu automatisieren und in bestehende Entwicklungsworkflows zu integrieren.
Das zu entwickelnde Framework könnte diese Tools erweitern oder mit ihnen interagieren, um beispielsweise automatisierte Tests in den Build-Prozess zu integrieren oder Informationen über durchgeführte Tests zu extrahieren.

[14] Apache Maven: `https://maven.apache.org/`

### 2.2.2 Mobile Web-Apps

Mobile Web-Apps sind für mobile Geräte optimierte Websites, die mit Webtechnologien wie HTML5, CSS3 und JavaScript entwickelt werden (Delia u. a. 2015)

Sie nutzen responsive Design-Techniken[15], um sich an verschiedene Bildschirmgrößen anzupassen. Web-Apps laufen in einem Webbrowser und erfordern keine Installation auf dem Gerät. Sie werden auf einem Webserver gehostet und über den Browser des mobilen Geräts aufgerufen, was im Vergleich zu nativen Apps zu Leistungseinbußen führen kann.

Durch diesen Ansatz gehen auch einige Funktionen gegenüber nativen Apps verloren. Um diesen Nachteil teilweise auszugleichen, können Web-APIs wie Geolocation und Web Storage genutzt werden. Moderne Web-Apps können auch als Progressive Web Apps (PWAs) entwickelt werden, die einige Funktionen nativer Apps nachahmen, wie Offline-Funktionalität und Push-Benachrichtigungen. Für PWAs werden Technologien wie Service Workers eingesetzt, die Hintergrundskripte ausführen und Netzwerkanfragen abfangen können.

Beliebte Frameworks für die Entwicklung von Web-Apps sind React[16], Angular[17], und Vue.js[18]. Diese Frameworks erleichtern die Erstellung reaktiver Benutzeroberflächen.

[15] Responsive Design beschreibt einen Gestaltungsansatz, bei dem sich Layouts und Inhalte automatisch an die Eigenschaften des Ausgabegeräts wie Bildschirmgröße, Auflösung und Ausrichtung anpassen (`https://alistapart.com/article/responsive-web-design/`)

[16] React: `https://reactjs.org/`

[17] Angular: `https://angular.io/`

[18] Vue.js: `https://vuejs.org/`

| Mobile Web-Apps | |
|---|---|
| **Vorteile** | • Plattformunabhängige Entwicklung<br>• Einfache Wartung und Updates<br>• Keine Installation erforderlich<br>• Geringere Entwicklungskosten<br>• Schnelle Bereitstellung von Updates |
| **Nachteile** | • Eingeschränkter Zugriff auf Gerätefunktionen<br>• Abhängigkeit von Internetverbindung<br>• Möglicherweise geringere Leistung im Vergleich zu nativen Apps<br>• Begrenzte Möglichkeiten für tiefgreifende Systemtests |

**Tabelle 2.2:** Vor- und Nachteile der Mobilen Web-Apps

### 2.2.3 Hybride Apps

Hybride Apps kombinieren Elemente von nativen und Web-Apps. Sie werden mit Webtechnologien entwickelt, aber in einem nativen Container ausgeführt, der einen gewissen Zugriff auf Gerätefunktionen ermöglicht (Delia u. a. 2015).

Hybride Apps bestehen aus einem nativen Wrapper, der eine eingebettete WebView enthält. Die eigentliche Anwendungslogik wird in dieser WebView mit HTML, CSS und JavaScript implementiert. Der native Wrapper bietet Plugins, die den Zugriff auf native Gerätefunktionen ermöglichen.

Beliebte Frameworks für die Entwicklung hybrider Apps sind Ionic[19], Apache Cordova[20], und React Native[21].

Diese Frameworks bieten vorgefertigte GUI-Komponenten und Plugins für den Zugriff auf Gerätefunktionen. React Native geht einen Schritt weiter und rendert native GUI-Komponenten, was zu einer besseren Leistung führt.

Der Web-Teil einer hybriden App wird ähnlich wie eine Web-App entwickelt. Der native Container wird für jede Zielplattform separat kompiliert und enthält eine WebView-Komponente, die den Web-Code ausführt. Plugins fungieren als Brücke zwischen dem JavaScript-Code und den nativen APIs des Betriebssystems. Dies ermöglicht den Zugriff auf Funktionen wie die Kamera oder GPS, die normalerweise nicht über einen Browser verfügbar wären.

| **Hybride Apps** | |
|---|---|
| **Vorteile** | • Eine Codebase für mehrere Plattformen<br>• Teilweiser Zugriff auf Gerätefunktionen<br>• Einfachere Entwicklung als bei nativen Apps<br>• Nutzung von Web-Entwicklungskenntnissen |
| **Nachteile** | • Geringere Leistung als native Apps<br>• Eingeschränkter Zugriff auf plattformspezifische Features<br>• Kompromisse bei der Benutzererfahrung<br>• Komplexere Fehlersuche bei plattformspezifischen Problemen |

**Tabelle 2.3:** Vor- und Nachteile der Hybriden Apps

[19] Ionic: `https://ionicframework.com/`
[20] Apache Cordova: `https://cordova.apache.org/`
[21] React Native: `https://reactnative.dev/`

Die Wahl des Entwicklungsansatzes hängt von den Anforderungen der App, der Kritikalität, den verfügbaren Ressourcen und dem gewünschten Benutzererlebnis ab. Wie aus der Tabelle 2.3 mit den Vor- und Nachteilen der verschiedenen Entwicklungsansätze hervorgeht, bleibt für aufwendigere Apps mit hohen Leistungsanforderungen und vielen Funktionen, die native Entwicklung oft die bevorzugte Wahl. Native Apps bieten die notwendige Flexibilität, um Testansätze wie den Einsatz von Large Language Models und Reinforcement Learning zu implementieren. Sie ermöglichen zudem einen tieferen Zugriff auf Systemfunktionen, was für die Durchführung und Bewertung von Tests wichtig ist.

# 3 Grundlagen des Softwaretestens

In diesem Kapitel werden die Grundbegriffe des Softwaretestens erläutert, um ein Fundament für die weitere Arbeit zu schaffen. Zunächst wird der Begriff der Softwarequalität (SQ) eingeführt und definiert. Dies dient als Basis für die Zieldefinition und -abgrenzung der vorliegenden Arbeit.

Es gibt eine breite Palette von Softwarearten, die von komplexen, unternehmensweiten Systemen wie SAP bis hin zu relativ kleinen mobilen Applikationen reichen. Diese verschiedenen Softwaretypen unterscheiden sich nicht nur in ihrer Größe und Komplexität, sondern auch in ihrer Kritikalität. Die Anforderungen an SQ und Testintensität variieren entsprechend stark.

Beispielsweise erfordern Steuerungssysteme in kritischen Infrastrukturen hohe Zuverlässigkeit und Sicherheit, da Fehler katastrophale Folgen haben können. Im Gegensatz dazu kann eine Wetter-App für den durchschnittlichen Benutzer zunächst weniger kritisch sein. Jedoch kann selbst bei scheinbar unkritischen Applikationen der Nutzungskontext die Anforderungen an die SQ erheblich beeinflussen. Eine Wetter-App für Jogger hat andere Qualitätsanforderungen als eine für Segler, bei der ungenaue oder fehlerhafte Informationen zu gefährlichen Situationen führen könnten.

Basierend auf dem Konzept der SQ werden anschließend Softwarefehler definiert und in den Kontext der Qualitätssicherung eingeordnet. Die Anwesenheit von Softwarefehlern kann direkt die Qualität der Software beeinflussen. Eine Klassifikation von Softwarefehlern wird vorgestellt, wobei der Fokus auf jenen Fehlertypen liegt, die für diese Arbeit besonders relevant sind. Diese Priorisierung ermöglicht eine gezielte Ausrichtung der späteren Teststrategien.

Aufbauend auf diesen grundlegenden Konzepten werden gängige Softwaretestverfahren erläutert. Besondere Aufmerksamkeit wird dabei auf Testmethoden für mobile Applikationen gelegt, da diese den Schwerpunkt der vorliegenden Arbeit bilden. Die Darstellung dieser Verfahren berücksichtigt die zuvor etablierten Qualitätsziele und Fehlerklassifikationen.

Durch diese strukturierte Herangehensweise wird ein umfassendes Verständnis der Grundlagen des Softwaretestens vermittelt. Dies bildet die notwendige Grundlage für die nachfolgenden Kapitel, in denen Testmethoden unter Einsatz von Large Language Models und Reinforcement Learning für mobile Applikationen entwickelt und evaluiert werden.

## 3.1 Softwarequalität

Die Sicherstellung einer angemessenen Softwarequalität (SQ) ist wichtig in der Softwareentwicklung und hängt von den Qualitätsanforderungen der entwickelten Software ab. Um die Qualität von Software systematisch zu bewerten und zu verbessern, ist ein differenziertes Verständnis des Begriffs „Softwarequalität (SQ)“ und der verschiedenen Ansätze zu dessen Definition unerlässlich.

Der Begriff SQ beschreibt die Gesamtheit der Eigenschaften und Merkmale einer Software, die deren Fähigkeit determiniert, spezifizierte Anforderungen zu erfüllen und kategorisiert somit deren Qualität (Liggesmeyer 2009).

Zur Erläuterung von SQ existieren verschiedene Definitionen, die unterschiedliche Schwerpunkte setzen.

Die Norm ISO/IEC/IEEE 24765:2017, Abschnittsnummer: 3.3835 verweist auf mehrere verwandte Definitionen von SQ aus verschiedenen Normen ISO/IEC 24000:2014; ISO/IEC 25010:2023; IEEE 730:2014. Diese Definitionen bieten komplementäre Perspektiven:

1. „*capability* of software product[sic] to satisfy stated and implied needs when used under specified conditions [ISO/IEC 25000:2014 Systems and software Engineering — Systems and software product Quality Requirements and Evaluation (SQuaRE) (siehe Abschnitt 3.1.1 Anm. Autor) — Guide to SQuaRE, 4.33]

2. *degree* to which a software product satisfies stated and implied needs when used under specified conditions [ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, 4.3.13]

3. *degree* to which a software product meets established *requirements* [IEEE 730-2014 IEEE Standard for Software Quality Assurance Processes, 3.2][Hervorhebungen sind vom Autor hinzugefügt, Grammatikfehler wurden übernommen]“

Diese komplementären Definitionen verdeutlichen die Vielschichtigkeit von SQ und ihrer Bewertung. Sie ergänzen sich gegenseitig und bieten verschiedene Perspektiven auf das Konzept der SQ, anstatt eine einzelne, umfassende Definition zu liefern. Jede Definition betont unterschiedliche Aspekte, die bei der Bewertung von SQ berücksichtigt werden sollten. Die Bewertung der SQ erfolgt dabei auf einem Kontinuum. Während die Vorstellung von *niedriger* bis *hoher* Qualität intuitiv ist, ist die präzise Definition und Messung dieser Abstufungen komplex und oft kontextabhängig. Sie erfordert die Berücksichtigung vielfältiger Faktoren und kann je nach Anwendungsfall und Benutzer-Perspektive variieren.

Die Definitionen lassen sich in zwei Hauptkategorien einteilen, die sich in ihrem Fokus unterscheiden, aber nicht unbedingt gegensätzlich sind:

- **Fokus auf Benutzerbedürfnisse**: Die ersten beiden Definitionen, aus den Normen ISO/IEC 24000:2014 und ISO/IEC 25010:2023, stellen die Benutzerbedürfnisse in den Vordergrund. Sie betonen die Fähigkeit der Software, sowohl explizit formulierte als auch implizite Bedürfnisse bei der Nutzung zu erfüllen. Dies berücksichtigt, dass nicht alle Benutzerbedürfnisse formell als Anforderungen dokumentiert sein müssen.
- **Fokus auf Anforderungen**: Die dritte Definition der IEEE 730:2014 legt den Schwerpunkt auf die Erfüllung etablierter Anforderungen. Hierbei wird davon ausgegangen, dass die relevanten Benutzerbedürfnisse, Wünsche und Erwartungen im Vorfeld als formelle Anforderungen erfasst wurden.

Idealerweise sollten die Benutzerbedürfnisse und Anforderungen übereinstimmen. In der Praxis können jedoch implizite oder sich ändernde Bedürfnisse existieren, die nicht vollständig in den formalen, expliziten Anforderungen erfasst sind.

Explizite Bedürfnisse werden von den Entwicklern der Software direkt formuliert, beispielsweise in Form von Anforderungen oder Spezifikationen, und sollten explizit dokumentiert werden. Im Gegensatz dazu werden implizite Bedürfnisse nicht ausdrücklich von den Benutzern geäußert, sind jedoch dennoch vorhanden. Diese Bedürfnisse ergeben sich aus dem Nutzungskontext sowie den Erwartungen der Benutzer und basieren auf individuellen Wahrnehmungen, Vorlieben und Erfahrungen. Sie sind oft schwer zu messen und können von Person zu Person, bzw. Sicht der betrachtenden Person, stark variieren (Liggesmeyer 2009, S. 461).

Die Berücksichtigung impliziter und expliziter Bedürfnisse variiert je nach Kritikalität und Anwendungskontext der Software. Bei hochkritischen Systemen liegt der Fokus primär auf expliziten, klar definierten Anforderungen, um hohe Sicherheit und Zuverlässigkeit zu gewährleisten. Bei weniger kritischen Applikationen, insbesondere mobilen Applikationen, gewinnen implizite Bedürfnisse an Bedeutung, da das Benutzererlebnis hier zentral ist.

Mobile Applikationen werden in vielfältigen Kontexten genutzt - von geschäftlichen Applikationen bis hin zu Unterhaltung und sozialer Interaktion. Diese Vielfalt sowie die intuitive, unmittelbare Interaktion mit mobilen Geräten erfordern eine nuancierte Berücksichtigung impliziter Erwartungen und Bedürfnisse. Das Beispiel der Wetter-App für Jogger versus Segler verdeutlicht, wie der genaue Nutzungskontext die Anforderungen an die SQ beeinflusst.

Die Gewichtung expliziter und impliziter Bedürfnisse ist ein komplexer, dynamischer Prozess, der sich im Laufe des Produktlebenszyklus ändern kann und durch die Entwickler vorgenommen werden sollte. Die Sicherstellung und Verbesserung der SQ erstreckt sich über den gesamten Lebenszyklus der Software - von der Entwicklung über den Einsatz bis zur eventuellen Ablösung. Dabei spielen verschiedene Vorgehensmodelle der Softwareentwicklung eine wichtige Rolle in der Herangehensweise an SQ:

- In agilen Modellen wird SQ kontinuierlich in kurzen Iterationen adressiert (Baumgartner u. a. 2023).
- Wasserfall-Modelle sehen oft dedizierte Phasen für Qualitätssicherung vor (Liggesmeyer 2009).
- DevOps-Ansätze integrieren SQ eng in den gesamten Entwicklungs- und Betriebsprozess (Céspedes u. a. 2020).

Um die Qualitätseigenschaften einer Software über ihren gesamten Lebenszyklus hinweg strukturiert und einheitlich zu bewerten, zu verbessern und objektiv zu messen, bietet die ISO/IEC 24000:2014-Normenreihe (SQuaRE) einen umfassenden Rahmen. Dieser ermöglicht es, SQ aktiv zu managen und kontinuierlich zu verbessern.

### 3.1.1 SQuaRE

Die vorliegende Arbeit stützt sich auf die ISO/IEC 25000-Normenreihe, auch Qualitätskriterien und Bewertung von System- und Softwareprodukten (SQuaRE) genannt, als Grundlage für Definitionen und Konzepte der SQ. Diese Wahl basiert auf der internationalen Anerkennung und Standardisierung von SQuaRE, die einen konsistenten und vergleichbaren Ansatz zur Qualitätsbewertung ermöglicht. Durch die Orientierung an SQuaRE wird ein strukturierter, normenkonformer Aufbau des Testframeworks für mobile Applikationen sichergestellt, was die Vergleichbarkeit und Übertragbarkeit der Ergebnisse fördert.

Diese Normenreihe erweitert und ersetzt die frühere ISO/IEC 9126:2011 und gliedert sich in verschiedene Normengruppen, die jeweils spezifische Aspekte bei der Umsetzung der SQ ISO/IEC 24000:2014 adressieren. Dabei bietet die *SQuaRE*-Normenreihe nur einen Rahmen, welcher anwendungsspezifisch angepasst und verwendet werden kann.

Im Kontext dieser Arbeit sind folgende Normen von besonderer Bedeutung:

- ISO/IEC 25010:2023 (Qualitätsmodell): Definiert die Qualitätsmerkmale von Software.
- ISO/IEC 24020:2019 (Messrahmenwerk): Definiert Leitlinien für die Entwicklung und Anwendung von Qualitätsmaßen.
- ISO/IEC 25022:2016 (Messung der Qualität im Einsatz): Definiert die Kriterien zur Bewertung der SQ aus Benutzerperspektive.
- ISO/IEC 25023:2016 (Messung der System- und SQ): Definiert Metriken für die technischen Aspekte der Software.
- ISO/IEC 25024:2015 (Messung der Datenqualität): Definiert die Qualitätsmerkmale von Daten.
- ISO/IEC 25030:2019 (Qualitätsanforderungen): Definiert die Spezifikation von Qualitätsanforderungen.
- ISO/IEC 25040:2011 (Evaluationsprozess): Definiert einen strukturierten Prozess zur Evaluation der SQ.

Als grundlegendes Rahmenwerk der gesamten Normenreihe dient die ISO/IEC 25040:2011. Diese Normgruppe beschreibt den vollständigen Evaluationsprozess sowie die notwendigen Schritte zur Bewertung der SQ. Evaluation im Kontext der SQ wird in der Norm als systematische Untersuchung zur Bestimmung des Ausmaßes definiert, in dem eine Softwarekomponente oder eine Software spezifizierte oder implizierte Kriterien erfüllt ISO/IEC 25040:2011. Abbildung A.2 stellt diesen Zusammenhang noch einmal genauer dar.

Der Evaluationsprozess gliedert sich in fünf Hauptphasen und ist in 3.1 dargestellt (ISO und IEC 2011a). Eine ausführlichere Darstellung ist in Anhang A.2 abgebildet.

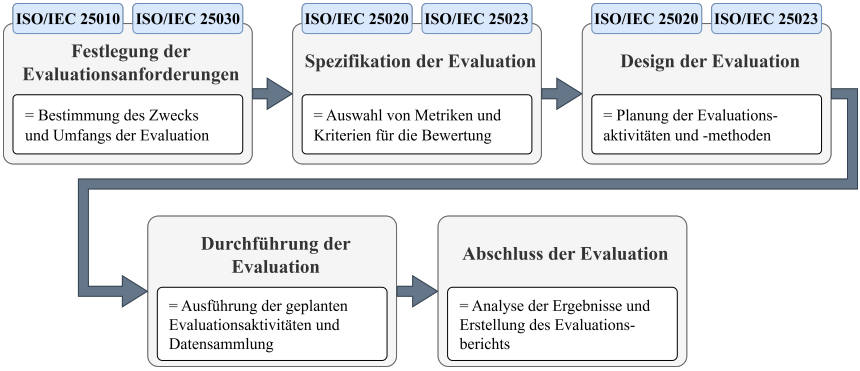



**Abbildung 3.1:** Zusammenhang der ISO 25000-Normenreihe mit ISO/IEC 25040:2011 als Basis (vgl. ISO/IEC 25040:2011)

In der Phase der Festlegung der Evaluationsanforderungen spielen die Normengruppen ISO/IEC 25010:2023 (Qualitätsmodell) und ISO/IEC 25030:2019 (Qualitätsanforderungen) eine zentrale Rolle. Die ISO/IEC 25010:2023 definiert acht Hauptqualitätsmerkmale für Software, darunter funktionale Eignung, Zuverlässigkeit und Benutzbarkeit.

Die **Funktionale Eignung** beschreibt den Grad, in dem eine Software die spezifizierten Funktionen erfüllt und die gewünschten Ergebnisse erzielt. Dies umfasst Aspekte wie Korrektheit, Vollständigkeit und Angemessenheit der Funktionen.

Die **Nicht-funktionale Eignung** bezieht sich auf Qualitätsmerkmale, die nicht direkt mit der Funktionalität verbunden sind, sondern das Verhalten und die Eigenschaften der Software beschreiben. Beispiele hierfür sind Leistung, Sicherheit, Benutzbarkeit, Zuverlässigkeit, Wartbarkeit und Portabilität.

Diese Qualitätsmerkmale bilden die Grundlage für die Definition konkreter Qualitätsanforderungen im Rahmen der ISO/IEC 25030:2019. Basierend auf diesen Normen können präzise Qualitätsanforderungen für Prüftechniken formuliert werden, welche die definierten Anforderungen der Software testen.

In der Spezifikations- und Designphase kommen die Normengruppen ISO/IEC 24020:2019 (Messrahmenwerk) und ISO/IEC 25023:2016 (Messung der System- und SQ) zum Einsatz.

Die ISO/IEC 24020:2019 bietet Leitlinien für die Entwicklung und Anwendung von Qualitätsmaßen, während die ISO/IEC 25023:2016 konkrete Metriken für die verschiedenen Qualitätsmerkmale definiert. Diese Normen ermöglichen eine anwendungsspezifische Anpassung der Evaluation in den entsprechenden Phasen.

Für die in dieser Arbeit betrachteten KI-gestützten Prüftechniken ist zusätzlich die ISO/IEC 25024:2015 relevant. Diese Norm stellt konkrete Metriken zur Bewertung der Datenqualität bereit, was im Kontext des KI-unterstützten Testens besonders wichtig ist und angewendet werden kann (Nakajima und Nakatani 2021). Sie ermöglicht eine fundierte Bewertung sowohl der Qualität von KI-Trainingsdaten als auch der erzielten Testergebnisse.

Die Beachtung der SQuaRE-Normenreihe bietet bei der Erstellung eines Testframeworks Vorteile für den Aufbau und die Struktur. Im Kontext dieser Arbeit ergeben sich folgende Vorteile für KI-gestützte Testframeworks:

1. Standardisierte Qualitätsdefinition: Durch die Verwendung des in ISO/IEC 25010:2023 definierten Qualitätsmodells wird eine umfangreiche und einheitliche Basis für die Beurteilung von SQ geschaffen.

2. Messbare Qualitätsziele: Die in ISO/IEC 25023:2016 definierten Metriken ermöglichen eine objektive Bewertung der SQ und der Effektivität der Prüftechniken bezogen auf bestimmte Anforderungen.

3. Datenqualität: Die Berücksichtigung der ISO/IEC 25012:2008 stellt sicher, dass die für die KI-Modelle verwendeten Daten von hoher Qualität sind, was die Zuverlässigkeit der Testergebnisse erhöht.

4. Strukturierter Evaluationsprozess: Die Anwendung des in ISO/IEC 25040:2011 definierten Prozesses gewährleistet eine systematische und nachvollziehbare Qualitätsbewertung.

5. Kontinuierliche Verbesserung: Durch die Integration von KI-gestützten Prüftechniken in den SQuaRE-Rahmen können Qualitätsprobleme frühzeitig erkannt und behoben werden.

Diese strukturierte und normenkonforme Herangehensweise ermöglicht es, ein Framework für KI-gestützte Tests systematisch aufzubauen und die Ergebnisse einheitlich zu bewerten. Sie bildet zudem die Grundlage für die nachfolgende Betrachtung konkreter Softwarefehler und deren Auswirkungen auf die Qualität mobiler Applikationen.

### 3.1.2 Softwarefehler

Die Gewährleistung einer hohen SQ steht in direktem Zusammenhang mit der Identifikation und Behebung von Softwarefehlern.
(Balzert 1998) definiert einen Fehler als jegliche Abweichung und Inkonsistenz von den Anforderungen des Auftraggebers, während (Andreas Spillner und Tilo Linz 2021) dies als „[...] die Nichterfüllung einer festgelegten Anforderung, eine Abweichung zwischen dem → Istverhalten (während der Ausführung der Tests oder des Betriebs festgestellt) und dem → Sollverhalten (in der Spezifikation oder den Anforderungen festgelegt)“ präzisiert.

Die ISO/IEC/IEEE 24765:2017 unterscheidet folgende Fehlerbezeichnungen:

- **Fehler (Fault):** Eine Anomalie im Code oder in der Systemstruktur, die bei Ausführung zu einem Defekt oder Ausfall führen kann. Ein Fehler ist die *technische Ursache* für einen Defekt. Dies kann ein inkorrekter Schritt, Prozess oder eine fehlerhafte Datendefinition in einem Computerprogramm sein ISO/IEC 25040:2011.

- **Defekt (Defect):** Ein Mangel in einem Arbeitsprodukt, der nicht den *Anforderungen* oder *Spezifikationen* entspricht IEEE 1044:2009. Ein Defekt kann durch einen Fehler im Code oder im Design entstehen. Es ist die direkte Auswirkung eines Fehlers.
- **Störung (Error):** Ein fehlerhafter Zustand des Systems, der durch einen Fehler im Code oder im Design verursacht wird. Kann durch eine *menschliche Eingabe* ausgelöst sein IEEE 1044:2009. Es ist die unmittelbare Auswirkung eines Defekts und kann zu einem Ausfall führen.
- **Ausfall (Failure):** Die *sichtbare Auswirkung* eines Fehlers, bei der das System seine Funktion nicht mehr wie vorgesehen erfüllt. Es kann eine erforderliche Funktion nicht mehr ausführen, oder innerhalb zuvor spezifizierter Grenzen funktionieren ISO/IEC/IEEE 15026-1:2019. Es ist die letzte Konsequenz eines Fehlers und kann zu einem kompletten Systemausfall führen.
- **Problem:** Ein allgemeiner Begriff für eine Schwierigkeit oder ein unerwünschtes Ereignis, das eine Untersuchung und Korrekturmaßnahmen erfordert ISO/IEC/IEEE 24748-1:2024. Es ist die oberste Ebene und kann verschiedene Ursachen haben, darunter Fehler, Defekte, Störungen und Ausfälle.

Sie zeigt, dass ein Problem, das während der Anwendung der Software auftritt, oft auf einen Fehler zurückzuführen ist. Diese Beziehungen sind in Abbildung 3.2 dargestellt. Der Fehler kann entweder ein Defekt im Code, oder eine Störung sein, die durch eine Benutzereingabe herbeigeführt wurde. Beides kann zu Ausfällen der Software führen.
Softwarefehler sind somit Abweichungen vom spezifizierten, erwarteten oder beabsichtigten Verhalten einer Software.

Diese Definition umfasst, wie die bereits erwähnte Definition der SQ in 3.1, sowohl explizite als auch implizite Anforderungen, wobei explizite Anforderungen in Spezifikationsdokumenten festgehalten sind und implizite Anforderungen oft aus dem Nutzungskontext oder allgemeinen Qualitätserwartungen resultieren.

Die Beziehung zwischen Fehlern und erfüllten Anforderungen lässt sich wie folgt beschreiben:

- Je mehr Fehler oder Defekte in einem System vorhanden sind, desto weniger Anforderungen können vollständig erfüllt werden.
- Umgekehrt gilt: Je mehr Anforderungen vollständig erfüllt sind, desto weniger Fehler oder Defekte sollten im System vorhanden sein.

Zur effektiven Fehlererkennung und -behandlung müssen die Fehler während und nach dem Entwicklungsprozess in Fehlerprotokollen festgehalten werden.

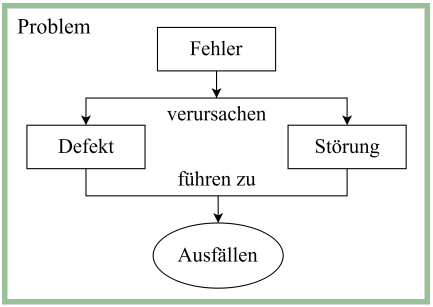



**Abbildung 3.2:** Beziehung zwischen den Fehlerbezeichnungen (vgl. IEEE 1044:2009)

#### 3.1.2.1 Kategorisierung von Softwarefehlern

Die Kategorisierung von Softwarefehlern ist ein wichtiger Aspekt eines Testprozesses. Sie ermöglicht es Fehler im Rahmen eines Testprozesses zu erfassen und diese basierend auf den anwendungsspezifischen SQ-Anforderungen systematisch zu kategorisieren. Ein Testprozess beschreibt die systematische und standardisierte Abfolge von Aktivitäten zur Qualitätssicherung von Software ISO/IEC/IEEE 29119-2:2021.

Basierend auf der Arbeit von Z. Li u. a. (2006) und in Anlehnung an die ISO/IEC 25010:2023 lässt sich eine mehrdimensionale Kategorisierung vornehmen:

**Schweregrad:** Kritisch, Schwerwiegend, Moderat, Gering

**Ursprung:** Anforderungen, Design, Implementierung, Dokumentation

**Betroffene Qualitätsdimension:**

- Funktionale Fehler: Beeinträchtigen die funktionale Eignung der Software.
- Leistungsfehler: Wirken sich auf die Leistungseffizienz aus. Die Leistungseffizienz ist hierbei nach ISO/IEC 25010:2023 definiert als die Fähigkeit eines Produkts, seine Funktionen innerhalb einer spezifizierten Zeit auszuführen.
- Kompatibilitätsfehler: Beeinflussen die Interoperabilität oder Ko-Existenz mit anderen Systemen.
- Benutzbarkeits- und Oberflächenfehler (UI): Beeinträchtigen die Bedienbarkeit und Benutzererlebnis.
- Zuverlässigkeitsfehler: Betreffen die Verfügbarkeit und Fehlertoleranz des Systems.

- Sicherheitsfehler: Gefährden die Vertraulichkeit, Integrität oder Authentizität.
- Wartbarkeitsfehler: Erschweren die Modifizierbarkeit oder Testbarkeit der Software.
- Portabilitätsfehler: Behindern die Anpassungsfähigkeit oder Installierbarkeit auf verschiedenen Plattformen.

Durch die Integration der SQ-Anforderungen in die Kategorisierung können Testanforderungen identifiziert werden. Dies hilft dabei zu definieren, welche Fehler getestet werden müssen, um diese anschließend beheben zu können und folglich die Gesamtqualität der Software zu gewährleisten.

Jeder Fehler kann potenziell eines oder mehrere dieser Merkmale beeinträchtigen und somit die Gesamtqualität der Software mindern.

Im Rahmen dieser Arbeit werden jedoch bestimmte Einschränkungen vorgenommen, um den Fokus auf folgende Aspekte des Softwaretestens zu legen:

- Es wird ausschließlich nicht-kritische Software betrachtet, womit sicherheitskritische Applikationen wie beispielsweise Banking-Apps ausgeschlossen werden.
- Es werden vor allem implizite Anforderungsfehler betrachtet, wie beispielsweise das unerwartete Abstürzen oder Einfrieren einer Applikation.
- Sicherheitsfehler, Wartbarkeitsfehler und Portabilitätsfehler werden nicht explizit untersucht.
- Leistungstests werden nur durchgeführt, wenn Leistungsprobleme implizit während des Testprozesses auffallen.
- Dokumentations- und Designfehler liegen außerhalb des Betrachtungsbereichs dieser Arbeit.

Diese Einschränkungen ermöglichen eine fokussierte Untersuchung der Fehlertypen und Testmethoden, die für die Qualitätssicherung von nichtkritischen mobilen Applikationen besonders relevant sind.

#### 3.1.2.2 Spezifische Fehlerausprägungen bei mobilen Applikationen

Der typische Lebenszyklus eines Softwarefehlers umfasst die Phasen *Entstehung*, *Entdeckung*, *Analyse*, *Behebung*, *Verifikation* und *Schließung* (angelehnt an (IEEE 2010)). Dieser Zyklus bildet ein Rahmenwerk für den Testprozess und unterstreicht die Bedeutung einer systematischen Herangehensweise an die Qualitätssicherung.

Jedoch ist es bei der Entwicklung eines Testframeworks für mobile Applikationen wichtig, die charakteristischen Eigenschaften zu berücksichtigen (siehe Kapitel 2). Muccini, Di Francesco und Esposito (2012) betonen die Notwendigkeit neuer Testansätze, die explizit auf diese Herausforderungen eingehen, wie etwa:

- Kontextbasierte Testfallauswahl: Überdeckung verschiedener Kontextszenarien. Diese Szenarien werden typischerweise von Entwicklern und Testern basierend auf der Anwendungsspezifikation, Benutzerszenarien und möglichen Umgebungsbedingungen definiert.
- GUI Tests (auch UI-Tests): Überprüfung der Benutzeroberfläche unter unterschiedlichen Bedingungen. Die Testbedingungen werden üblicherweise vom UI/UX-Team in Zusammenarbeit mit Entwicklern und Qualitätssicherungsexperten festgelegt, basierend auf Designrichtlinien und erwarteten Nutzungsmustern.

In Kombination mit dem KI-basierten Kontext des Testframeworks können die verschiedenen Fehlertypen mit unterschiedlichen Ansätzen identifiziert werden, auf die in Abschnitt 3.2.1.2 noch genauer eingegangen wird.

- **White-Box-Tests mit hoher Testüberdeckung:** Effektiv bei der Identifizierung von funktionalen Fehlern, Leistungsfehlern und Sicherheitsfehlern auf Code-Ebene.
- **Black-Box-Tests (insbesondere mit Reinforcement Learning):** Effektiv für die Erkennung von Benutzbarkeits- und GUI-Fehlern, Kompatibilitätsfehlern und kontextabhängigen Fehlern.

Die Kombination beider Ansätze ermöglicht eine umfassendere Fehlererkennung, die sowohl die interne Struktur der Software als auch ihr Verhalten aus Benutzersicht miteinbezieht und somit die Gesamtqualität der Software berücksichtigt.

Die enge Verbindung zwischen SQ und Prüftechniken bildet das Fundament effektiver Qualitätssicherung in der Softwareentwicklung. Die Kategorisierung von Softwarefehlern nach Fehlerart und ihrer Auswirkung hilft, die Anforderungen zu identifizieren, die ein Test überprüfen soll. Das Ziel ist es, zu definieren, „was“ getestet werden muss.

## 3.2 Softwaretests

Innerhalb dieses Testprozesses kommen verschiedene Testmethoden zum Einsatz. Unter Testmethoden versteht man systematische Vorgehensweisen und Techniken, die zur Durchführung von Tests angewandt werden. Sie umfassen sowohl traditionelle Ansätze wie manuelle Tests und automatisierte Skripte als auch fortschrittliche Techniken wie modellbasiertes Testen oder KI-gestützte Verfahren. Die Wahl der geeigneten Testmethode hängt von Faktoren wie der Art der zu testenden Software, den Qualitätsanforderungen und den verfügbaren Ressourcen ab.

Softwaretests tragen als primäres Instrument zur Identifikation von Softwarefehlern (siehe Abschnitt 3.1.2) maßgeblich zur Steigerung der SQ bei. Sie dienen im Rahmen eines strukturierten *Testprozesses* der Qualitätssicherung und Zuverlässigkeitsverbesserung von Software. Der Testprozess, definiert nach ISO/IEC/IEEE 29119-1:2022, umfasst alle systematischen Aktivitäten, die zur Überprüfung und Bewertung einer Software durchgeführt werden. Innerhalb dieses Testprozesses kommen verschiedene Testmethoden zum Einsatz. Unter Testmethoden versteht man systematische Vorgehensweisen und Techniken, die zur Durchführung von Tests angewandt werden.

In den folgenden Abschnitten werden die fundamentalen Konzepte und Klassifikationen des Softwaretestens erörtert, wobei ein besonderer Schwerpunkt auf moderne Ansätze wie KI-gestützte Prüftechniken gelegt wird.

Das Hauptziel/Hauptfunktion eines Softwaretests ist, die in Abschnitt 3.1.2 genannten Fehlertypen, zu entdecken um diese beheben zu können (Sommerville 2011).

Die ISO/IEC/IEEE 29119-1:2022 erweitert in der Definition über den Zwecke des Testens (*„purpose of testing“*) die im vorherigen Kapitel diskutierten Fehlerdefinitionen und betrachtet die vielfältigen Rollen, die Tests im Softwareentwicklungsprozess spielen:

- **Erkennen von Defekten:** Dies ermöglicht deren anschließende Beseitigung und erhöht somit die SQ.
- **Sammeln von Informationen über die Software:** Tests generieren Informationen, die für verschiedene Stakeholder nützlich sind:
  - Entwickler können die Informationen nutzen, um Defekte zu beseitigen und die Codequalität zu verbessern.
  - Tester können die Informationen verwenden, um bessere Testfälle zu erstellen.

- **Schaffung von Vertrauen und Entscheidungsfindung:** Durch den Nachweis der korrekten Funktionsweise unter definierten Bedingungen wird das Vertrauen in die Software gestärkt.

Der Testprozess dient nicht nur dazu, die SQ im Nachhinein zu verbessern, sondern auch durch präventive Maßnahmen, was zusätzlich das Vertrauen stärkt. Um dies zu erreichen, ist wie bei anderen qualitätserhaltenden Maßnahmen ein systematisches Vorgehen bei der Erstellung und Durchführung von Tests unerlässlich.

Dabei müssen die Tests „[...] systematisch geplant, durchgeführt, kontrolliert, ausgewertet und dokumentiert werden“ (Liggesmeyer 2009).

Eine strukturierte Dokumentation der Testergebnisse, die Testfallbeschreibungen, Ein- und Ausgabedaten, Umgebungsbedingungen sowie Fehlermeldungen umfasst, ermöglicht die präzise Identifikation fehlerhafter Programmteile. Diese detaillierte Aufzeichnung macht Testergebnisse nachvollziehbar und erleichtert die Reproduktion von Defekten, Störungen oder Ausfällen, was wiederum die Fehleranalyse und -behebung unterstützt.

Es existieren eine Reihe von Softwaretests, die alle die Zwecke des Testens erfüllen, aber unterschiedliche Fehlerarten abdecken. Um die passenden Softwaretests für die entsprechenden Fehlerarten zu wählen, werden im Folgenden die verschiedenen Klassifikationen von Softwaretests vorgestellt.

Dieses Kapitel behandelt verschiedene Aspekte des Softwaretestens, einschließlich der Klassifikation von Tests nach Prüfebene (siehe Abschnitt 3.2.1.4) und Testobjekt (siehe Abschnitt 3.2.1.2). Moderne Ansätze wie modellbasiertes Testen und KI-gestützte Prüftechniken werden im Kontext des gesamten Testprozesses betrachtet.

### 3.2.1 Klassifikationen von Softwaretests

Die Klassifikation von Softwaretests bildet einen wesentlichen Aspekt des systematischen Testprozesses. In diesem Kapitel werden verschiedene Dimensionen der Testklassifikation vorgestellt, die eine gezielte Auswahl und effektive Anwendung geeigneter Testmethoden ermöglichen.

Die Klassifikation umfasst die in Tabelle 3.1 dargestellten Aspekte.

Diese Klassifikation ermöglicht es Tests einzusortieren und einen effektiven, zielgerichteten Testprozess zu gestalten.

| Kriterium | Kategorien | Beschreibung |
|---|---|---|
| Software-Prüftechniken | Statisch | Analyse des Codes ohne Ausführung |
| | Dynamisch | Tests während der Programmausführung |
| Testobjekt | White-Box | Tests mit Kenntnis der internen Struktur |
| | Black-Box | Tests ohne Kenntnis der internen Struktur |
| | Gray-Box | Kombination aus White- und Black-Box |
| Testtechniken und Testreferenzen | Analysierend | Identifikation potenzieller Probleme |
| | Verifizierend | Überprüfung der Korrektheit |
| | Strukturorientiert | Fokus auf Codestruktur |
| | Funktionsorientiert | Fokus auf Funktionalität |
| | Diversifizierend | Vergleich verschiedener Versionen |
| Prüfebene | Unit-Tests | Tests einzelner Komponenten |
| | Integrationstests | Tests der Komponenteninteraktion |
| | Systemtests | Tests des Gesamtsystems |
| | End-to-End-Tests | Tests des kompletten Workflows |
| Testfall | - | Grundlegende Einheit des Testprozesses, die auf den anderen Kriterien aufbaut. Ein Testfall definiert Eingabedaten, Ausführungsbedingungen und erwartete Ergebnisse, um ein bestimmtes Ziel oder eine Testbedingung zu überprüfen. |

**Tabelle 3.1:** Klassifikation von Softwaretests
(Erweiterung zur Arbeit von Andreas Spillner und Tilo Linz (2021))

#### 3.2.1.1 Klassifikation nach Prüftechniken

Softwaretests werden nach Miillerburg (1970), Liggesmeyer (1990) und Liggesmeyer (2009) in zwei Hauptklassen der Software-Prüftechniken unterteilt: *statisch* und *dynamisch*. Diese Unterscheidung ermöglicht es, die verschiedenen **Testansätze** und ihre Anwendungsbereiche sinnvoll einzuteilen.

**Statisches Testen**

Statisches Testen bezeichnet die Überprüfung von Software, ohne den Code auszuführen (Liggesmeyer 2009) und kann daher prinzipbedingt ohne Computerunterstützung erfolgen. Dieser Ansatz zielt darauf ab, potenzielle Fehler, Inkonsistenzen oder Verbesserungsmöglichkeiten im Code frühzeitig zu identifizieren, noch bevor die Software in Betrieb genommen wird. „Potenziell“, da ohne die tatsächliche Ausführung des Codes nicht sicher festgestellt werden kann, ob die identifizierte Codestelle tatsächlich zu einer Störung oder einem Ausfall führt.

Es können keine Aussagen über die vollständige Korrektheit oder Zuverlässigkeit getroffen werden (Liggesmeyer 2009).

Das Testen ohne Ausführung ermöglicht eine gründliche Überprüfung des Codes, was in frühen Entwicklungsphasen von Vorteil ist (Liggesmeyer 2009). Statische Prüftechniken umfassen Techniken wie Code-Reviews und automatisierte statische Codeanalyse. Obwohl prinzipiell ohne Computerunterstützung durchführbar, ist der Einsatz von Werkzeugen in der Praxis oft unerlässlich, insbesondere für die Visualisierung komplexer Codestrukturen, mit vielen Abhängigkeiten.

**Dynamisches Testen**

Dynamisches Testen beinhaltet im Gegensatz zum statischen Testen die tatsächliche Ausführung der Software unter konkreten Testbedingungen. Dieser Ansatz zielt darauf ab, die Funktionalität und Zuverlässigkeit der Software in einem realistischen Umfeld zu überprüfen (Liggesmeyer 2009).

Die charakteristischen Merkmale dynamischer Tests sind (Liggesmeyer 2009):

1. *„Die übersetzte, ausführbare Software wird mit konkreten Eingabewerten versehen und ausgeführt.*
2. *Es kann in der realen Betriebsumgebung getestet werden.*
3. *Dynamische Prüftechniken sind Stichprobenverfahren, da sie nur eine begrenzte Auswahl möglicher Eingaben und Szenarien testen können.*
4. *Dynamische Prüftechniken können die Korrektheit [Anmerkung des Autors: Korrektheit bezeichnet hier die Abwesenheit von Fehlern.] der getesteten Software nicht beweisen.“*

Dynamische Prüftechniken umfassen funktionale Tests, nicht-funktionale Tests, Regressionstests und Lasttests (Myers 1979; Andreas Spillner und Tilo Linz 2021; Sommerville 2011). Jede dieser Kategorien adressiert spezifische Aspekte der SQ. Ein wichtiger Aspekt dynamischer Tests ist, dass sie zwar die Anwesenheit von Fehlern nachweisen, nicht aber deren Abwesenheit garantieren können (Myers, Badgett und C. Sandler 2012). Dies liegt daran, dass es in der Regel unmöglich ist, alle möglichen Eingabekombinationen und Szenarien zu testen.

Die Bedeutung dynamischer Tests in der Softwareentwicklung liegt in ihrer praktischen Anwendbarkeit und der Möglichkeit, das tatsächliche Verhalten der Software unter realen Bedingungen zu beobachten. Sie ergänzen statische Prüftechniken und tragen wesentlich zur Sicherstellung der SQ bei.

#### 3.2.1.2 Klassifikation nach Testobjekt

Die Klassifikation von Softwaretests nach Testobjekt stellt einen weiteren grundlegenden Ansatz in der Testmethode dar. Diese Einteilung basiert auf dem Grad der Kenntnis und der Zugänglichkeit der internen Struktur der zu testenden Software.

Das Testobjekt wird als die Komponente oder das System definiert, das dem Testprozess unterzogen wird. Diese Klassifizierung ist maßgeblich für die Auswahl geeigneter Teststrategien und -techniken, da sie den Fokus und die Tiefe des Testprozesses beeinflusst.

Bei der Auswahl des Testobjekts kann die Frage gestellt werden, „wie und worauf“ getestet werden soll. Es werden dabei folgende Aspekte berücksichtigt ISO/IEC/IEEE 29119-1:2022:

- **Software under Test (Zu testende Software, SUT)** (z.B. generalisierte oder native Android-Apps).
- **Language under Test (Zu testende Programmiersprache, LUT)** (z.B. Dart, Java).
- **Level under Test (Zu testende Testebene, LvlUT)** (z.B. GUI oder Code).
- **Component under Test (Zu testende Komponente, CUT)** (Hardware oder Software).
- **Device under Test (Zu testendes Gerät, DUT)** (konkrete Hardware).

Die Einteilung nach Testobjekt führt zu den drei Hauptkategorien: White-Box-Tests, Black-Box-Tests und Gray-Box-Tests.

**White-Box-Tests**

White-Box-Tests repräsentieren strukturelle Prüftechniken, die auf detaillierten Kenntnissen des Quellcodes basieren (Myers 1979). Diese Methode nutzt die interne Struktur von Systemkomponenten oder des gesamten Systems zur Verifikation eines Programmablaufs. Aufgrund der Kenntnisse über die interne Programmstruktur ermöglichen sie in der Theorie eine vollständige Codeüberdeckung und tragen zur Sicherstellung der Codequalität bei.

Im Kontext von Oberflächen-Tests werden White-Box-Tests verwendet, um Elemente im Quellcode zu identifizieren und mit ihnen zu interagieren. Bei der Anwendung von LLMs für Unit Tests (siehe Abschnitt 3.2.1.4) bieten White-Box-Tests die Möglichkeit, präzise und kontextsensitive Testfälle zu generieren, die eindeutige Codepfade abdecken.

Allerdings besteht die Gefahr, dass Tester Testfälle basierend auf ihrer Codekenntnis statt auf der Spezifikation auswählen, was zu einer unzureichenden Testüberdeckung führen kann. Zudem ist die Prüfebene auf die Code-Ebene beschränkt, wodurch die Benutzerperspektive nur eingeschränkt berücksichtigt wird.

### Black-Box-Tests

Black-Box-Tests konzentrieren sich im Gegensatz zu White-Box-Tests auf Tests, bei denen die interne Programmstruktur nicht bekannt ist oder nicht berücksichtigt wird (Myers 1979). Sie fokussieren sich auf dokumentierte Anforderungen und das erwartete Verhalten der Software. Im Kontext von GUI-Tests ermöglichen Black-Box-Tests eine realitätsnahe Simulation der Benutzerinteraktion, indem sie die Oberfläche eines Device under Test (DUT) einlesen und Testaktionen basierend auf der Oberfläche auswählen. Für RL Agenten bieten Black-Box-Tests einen idealen Rahmen, da sie es ermöglichen, durch Interaktion mit der Benutzeroberfläche Teststrategien zu entwickeln.

Es ist jedoch nicht immer möglich, alle potenziellen Testfälle abzudecken, was eine sorgfältigere Auswahl der Testfälle erfordert, die im Gegensatz zu White-Box-Tests aufwendiger ist. Zudem liefern sie weniger tiefgreifende Informationen über den Code, was das Aufdecken, Beheben und Reproduzieren von Störungen oder Ausfällen auf Code-Ebene erschweren kann.

### Gray-Box-Tests

Gray-Box-Tests repräsentieren einen hybriden Ansatz, der Elemente von White-Box-Tests und Black-Box-Tests kombiniert. Bei dieser Methode verfügen Tester über einige, aber nicht vollständige Kenntnisse der internen Programmstruktur. Das ermöglicht es, tiefgreifende Informationen über den Code zu sammeln und gleichzeitig die Benutzerperspektive zu berücksichtigen.

Diese Klassifikation ermöglicht es Testern, verschiedene Aspekte der Software zu untersuchen - von der detaillierten Code-Struktur bis hin zum Verhalten aus Benutzersicht.

In der Praxis werden oft Kombinationen dieser Ansätze verwendet, um eine umfassende Testüberdeckung zu gewährleisten, insbesondere im Kontext mobiler Applikationen, bei denen sowohl die interne Funktionalität als auch die Benutzerinteraktion relevant sind.

#### 3.2.1.3 Klassifikation nach Prüftechniken und Testreferenzen

Prüftechniken und Testreferenzen bilden zwei zentrale Aspekte in der Klassifikation von Softwaretests. Liggesmeyer (2009) definiert die Testreferenz als das Element, gegen das getestet wird und das zur Beurteilung der Testvollständigkeit und der Korrektheit der Testergebnisse dient. Die Prüftechniken hingegen beschreiben die Methoden und Ansätze zur Durchführung der Tests. Diese Klassifikation ermöglicht eine präzise Zuordnung von Testmethoden und unterstützt die Auswahl geeigneter Techniken für konkrete Testszenarien.

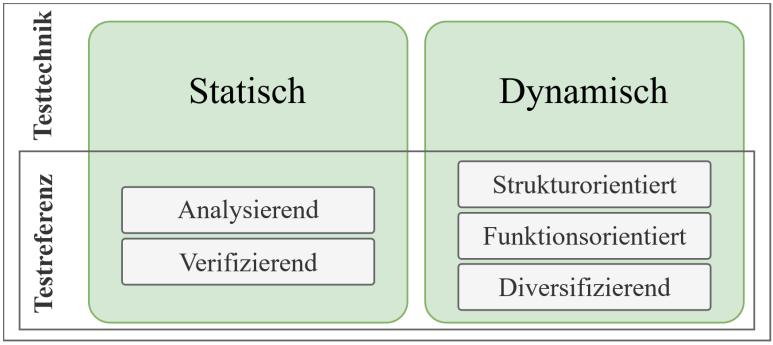



**Abbildung 3.3:** Klassifikation von Prüftechnik und Testreferenzen (vgl. Liggesmeyer 2009)

Wie in Abbildung 3.3 dargestellt, lassen sich die Abschnitt 3.3 gezeigten Prüftechniken im Kontext dieser Klassifikation weiter unterteilen, um zu beschreiben, wie mit der Testreferenz umgegangen wird. Im Anhang A.1 sind die Prüftechniken und Testreferenzen noch einmal detaillierter aufgeführt.

**Analysierende statische Prüftechniken**

Analysierende statische Prüftechniken zielen darauf ab, potenzielle Fehlerquellen und Probleme im Code zu identifizieren, ohne die Software auszuführen. Diese Techniken umfassen:

- **Statische Code-Analyse:** Automatisierte Tools überprüfen den Code auf mögliche Fehlerquellen, Sicherheitslücken und die Einhaltung von Coding-Standards. Diese Werkzeuge können vielschichtige Code-Strukturen systematisch analysieren und bieten einen schnellen Überblick über potenzielle Fehler. Beispiele hierfür sind Lint-Tools oder statische Analyseframeworks.
- **Code-Reviews:** Bei dieser manuellen Überprüfung des Codes durch Entwickler und Experten werden nicht nur Fehler identifiziert, sondern auch Designprobleme und Verbesserungsmöglichkeiten aufgedeckt. Code-Reviews fördern zudem den Wissensaustausch und die Zusammenarbeit im Entwicklerteam.
- **Metriken-basierte Analyse:** Hierbei werden quantitative Maße zur Bewertung der Codequalität verwendet.

Die Testreferenz bei analysierenden Techniken umfasst häufig Coding-Richtlinien, Best Practices und etablierte Qualitätsstandards. Diese dienen als Maßstab, gegen den der Code geprüft wird.

**Verifizierende Prüftechniken**

Verifizierende Prüftechniken konzentrieren sich auf die formale Überprüfung des Codes und die Einhaltung von Spezifikationen. Im Gegensatz zu analysierenden Techniken liegt der Fokus hier auf der mathematischen Beweisführung. Zu den prominenten Techniken zählen:

- **Formale Verifikation:** Diese Technik nutzt mathematische Methoden, um zu bestätigen, dass ein System bestimmte, vordefinierte Eigenschaften erfüllt. Sie basiert auf der Analyse formaler Spezifikationen und dem Einsatz mathematischer Beweise. Formale Verifikation ist besonders effektiv bei der Identifizierung von logischen Inkonsistenzen in frühen Phasen der Softwareentwicklung.
- **Symbolische Modelprüfung:** Diese Methode verwendet symbolische Ausdrücke zur Analyse des Systemverhaltens. Sie ermöglicht es, verschiedene Programmzustände systematisch zu untersuchen und deren Übereinstimmung mit festgelegten Anforderungen zu überprüfen.

Die Testreferenz bei verifizierenden Techniken sind formale Spezifikationen und mathematische Modelle des erwarteten Systemverhaltens. Diese dienen als Grundlage für die Beweisführung und Überprüfung.

Analysierende und verifizierende Testtechniken ergänzen sich in der statischen Analyse. Während analysierende Verfahren praktische Probleme und Qualitätsmängel aufdecken, stellen verifizierende Verfahren die formale Korrektheit und Spezifikationskonformität sicher.

**Strukturorientierte Testtechniken**

Strukturorientierte Testtechniken bilden eine zentrale Kategorie innerhalb der dynamischen Prüftechniken. Diese Techniken nutzen die interne Struktur des Programmcodes als Basis für die Entwicklung und Bewertung von Testfällen. Dabei dient der Quellcode selbst als Testreferenz, was diese Techniken zu einer Form des White-Box-Testens macht (Liggesmeyer 2009).

Im Kern lassen sich strukturorientierte Techniken in zwei Hauptansätze unterteilen:

- **Kontrollflussorientierte Testtechniken** konzentrieren sich auf die Ausführungsreihenfolge von Anweisungen und Verzweigungen im Programmcode. Sie basieren auf der Annahme, dass Fehler häufig in der Kontrollstruktur von Programmen zu finden sind. Die Kontrollstruktur bezieht sich dabei auf die logische Abfolge von Anweisungen, einschließlich Verzweigungen, Schleifen und Funktionsaufrufen. Ziel dieser Techniken ist es, diese Strukturen so umfassend wie möglich durch Tests abzudecken.
- **Datenflussorientierte Testtechniken** hingegen fokussieren sich auf die Verwendung und Manipulation von Daten innerhalb des Programms. Sie verfolgen den „Lebenszyklus“ von Variablen, von ihrer Definition über ihre Verwendung bis hin zu ihrer Löschung, um potenzielle Fehler in der Datenverarbeitung aufzudecken.

Die Bewertung der Testvollständigkeit erfolgt bei strukturorientierten Techniken primär mittels der Metrik der Testüberdeckung. Diese Metrik quantifiziert, wie viel des Quellcodes durch die Testfälle ausgeführt wurde. Aufgrund ihrer zentralen Bedeutung für strukturorientierte Tests wird die Testüberdeckung in Abschnitt 3.2.3 detaillierter erläutert.

Strukturorientierte Testtechniken bieten den Vorteil, dass sie eine systematische und messbare Herangehensweise an das Testen ermöglichen. Sie können Fehler aufdecken, die bei funktionsorientierten Tests möglicherweise unentdeckt bleiben, insbesondere solche, die in selten ausgeführten Codepfaden liegen. Allerdings haben sie auch Limitationen: Sie können nicht die Korrektheit des Programms in Bezug auf seine Spezifikation garantieren und sind möglicherweise nicht in der Lage, Fehler zu finden, die aus einer fehlerhaften Umsetzung der Anforderungen resultieren.

**Funktionsorientierte Testtechniken**

Funktionsorientierte Testtechniken überprüfen die Software gegen ihre funktionalen Spezifikationen, ohne Kenntnis der internen Struktur oder des Quellcodes zu benötigen. Dieser Ansatz entspricht dem Konzept des Black-Box-Testens und testet die Erfüllung der Benutzeranforderungen. Funktionsorientierte Tests werden typischerweise auf verschiedenen Prüfebenen des Softwareentwicklungsprozesses durchgeführt (siehe Abschnitt 3.2.1.4).

Zu den gängigen funktionsorientierten Testtechniken gehören (Liggesmeyer 2009; Andreas Spillner und Tilo Linz 2021):

- Äquivalenzklassenbildung
- Grenzwertanalyse
- Zustandsbasiertes Testen
- Entscheidungstabellentests
- Use-Case-Tests

Diese Techniken helfen, Testfälle zu generieren, die verschiedene Aspekte der Softwarefunktionalität abdecken. Sie sind besonders nützlich, um Randfälle und unerwartete Benutzereingaben zu testen, die bei strukturorientierten Tests möglicherweise übersehen werden. Randfälle sind hierbei Testeingaben an den Rändern einer Funktionsänderung.

Die Testreferenz bei funktionsorientierten Techniken ist die Spezifikation der Software. Während strukturorientierte Techniken die Vollständigkeit anhand der in Abschnitt 3.2.3 beschriebenen Metriken messen, nutzen funktionsorientierte Techniken die Spezifikation als Maßstab zur Beurteilung der Testvollständigkeit und der Korrektheit der Ergebnisse (Liggesmeyer 2009).

**Diversifizierende Testtechniken**

Diversifizierende Testtechniken ermöglichen einen systematischen Vergleich verschiedener Softwareversionen oder Komponenten. Gerade bei mobilen Applikationen, die eine hohe Heterogenität der Betriebssystemversionen auf verschiedenen Geräten aufweisen, sind davon betroffen.

Somit basiert im Gegensatz zu strukturorientierten oder funktionsorientierten Techniken die Beurteilung der Testvollständigkeit bei diversifizierenden Techniken auf dem Vergleich der Ergebnisse verschiedener Softwareversionen oder -implementierungen (Liggesmeyer 2009).

Zu den wichtigsten diversifizierenden Testtechniken gehören:

- **Back-to-Back-Tests:** Bei dieser Technik werden zwei oder mehr Versionen einer Software mit identischen Eingaben getestet und ihre Ausgaben verglichen. Dies ist nützlich, wenn mehrere Implementierungen desselben Algorithmus oder Systems vorliegen, z.B. bei der Entwicklung sicherheitskritischer Systeme.

- **Regressionstests:** Diese Tests überprüfen, ob Änderungen oder Erweiterungen an der Software unbeabsichtigte Auswirkungen auf bereits existierende Funktionalitäten haben. Myers, Badgett und C. Sandler (2012) definiert Regressionstests als „erneutes Testen einer zuvor getesteten Applikation oder eines Systems nach einer Modifikation, um sicherzustellen, dass durch die Änderungen keine neuen Fehler eingeführt oder vorher behobene Fehler nicht reaktiviert wurden."
- **N-Versionen-Programmierung:** Bei dieser Technik werden mehrere unabhängige Implementierungen derselben Spezifikation entwickelt und ihre Ergebnisse verglichen, um Fehler zu identifizieren und die Zuverlässigkeit zu erhöhen.

Diese Verfahren tragen zur Qualitätssicherung in heterogenen Umgebungen bei und sind besonders wichtig, um Unterschiede oder neu eingeführte Fehler zwischen unterschiedlichen Versionen einer Applikation zu erkennen. Insbesondere Regressionstests tragen dazu bei, die Stabilität und Funktionalität über verschiedene Entwicklungsphasen hinweg zu gewährleisten.

In der neueren Softwareentwicklung, vor allem in agilen Umgebungen und in Continuous Integration and Continuous Delivery (CI/CD)-Pipelines, nehmen Regressionstests eine zentrale Stellung ein. Sie ermöglichen es Entwicklungsteams, schnell und häufig Code zu ändern, ohne die Stabilität des Gesamtsystems zu gefährden. Allerdings erfordert die Anwendung diversifizierender Testtechniken oft einen erhöhten Ressourcenaufwand, da mehrere Versionen oder Implementierungen entwickelt und gewartet werden müssen.

Die vorgestellte Klassifikation ermöglicht eine systematische und differenzierte Herangehensweise an den Testprozess. In der praktischen Anwendung werden diese Techniken häufig kombiniert, um eine umfassende Qualitätssicherung zu erreichen.

#### 3.2.1.4 Klassifikation nach Prüfebene

Die Klassifikation nach Prüfebene beschreibt die hierarchische Stufe im Testprozess und ist eng verknüpft mit der Entwicklung und Integration von Hardware- und Softwarekomponenten, während die Testtiefe den Detailgrad der Tests innerhalb einer Ebene angibt. Jede Prüfebene adressiert festgelegte Aspekte des Systems und zielt auf die Identifikation bestimmter Fehlerarten ab.

Eine zentrale Darstellung der Prüfebenen ist die Testpyramide in Abbildung 3.4. Sie verdeutlicht, dass mit steigender Prüfebene die Komplexität, der Aufwand und damit die Ausführungszeit der Tests zunehmen, während ihre Anzahl abnimmt.

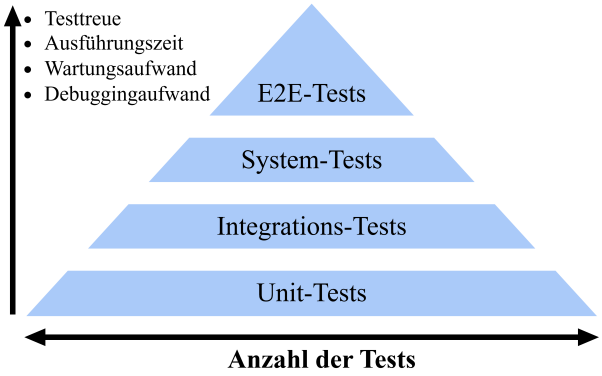



**Abbildung 3.4:** Testpyramide - mit aufsteigendem Aufwand steigen die Testtreue und die Ausführungszeit, daher sinkt die Anzahl der Tests (vgl. Cohn 2010)

Die Basis der Pyramide bilden Unit-Tests, gefolgt von Integrations-, System- und schließlich Ende-zu-Ende-Tests (Cohn 2010). Diese Einteilung ergänzt die zuvor diskutierten Klassifikationen und ermöglicht eine strukturierte Herangehensweise an den Testprozess.

Besonders relevant ist dieser Aufbau bei mobilen Applikationen. Die Komplexität mobiler Geräte und Nutzungsszenarien erfordert eine Balance zwischen den Testebenen. UI-Tests spielen eine große Rolle, um die korrekte Darstellung, Benutzerinteraktion und Performance der App auf unterschiedlichen Geräten sicherzustellen. Gleichzeitig müssen Entwickler die Herausforderungen wie längere Ausführungszeiten und höhere Wartungskosten dieser Tests berücksichtigen. Eine sinnvolle Teststrategie für mobile Applikationen verwendet daher eine begrenzte Anzahl an ausgewählten GUI-Tests, um sowohl Effizienz als auch umfassende Qualitätssicherung zu gewährleisten.

Die Norm ISO/IEC/IEEE 29119-1:2022 definiert die verschiedenen Testebenen in der Softwareentwicklung. Neben Unit- und Integrationstests werden Ende-zu-Ende-Tests in System- und Akzeptanztests unterteilt. Diese detaillierte Klassifizierung ermöglicht eine präzise Auswahl der passenden Testmethoden und -werkzeuge für jede Ebene. So werden beispielsweise für Unit-Tests andere Verfahren eingesetzt als für GUI-Tests. Die *Testtiefe*, das *Testobjekt* und die *Testtechnik* müssen an die jeweilige Prüfebene angepasst werden.

**Unit-Tests**

Unit-Tests (auch Modultests oder Einzeltests genannt) fokussieren sich auf die kleinsten testbaren Einheiten eines Programms, typischerweise einzelne Funktionen oder Methoden (Ludewig und Lichter 2013; ISO, IEC und IEEE 2017). Unit-Tests adressieren primär die Qualitätsmerkmale der funktionalen Eignung und Zuverlässigkeit auf Codeebene.

Das Ziel von Unit-Tests ist es, die Funktion einer Methode mit der funktionalen oder Schnittstellenspezifikation zu vergleichen, die das Modul definiert (Myers, Badgett und C. Sandler 2012). Sie tragen wesentlich zur frühzeitigen Fehlererkennung bei und sollten daher den größten Anteil der erstellten Tests ausmachen.

Bei der Durchführung von Unit-Tests wird jede zu testende Einheit isoliert betrachtet. Eine gängige Methode zur Visualisierung der zu testenden Codestruktur ist der Kontrollflussgraph (Myers, Badgett und C. Sandler 2012, S.24), in dem jede Anweisung als eigener Knoten dargestellt wird (siehe Abschnitt 3.2.1.3). Zur Bewertung der Effektivität dieser Tests dienen Metriken für Testüberdeckung, wie Anweisungsüberdeckung, Zeilenüberdeckung, Zweigüberdeckung und Pfadüberdeckung (siehe Abschnitt 3.2.3).

In Fällen, in denen benötigte Objekte noch nicht existieren oder Systemaufrufe nicht möglich sind, können sogenannte Mock-Objekte verwendet werden. Mock-Objekte sind Funktionen oder Objekte im Code, die eine Funktionalität oder die Rückgabe von Werten simulieren. So ist der Test trotz eigentlich noch vorhandener Lücken ausführbar. In Android beispielsweise ermöglicht das Framework Mockito[22] die Erstellung solcher Objekte.

**Integrationstests (Integration Tests)**

Integrationstests bilden die nächste Stufe in der Testpyramide und fokussieren sich auf das Zusammenspiel verschiedener Hardware-, Softwaremodule oder Komponenten. Sie adressieren primär die Qualitätsmerkmale der Interoperabilität und funktionalen Eignung auf der Ebene von Komponenteninteraktionen.

Integrationstests stellen sicher, dass die durch Unit-Tests geprüften Komponenten auch im Zusammenspiel korrekt funktionieren (Ludewig und Lichter 2013). Dabei werden die in Unit-Tests verwendeten Mock-Objekte durch reale Implementierungen ersetzt. Einzelne Komponenten, die zusammen getestet werden, können jeweils auch wieder aus Komponenten zusammengesetzt sein (Winter u. a. 2013).

[22] Mockito: Framework zur Erstellung von Mock-Objekten (`https://site.mockito.org/`)

IEEE (1990) definiert Integration als den Prozess, bei dem Software- und Hardwarekomponenten zu einem Gesamtsystem kombiniert werden. Im Kontext mobiler Applikationen ist der Aspekt der Interaktion der Soft- mit der Hardware, aufgrund der Herausforderungen, wie beispielsweise Toucheingaben, besonders relevant. Auch bei Interaktion mit Gerätesensoren, Netzwerkschnittstellen oder Datenbanken interagiert die Software eng mit der Hardware des Geräts. Folglich ist das Testen der Integration bei mobilen Applikationen besonders zu berücksichtigen. Frameworks wie Espresso für Android oder XCUITest[23] für iOS ermöglichen die Implementierung automatisierter Integrationstests für mobile Applikationen.

**Systemtests (System Tests)**

Systemtests repräsentieren eine Ebene in der Testpyramide und zielen darauf ab, die vollständige, integrierte Software in einer produktionsnahen Umgebung zu überprüfen (Sommerville 2011). Sie adressieren eine breite Palette von Qualitätsmerkmalen, darunter funktionale Eignung, Leistungseffizienz, Kompatibilität, Benutzbarkeit, Zuverlässigkeit, Sicherheit und Portierbarkeit.

Im Gegensatz zu Unit- und Integrationstests evaluieren Systemtests die Software als Ganzes. Sie umfassen sowohl funktionale als auch nicht-funktionale Aspekte und testen, ob das System unter realen Bedingungen zuverlässig funktioniert.

Somit ermöglichen Systemtests das Testen der Software unter Berücksichtigung der besonderen Herausforderungen im Bereich mobiler Applikationen, wie die Vielfalt der Geräte und Betriebssystemversionen. Weitere Herausforderungen wie Energieverbrauch, Offline-Funktionalität und Integration mit Gerätesensoren sind zu adressieren.

[23] XCUITest: Unterklasse von XCTests, einem Framework zur Testerstellung bei iOS-Applikationen (`https://developer.apple.com/documentation/xctest/user_interface_tests`)

Automatisierungstools wie Appium[24] ermöglichen die Erstellung automatisierter Systemtests, die Benutzerinteraktionen simulieren und das Systemverhalten unter verschiedenen Bedingungen überprüfen können.

Systemtests werden häufig von einer von der Softwareentwicklung unabhängigen Testgruppe durchgeführt, was eine objektive Bewertung der Software gewährleistet. Sie stellen oft den letzten Schritt vor der Softwarefreigabe dar.

**Ende zu Ende Tests (End-to-end-test, E2E)**

Ende zu Ende Tests bilden die Spitze der Testpyramide und evaluieren das System aus der Perspektive des Endbenutzers, wobei sie alle vorherigen Prüfebenen mit einbeziehen. Im Gegensatz zu Systemtests, die sich auf die umfassende Funktionalität des Systems konzentrieren, simulieren E2E-Tests konkrete Benutzerszenarien von Anfang bis Ende.

E2E-Tests behandeln die Applikation als Black-Box und interagieren mit ihr ausschließlich über die Benutzeroberfläche, so wie es ein Endbenutzer tun würde. Sie umfassen alle Schichten der Applikation, von der Benutzeroberfläche über die Geschäftslogik bis hin zu Datenbankinteraktionen und externen Serviceanfragen. Dabei adressieren sie primär die Qualitätsmerkmale der funktionalen Eignung, Benutzbarkeit und Zuverlässigkeit des Gesamtsystems aus Benutzersicht.

[24] Appium: Framework zur Automatisierung von Oberflächennutzungen wie zum Beispiel GUI-Tests (`https://appium.io/docs/en/latest/`)

UI-Tests sind, ebenso wie Systemtests, ein integraler Bestandteil von E2E-Tests, gehen aber über isolierte GUI-Funktionalitätstests hinaus. Bei E2E-Tests werden GUI-Interaktionen im Kontext vollständiger Benutzerszenarien durchgeführt, um sicherzustellen, dass alle Komponenten nahtlos zusammenarbeiten und keine Ausfälle bei der Kombination aller Komponenten einer Applikation auftreten. Im Kontext mobiler Applikationen müssen E2E-Tests, ebenso wie bei Systemtests, die Vielfalt der Geräte, Betriebssysteme und insbesondere der Netzwerkbedingungen berücksichtigen.

Sie helfen, Probleme zu identifizieren, die in isolierten Tests übersehen wurden, wie etwa Fehler in der Interaktion mit externen Systemen oder Inkonsistenzen im Ablauf der Applikation. Ein Beispiel wäre das Versagen einer App beim Abrufen von Daten von einem externen Server.

Die Durchführung von E2E-Tests ist aufwendig und erfordert eine sorgfältige Planung. Ein Nachteil von E2E-Tests ist die Schwierigkeit der Fehlerlokalisierung aufgrund ihres umfassenden Charakters und häufig manueller Ausführung.

### 3.2.2 Testfall

Während die vorangegangenen Abschnitte die verschiedenen Arten und Klassifikationen von Softwaretests beleuchtet haben, fokussiert sich dieser Abschnitt auf den Testfall als Kernkonzept und grundlegendstes Element des Testprozesses (ISO/IEC/IEEE 24765:2017).

Ein Testfall definiert, was genau getestet wird, indem er das Testobjekt (die zu testende Komponente oder das System), die Eingaben, erwarteten Ausgaben, Ausführungsbedingungen und gegebenenfalls weitere relevante Parameter detailliert festlegt. Das Testobjekt, wie bereits in Abschnitt 3.2.1.2 erläutert, ist ein zentraler Bestandteil des Testfalls, da es den konkreten Fokus des Tests bestimmt.

Er ist ein übergeordnetes Konzept, das auf verschiedenen Testebenen anwendbar ist (siehe Abschnitt 3.2.1.4). Die korrekte Konzeption und Durchführung von Testfällen ist entscheidend für die Effektivität des gesamten Testprozesses.

Die ISO, IEC und IEEE (2017, S. 464) bietet mehrere komplementäre Definitionen des Begriffs *Testfall*, die verschiedene Aspekte beleuchten:

1. „*set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement [IEEE 1012-2012 IEEE Standard for System and Software Verification and Validation, 3.3.31][Anmerkung des Autors: Die IEEE1012-2012 wurde inzwischen von der IEEE 1012:2016 abgelöst]*

2. *documentation specifying inputs, predicted results, and a set of execution conditions for a test item [IEEE 1012-2012 IEEE Standard for System and Software Verification and Validation, 3.1.31]*

3. *set of test case preconditions, inputs (including actions, where applicable), and expected results, developed to drive the execution of a test item to meet test objectives, including correct implementation, error identification, checking quality, and other valued information [ISO/IEC/IEEE 29119-1:2013 Software and systems engineering — Software testing — Part 1: Concepts and definitions, 4.48][Anmerkung des Autors: Die ISO/IEC/IEEE 29119-1:2013 wurde inzwischen von der ISO/IEC/IEEE 29119-1:2022 abgelöst bleibt inhaltlich aber kongruent]* “

Diese Definitionen ergänzen sich gegenseitig und bieten verschiedene Perspektiven auf das Konzept eines Testfalls. Die erste Definition betont die Zielsetzung, die zweite fokussiert die Dokumentation, und die dritte gibt einen umfassenderen Überblick über die Komponenten und Ziele eines Testfalls.

Basierend auf diesen Definitionen lässt sich ein Testfall als eine strukturierte Zusammenstellung von Elementen beschreiben, die darauf abzielen, bestimmte Aspekte einer Software unter kontrollierten Bedingungen zu evaluieren.

Die Komponenten eines Testfalls lassen sich wie folgt detaillieren (ISO/IEC/IEEE 29119-1:2022):

- **Testobjekt:** Die zu testende Komponente oder das System, wie in Abschnitt 3.2.1.2 erläutert. Bei White-Box-Tests ist dies der Quellcode, bei Black-Box-Tests das DUT.
- **Testumgebung:** Umfasst alle für die Testdurchführung erforderlichen Elemente, einschließlich Hardware (z.B. physisches Gerät oder Emulator), Software und Netzwerkeinstellungen. Sie kann mehrere Umgebungen integrieren um unterschiedliche Testobjekte und Prüfebenen abdecken zu können (ISO, IEC und IEEE 2021a).
- **Vorbedingungen:** Beschreiben den Ausgangszustand des Systems oder der Komponente vor der Testdurchführung.
- **Eingaben und Aktionen:** Spezifische Werte oder Bedingungen, die als Eingaben verwendet werden, einschließlich Benutzereingaben, Gesten oder Sensordaten.
- **Erwartete Ergebnisse:** Definieren die antizipierten unmittelbaren Ausgaben oder das Verhalten der Applikation während und direkt nach der Testausführung. Dies kann definierte Rückgabewerte, Bildschirmausgaben oder andere direkt beobachtbare Reaktionen umfassen.

- **Nachbedingungen:** Beschreiben den erwarteten Gesamtzustand des Systems nach Abschluss der Testdurchführung. Dies kann langfristigere Auswirkungen, Änderungen in der Datenbank, den Zustand von Systemressourcen oder andere indirekte Effekte umfassen, die über die unmittelbaren Ausgaben hinausgehen.
- **Testreferenz (engl. Test Oracle):** Ein Mechanismus zur Bestimmung, ob ein Test bestanden oder fehlgeschlagen ist. Es vergleicht die tatsächlichen Ergebnisse mit den erwarteten Ergebnissen und gehört damit zu den erwarteten Ergebnissen.

Ein Testfall stellt also eine strukturierte Zusammenstellung der genannten Komponenten in Form einer eindeutigen Dokumentation dar, um festgelegte Aspekte einer Software unter kontrollierten Bedingungen zu evaluieren. Diese Zusammenstellung wird für ein bestimmtes Ziel entwickelt, beispielsweise um einen bestimmten Programmpfad zu durchlaufen oder die Einhaltung einer Anforderung zu überprüfen.

In Bezug auf die in Abschnitt 3.2.1 diskutierten Klassifikationen lassen sich Testfälle für mobile Applikationen aufgrund ihrer Herausforderungen wie folgt einordnen:

- Sie sind Teil der dynamischen Prüftechniken (siehe Abschnitt 3.2.1.1).
- Je nach Fokus können sie White-Box- oder Black-Box-Elemente enthalten (siehe Abschnitte 3.2.1.2 und 3.2.1.2).
- Sie folgen oft einer strukturorientierten Testtechnik, insbesondere wenn Testüberdeckungs-Metriken zur Bewertung herangezogen werden (siehe Abschnitte 3.2.1.3 und 3.2.3).

Ein wichtiger Aspekt bei der Erstellung von Testfällen ist der Zeitpunkt ihrer Entwicklung im Softwareentwicklungsprozess:

- **A priori Testgenerierung (Test Driven Development (TDD)) (S. Fraser u. a. 2003):** Testfälle werden vor der Implementierung basierend auf Anforderungen und Spezifikationen erstellt. TDD folgt dem Rot-Grün-Refaktor-Zyklus:

  1. **Rot:** Zunächst wird ein Test geschrieben, der die gewünschte Funktionalität überprüft. Da der zu testende Code noch nicht existiert, schlägt dieser Test erwartungsgemäß fehl („rot“), das heißt der Test zeigt die Nichterfüllung der erwarteten Ergebnisse an.

  2. **Grün:** Der minimal notwendige Code wird implementiert, um den Test zu bestehen („grün“).

  3. **Refaktor:** Der Code wird verbessert, ohne die Funktionalität zu verändern. Die Tests dienen als Überprüfen, um sicherzustellen, dass die Anpassungen keine Fehler einführen.

- **Nachträgliche Testerstellung:** Testfälle werden nach der Implementierung entwickelt, oft basierend auf dem tatsächlichen Code.

Beide Ansätze haben ihre Vor- und Nachteile. TDD kann zu einer robusteren und besser testbaren Codebase führen, während die nachträgliche Testerstellung möglicherweise die tatsächliche Implementierung genauer abdeckt. In der Praxis wird oft eine Kombination beider Ansätze verwendet. Im Kontext von KI-basierten Testmethoden ergeben sich neue Möglichkeiten:

- Bei der Verwendung von LLMs für die Testfallerstellung kann ein White-Box-Tests -Ansatz verfolgt werden, bei dem der Quellcode analysiert wird, um relevante Testfälle zu generieren. Im Rahmen dieser Arbeit geschieht dies nachträglich auf Basis des bestehenden Codes, könnte aber auch zur Erstellung von TDD-Tests basierend auf den Anforderungen erfolgen.

- Bei der Verwendung von Reinforcement Learning (RL) kann ein Black-Box-Tests -Ansatz verfolgt werden, bei dem die fertig erstellte App auf unterschiedlichen Geräten ausgeführt wird. Dies geschieht immer nachträglich.

Testfälle sollten sowohl typische Fälle als auch Grenzfälle abdecken, um eine möglichst umfassende Qualitätssicherung zu gewährleisten.

#### 3.2.2.1 Kriterien eines Testfalls

Bei der Erstellung und Bewertung von Testfällen, insbesondere im Kontext von durch LLMs generierten Tests, sind mehrere Kriterien zu berücksichtigen. Diese Kriterien stellen sicher, dass die generierten Tests nicht nur technisch korrekt, sondern auch effektiv in der Qualitätssicherung von Software sind.

Die **syntaktische Korrektheit** der Testfälle bildet das Fundament für alle weiteren Aspekte. Sie umfasst die korrekte Verwendung der Programmiersprache sowie die Einhaltung etablierter Coding-Konventionen und Stilrichtlinien. Ein syntaktisch korrekter Test verwendet beispielsweise die richtigen Schlüsselwörter, Klammern und Semikolons an den richtigen Stellen und folgt den Namenskonventionen der jeweiligen Programmiersprache. Dies ist entscheidend für die Funktionalität und Wirksamkeit der Tests im Entwicklungsprozess, da syntaktische Fehler die Ausführung des Tests von vornherein verhindern würden (Ouédraogo u. a. 2024).

Die **Kompilierbarkeit** der Tests ist das nächste wichtige Kriterium. Ein kompilierbarer Test kann erfolgreich in ausführbaren Maschinencode übersetzt werden. Dies demonstriert die grundlegende Kompatibilität des Testcodes mit dem zu testenden Code und der Entwicklungsumgebung. Kompilierbarkeit setzt syntaktische Korrektheit voraus, geht aber darüber hinaus, indem sie auch die korrekte Verwendung von Typen, die Verfügbarkeit aller referenzierten Klassen und Methoden, und die Einhaltung der Sprachspezifikation sicherstellt (Ouédraogo u. a. 2024).

Darüber hinaus ist die **Ausführbarkeit** ein kritisches Kriterium, das über die bloße Kompilierbarkeit hinausgeht. Ein ausführbarer Test kann nicht nur kompiliert werden, sondern auch tatsächlich ohne Laufzeitfehler ausgeführt werden. Dies umfasst die korrekte Einbindung aller externen Abhängigkeiten, die Vermeidung von Nullpointer-Ausnahmen, die korrekte Initialisierung von Objekten und die Berücksichtigung möglicher Ausnahmesituationen. Ein ausführbarer Test sollte in der Lage sein, das zu testende System in einen definierten Zustand zu versetzen, die zu testenden Operationen durchzuführen und die Ergebnisse zu überprüfen, ohne selbst Fehler zu verursachen (Ouédraogo u. a. 2024).

Diese Kriterien bilden eine Hierarchie von Anforderungen an Testfälle, wobei jede Stufe auf der vorherigen aufbaut. Ein Test muss zunächst syntaktisch korrekt sein, um kompilierbar zu sein, und er muss kompilierbar sein, um ausführbar zu sein. Nur wenn alle diese Kriterien erfüllt sind, kann ein Test zur Qualitätssicherung der Software beitragen.

#### 3.2.2.2 Komponenten eines Testfalls

Ein typischer Unit-Test für mobile Applikationen besteht aus folgenden Kernkomponenten:

**Import-Anweisungen:** Zu Beginn des Testcodes stehen die notwendigen Import-Statements für Testbibliotheken und zu testende Klassen. Quellcode 3.1 zeigt ein Beispiel für solche Import-Anweisungen. Wie zu sehen ist, werden hier die JUnit-Testbibliothek und statische Assertion-Methoden importiert, die für die Durchführung von Tests unerlässlich sind.

```
1 import org.junit.Test;
2 import static org.junit.Assert.*;
```

**Code 3.1:** Import-Anweisungen

**Testklasse:** Eine Klasse, die alle Testmethoden für eine bestimmte Funktionalität enthält. In Quellcode 3.2 ist ein Beispiel für eine solche Testklasse dargestellt. Die Klasse ExampleUnitTest dient als Container für alle zugehörigen Testmethoden und ermöglicht eine strukturierte Organisation der Tests.

```
1 public class ExampleUnitTest {
2     // Testmethoden folgen hier
3 }
```

**Code 3.2:** Testklasse

**Testmethoden:** Einzelne Methoden, die jeweils einen konkreten Testfall repräsentieren. Quellcode 3.3 demonstriert eine typische Testmethode. Die Methode testLoginFunction() ist mit der @Test-Annotation versehen, was JUnit signalisiert, dass es sich um eine auszuführende Testmethode handelt.

```
1 @Test
2 public void testLoginFunction() {
3     // Testlogik hier
4 }
```

**Code 3.3:** Testmethode

**Methodenkörper:** Der Methodenkörper eines Unit-Tests folgt üblicherweise dem folgenden Schema:

- **Arrange**: Vorbereitung der Testdaten und Objekte.
- **Act**: Ausführung der zu testenden Methode oder Funktion.
- **Assert**: Überprüfung der erwarteten Ergebnisse durch Assertions.

**Assertions:** Assertions sind entscheidend für die Überprüfung, ob das Verhalten der zu testenden Methode den Erwartungen entspricht.

Sie vergleichen das tatsächliche Ergebnis mit dem erwarteten Ergebnis und erzeugen einen Fehler, wenn die Überprüfung fehlschlägt.

Quellcode 3.4 zeigt Beispiele für häufig verwendete Assertions. Es zeigt, wie assertEquals() verwendet wird, um zwei Werte auf Gleichheit zu prüfen, und wie assertTrue() eine Bedingung auf Wahrheit testet.

```
// Prüft, ob expected und actual gleich sind
assertEquals(expected, actual);
// Prüft, ob die Bedingung wahr ist
assertTrue(condition);
```

**Code 3.4:** Beispiele für Assertions

Diese Komponenten bilden das Grundgerüst eines Unit-Tests und ermöglichen die isolierte Überprüfung bestimmter Funktionalitäten in mobilen Applikationen. Durch die Verwendung dieser Struktur, wie in den Quellcodes 3.1 bis 3.4 gezeigt, können Entwickler robuste und aussagekräftige Tests erstellen, die die Qualität und Zuverlässigkeit ihrer mobilen Anwendungen sicherstellen.

### 3.2.3 Testüberdeckung und Überdeckungsmetriken

Im Softwaretestprozess dienen Metriken dazu, verschiedene Aspekte des Testens quantitativ zu erfassen und zu bewerten. Eine zentrale Metrik ist die Testüberdeckung. Sie quantifiziert, wie umfassend die durchgeführten Tests die zu testende Software abdecken.

Die Testüberdeckung hängt insbesondere mit der Klassifikation von Softwaretests nach Testreferenz zusammen (siehe Abschnitt 3.2.1.3). Sie ist relevant für Testtechniken, bei denen die Vollständigkeit der Tests anhand einer definierten Referenz, wie dem Quellcode oder den Anforderungen, gemessen werden kann.

Häufig wird der Begriff Codeüberdeckung (Code Coverage) synonym verwendet. Dieser bezieht sich jedoch ausschließlich auf die Überdeckung des Quellcodes durch Tests und berücksichtigt keine qualitativen Aspekte.

Die ISO, IEC und IEEE (2022a, S. 463) bietet zwei komplementäre Definitionen für *Testüberdeckung*, die verschiedene Aspekte des Konzepts beleuchten:

1. „degree, expressed as a percentage, to which specified test coverage items have been exercised by a test case or test cases [ISO/IEC/IEEE 29119-1:2013 Software and systems engineering — Software testing — Part 1: Concepts and definitions, 4.53][Anmerkung des Autors: Die 29119-1:2013 wurde inzwischen von der ISO/IEC/IEEE 29119-1:2022 abgelöst und ist kongruent zu dieser]“
2. „extent to which the test cases test the requirements for the system or software product [ISO/IEC 12207:2008 Systems and software engineering — Software life cycle processes, 4.51]“

Diese Definitionen umfassen sowohl die Ausführung von Codebestandteilen als auch die Überdeckung von Anforderungen, was direkt mit den White-Box- und Black-Box-Prüftechniken korrespondiert (siehe Abschnitt 3.2.1.2).

Die ISO/IEC/IEEE 29119-4:2021 bietet einen umfassenden Rahmen für die Berechnung und Anwendung von Überdeckungsmetriken. Gemäß diesem Standard wird die Überdeckung wie folgt berechnet:

$$C_t = (\frac{N}{T} \times 100)\% \tag{3.1}$$

Dabei ist:

- $C_t$ die erreichte Überdeckung für die jeweilige Testtechnik $t$,
- N die Anzahl der durch ausgeführte Testfälle abgedeckten Überdeckungselemente,

- T die Gesamtzahl der durch die Testtechnik identifizierten Überdeckungselemente.

Der Überdeckungsgrad kann dabei zwischen 0% und 100% liegen. Nicht alle Überdeckungselemente müssen zwangsläufig durchführbar sein. Ein Überdeckungselement gilt als nicht durchführbar, wenn es entweder technisch nicht ausführbar ist (z.B. aufgrund von Compiler-Optimierungen) oder wenn es logisch nicht durch einen Testfall abgedeckt werden kann (z.B. bei unerreichbarem Code).

Die Norm ISO/IEC/IEEE 29119-4:2021 schreibt nicht vor, ob nicht durchführbare Elemente bei der Überdeckungsberechnung mit einbezogen werden sollen oder nicht. Es muss jedoch festgelegt und dokumentiert werden, welche Vorgehensweise gewählt wurde.

Im Kontext der strukturorientierten Testtechniken (siehe Abschnitt 3.2.1.3) sind folgende Überdeckungsmetriken besonders relevant (Liggesmeyer 2009; Myers, Badgett und C. Sandler 2012; ISO, IEC und IEEE 2021b):

- **Anweisungsüberdeckung (Statement Coverage):** Diese Metrik misst den Anteil der ausgeführten Anweisungen im Quellcode. Sie ist relativ einfach zu ermitteln und bietet einen grundlegenden Einblick in die Testüberdeckung. Allerdings kann sie nicht garantieren, dass alle logischen Pfade (d.h. alle möglichen Ausführungsreihenfolgen durch den Code) durchlaufen werden. Gemäß ISO/IEC/IEEE 29119-4:2021 wird sie folgendermaßen definiert:

$$C_{\text{statement}} = \frac{\text{Anzahl der ausgeführten Anweisungen}}{\text{Gesamtzahl der ausführbaren Anweisungen}} \times 100\% \quad (3.2)$$

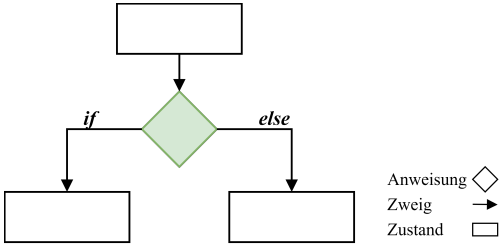



**Abbildung 3.5:** Anweisungsüberdeckung

- **Zeilenüberdeckung (Line Coverage):** Diese Metrik misst den Anteil der ausgeführten Codezeilen im Quellcode. Sie ähnelt der Anweisungsüberdeckung, berücksichtigt jedoch zusätzlich Zeilen wie Kommentare und Leerzeilen, die keine ausführbaren Anweisungen enthalten. Die Zeilenüberdeckung bietet einen leicht verständlichen und schnell zu erfassenden Überblick über die Testüberdeckung, da sie direkt mit den physischen Codezeilen korrespondiert. Sie wird folgendermaßen definiert:

$$C_{\text{line}} = \frac{\text{Anzahl der ausgeführten Codezeilen}}{\text{Gesamtzahl der Codezeilen}} \times 100\% \tag{3.3}$$

- **Zweigüberdeckung (Branch Coverage):** Diese Metrik erfasst, ob jede Verzweigung im Code (wie if-else-Statements) in beide Richtungen durchlaufen wurde. Sie wird teilweise auch als Entscheidungsüberdeckung (Decision Coverage) bezeichnet. Sie bietet eine tiefere Prüfung als die Anweisungsüberdeckung und hat sich in der Praxis als Minimalkriterium etabliert (Liggesmeyer 2009). Gemäß ISO/IEC/IEEE 29119-4:2021 wird sie folgendermaßen definiert:

$$C_{\text{branch}} = \frac{\text{Anzahl der durchlaufenen Zweige}}{\text{Gesamtzahl der Zweige}} \times 100\% \tag{3.4}$$

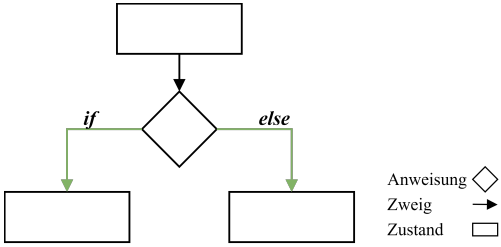



**Abbildung 3.6:** Zweigüberdeckung

- **Pfadüberdeckung (Path Coverage):** Diese Metrik zielt darauf ab, alle möglichen Ausführungspfade durch den Code zu testen. Sie bietet die umfassendste Prüfung der Logik eines Programms. In der Praxis ist eine vollständige Pfadüberdeckung oft nicht realisierbar, insbesondere bei Programmen mit Schleifen, da die Anzahl der möglichen Pfade exponentiell wachsen und sogar unendlich sein kann. Gemäß ISO/IEC/IEEE 29119-4:2021 wird sie folgendermaßen definiert:

$$C_{\text{path}} = \frac{\text{Anzahl der durchlaufenen Pfade}}{\text{Gesamtzahl der möglichen Pfade}} \times 100\% \qquad (3.5)$$

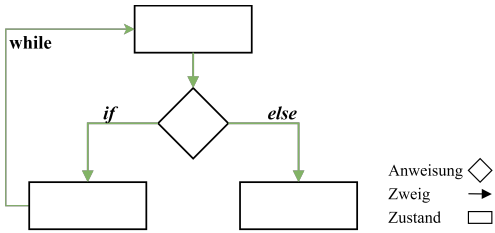



**Abbildung 3.7:** Pfadüberdeckung

Die Wahl der geeigneten Überdeckungsmetrik hängt von verschiedenen Faktoren ab, einschließlich der Art der Software, der verfügbaren Ressourcen und der Kritikalität des Systems (H. Zhu, Hall und May 1997).
In der Praxis werden zunehmend fortschrittliche Methoden wie *Productive Coverage* eingesetzt (Ivanković u. a. 2024). Dieser Ansatz erweitert traditionelle Codeüberdeckungsmetriken, indem er nicht abgedeckten Code basierend auf dessen Ähnlichkeit zu gut getestetem oder häufig ausgeführtem Produktionscode priorisiert. Die *Actionability* von Codeüberdeckungsinformationen beschreibt in diesem Kontext, wie gut diese Informationen Entwickler dabei unterstützen, konkrete Maßnahmen zur Verbesserung des Codes und der Tests abzuleiten. Durch die Priorisierung relevanter Codebereiche hilft Productive Coverage Entwicklern, sich auf die wichtigen, ungetesteten Teile des Codes zu konzentrieren.

Eine hohe Codeüberdeckung allein ist keine Garantie für die Abwesenheit von Fehlern. Sie dient vielmehr als Indikator für die Gründlichkeit der Tests und kann helfen, ungetestete Codebereiche zu identifizieren. Je mehr Überdeckung erreicht wird, desto wahrscheinlicher ist es, dass Fehler gefunden und behoben werden und gerade die letzten 20% Überdeckung können oft die größten Probleme aufdecken (Mockus, Nagappan und Dinh-Trong 2009). Dennoch ist 80% ein häufiges Ziel für die Codeüberdeckung, da es in der Praxis oft ausreichend ist, um die meisten Fehler zu finden.

Die Bewertung der Testvollständigkeit mittels Überdeckungsmetriken steht in engem Zusammenhang mit der Evaluation von Testmethoden, wie sie im SQuaRE-Rahmenwerk ISO/IEC 25010:2023 vorgeschlagen wird (siehe Abschnitt 3.1.1). Dabei können Überdeckungsmetriken als Teil der Qualitätskriterien für den Testprozess selbst betrachtet werden.

### 3.2.4 Sonderfälle

Bei der Entwicklung und dem Testen von Software treten häufig Szenarien auf, die besondere Aufmerksamkeit erfordern und nicht durch konventionelle Testmethoden ausreichend abgedeckt werden. Diese Sonderfälle stellen Entwickler und Tester vor Herausforderungen und erfordern konkrete Ansätze, um die Qualität und Sicherheit der Applikationen zu gewährleisten. In diesem Abschnitt werden zwei solcher Sonderfälle näher betrachtet: *Flaky-Tests* und *Fuzzing*.

#### 3.2.4.1 Flaky-Tests

Flaky-Tests stellen einen Sonderfall im Bereich des Softwaretestens dar, insbesondere bei mobilen Applikationen. Diese Tests zeichnen sich durch ein nicht-deterministisches Verhalten aus: Bei gleicher Codeversion und scheinbar identischen Testbedingungen können sie vermeintlich inkonsistente Ergebnisse liefern (Alshammari u. a. 2024).

Die inhärente Variabilität mobiler Umgebungen erhöht die Wahrscheinlichkeit von nicht-deterministischem Testverhalten erheblich. Faktoren wie wechselnde Netzwerkbedingungen, unterschiedliche Geräteleistungen, vielfältige Benutzerinteraktionen und häufige asynchrone Anfragen tragen zu dieser Problematik bei. Dieses unberechenbare Verhalten erschwert den Testprozess und beeinträchtigt die Effizienz der Softwareentwicklung signifikant (Parry u. a. 2022).

Es gibt unterschiedliche Ursachen für dieses unberechenbare Verhalten, wie Race Conditions in nebenläufigen Systemen über Asynchronität und Testabhängigkeiten bis hin zu Umgebungsfaktoren wie Netzwerklatenz oder Geräteleistung. Besonders in mobilen Applikationen, die häufig asynchrone Operationen verwenden und in variablen Umgebungen laufen, treten diese Probleme verstärkt auf (Luo u. a. 2014; Eck u. a. 2019; Sousa, Bezerra und Machado 2023).

Sie können vermeintliche Fehler aufdecken, die Entwicklerressourcen verschwenden, oder maskieren echte Fehler, die übersehen werden können. Dies kann zu einer verminderten Testüberdeckung führen, wenn Entwickler dazu übergehen, unzuverlässige Tests zu ignorieren. Zur Adressierung von Flaky-Tests werden verschiedene Strategien angewandt. Dazu gehören die wiederholte Ausführung von Tests, die Isolierung von Testumgebungen, deterministisches Testen durch Techniken wie Time Travel Debugging (Eine Art Rückwärtssuche im Code), die Verbesserung der Testlogik zur besseren Handhabung von Asynchronität und Timing-Problemen sowie der Einsatz von Monitoring- und Analysetools zur Identifikation und Klassifizierung von Flaky-Tests (Luo u. a. 2014; Bell u. a. 2018; Eck u. a. 2019; Sousa, Bezerra und Machado 2023; Alshammari u. a. 2024).

#### 3.2.4.2 Fuzzing

Fuzzing ist eine automatisierte Testtechnik, die zufällig generierte oder mutierte Eingabedaten verwendet, um unerwartetes Verhalten oder Sicherheitslücken in Software aufzudecken (Miller, M. Zhang und Heymann 2022). Im Bereich mobiler Applikationen gewinnt Fuzzing zunehmend an Bedeutung, da es Fehler identifizieren kann, die durch herkömmliche Testmethoden möglicherweise übersehen werden (Liang u. a. 2014). Bei mobilen Apps konzentriert sich Fuzzing oft auf die Eingabevalidierung, API-Aufrufe und die Verarbeitung von Daten aus unsicheren Quellen. Spezielle Fuzzing-Tools für mobile Plattformen, wie Android-Fuzzer oder FANS (H. Ye u. a. 2013; B. Liu u. a. 2020), generieren automatisch eine Vielzahl von Testfällen, die die App mit unerwarteten oder fehlerhaften Eingaben konfrontieren. Dies kann Ausfällen, Speicherlecks oder andere sicherheitsrelevante Probleme aufdecken.

Ein besonderer Vorteil des Fuzzings liegt in seiner Fähigkeit, Grenzfälle und unvorhergesehene Szenarien zu testen, die manuell schwer zu identifizieren wären. Allerdings erfordert Fuzzing oft eine sorgfältige Konfiguration und Anpassung an die jeweilige Applikation, um relevante und aussagekräftige Testergebnisse zu erzielen. In Kombination mit anderen Testmethoden wie statischer Analyse und manuellen Sicherheitstests bildet Fuzzing einen integralen Bestandteil eines umfassenden Sicherheitstestansatzes für mobile Applikationen.

## 3.3 Testen von mobilen Applikationen

Das Testen mobiler Applikationen erfordert eine Anpassung und Erweiterung der in Kapitel 3.2 diskutierten Grundlagen des Softwaretestens. Die in Kapitel 2 erörterten Eigenschaften und Herausforderungen mobiler Applikationen wirken sich direkt auf den Testprozess aus und erfordern spezielle Strategien und Methoden.

Die Heterogenität der Geräte und Betriebssystemversionen, wie in Abschnitt 2.1 beschrieben, stellt Tester vor die Aufgabe, eine breite Palette von Konfigurationen abzudecken. Dies erhöht die Komplexität des Testprozesses erheblich und erfordert effiziente Strategien zur Priorisierung und Automatisierung von Tests. Die Nutzung von Emulatoren in Kombination mit Tests auf realen Geräten ist dabei unerlässlich, um sowohl eine breite Überdeckung als auch realitätsnahe Testbedingungen zu gewährleisten.

Die in Abschnitt 2.2 diskutierten Entwicklungsansätze für mobile Apps beeinflussen direkt die Wahl der Testmethoden. Während native Apps oft plattformspezifische Testtools nutzen können, erfordern hybride und Web-Apps andere Testansätze.

Ein zentraler Aspekt beim Testen mobiler Applikationen ist die Benutzeroberfläche. Mobile Apps sind in der Regel stark auf intuitive und effiziente Benutzerinteraktionen ausgelegt, was die Bedeutung von GUI-Tests erhöht. Dies spiegelt sich in der in Abbildung 3.8 angepassten Testpyramide wider, die einen größeren Fokus auf GUI-Tests legt als die ursprüngliche Testpyramide von (Cohn 2010):

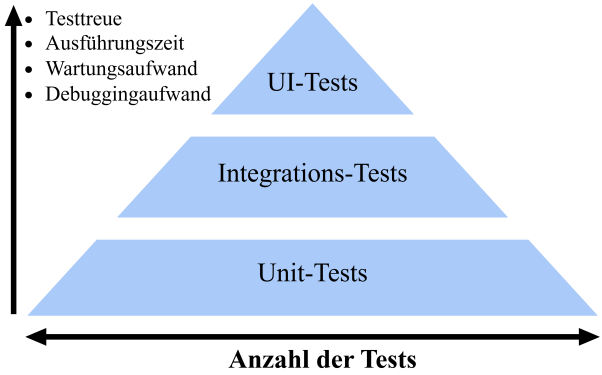



**Abbildung 3.8:** Testpyramide für mobile Geräte - Im Gegensatz zur traditionellen Testpyramide liegt der Schwerpunkt bei mobilen Apps auf GUI-Tests als Spitze der Pyramide.

Diese modifizierte Testpyramide veranschaulicht, dass GUI-Tests bei mobilen Apps einen höheren Stellenwert einnehmen als bei traditionellen Desktop-Anwendungen. Gleichzeitig bleiben Unit-Tests und Integrationstests weiterhin fundamental für die Sicherstellung der Code-Qualität und der korrekten Interaktion zwischen Komponenten.

Die Notwendigkeit, Tests unter verschiedenen Netzwerkbedingungen und mit unterschiedlichen Sensordaten durchzuführen, erfordert spezielle Testumgebungen und -werkzeuge. Diese müssen in der Lage sein, verschiedene Kontextszenarien zu simulieren, um die in Abschnitt 3.1.2.2 beschriebenen spezifischen Fehlerausprägungen bei mobilen Applikationen zu identifizieren.

Die hohe Updatefrequenz mobiler Apps und die Erwartung schneller Fehlerbehebungen erfordern zudem eine enge Integration des Testprozesses in CI/CD-Pipelines. Dies ermöglicht kontinuierliche Tests bei jeder Codeänderung und unterstützt schnelle Release-Zyklen, die für mobile Apps charakteristisch sind.

In den folgenden Abschnitten werden Testmethoden und -werkzeuge erläutert, die entwickelt wurden, um den besonderen Anforderungen beim Testen mobiler Applikationen gerecht zu werden. Dabei wird ein besonderer Fokus auf GUI-Tests gelegt, die eine Schlüsselrolle in der Qualitätssicherung mobiler Apps spielen.

Für automatisierte Tests bietet Android mehrere Möglichkeiten. Neben den bereits erwähnten Frameworks Espresso und JUnit können auch Werkzeuge wie *UIAutomator* für systemweite GUI-Tests und Robolectric für Unit-Tests ohne Emulator eingesetzt werden (Linares-Vásquez u. a. 2017).

### 3.3.1 GUI-Tests

GUI-Tests, auch als UI-Tests bezeichnet, verifizieren die korrekte Funktionalität der Benutzeroberfläche einer Applikation durch die Simulation vollständiger Interaktionsabläufe. Dafür benötigen sie eine kompilierte und ausführbare App. Sie nehmen eine zentrale Rolle im Testprozess mobiler Applikationen ein, da sie das primäre Interface zwischen Nutzer und Anwendung evaluieren.

Im Vergleich zu Unit-Tests erfordern GUI-Tests eine komplexere Teststruktur. Dies resultiert aus der Notwendigkeit, zusammenhängende Sequenzen von Interaktionen auszuführen und dabei verschiedene Zustände der Applikation zu berücksichtigen. Die Komplexität wird zusätzlich dadurch erhöht, dass bestimmte Funktionalitäten nur durch eindeutige Aktionssequenzen erreichbar sind, was die systematische Planung und Durchführung der Tests erschwert (Memon, Pollack und Soffa 1999).

Die Bedeutung von GUI-Tests für mobile Apps lässt sich aus den spezifischen Charakteristika dieser Anwendungen ableiten. Touch-basierte Interaktionen, die Vielfalt an Bildschirmgrößen und -auflösungen sowie die Notwendigkeit einer responsiven und intuitiven Benutzerführung stellen besondere Anforderungen an das GUI-Testen (Said u. a. 2021). Im Gegensatz zu Desktop-Anwendungen, die auch über die Tastatur, der Maus oder Hardwaretasten bedient werden können, erfolgt die Interaktion mit mobilen Apps nahezu ausschließlich über die grafische Benutzeroberfläche. Dies erhöht die Bedeutung der GUI-Tests, da die korrekte Funktion der Oberfläche für die gesamte Nutzbarkeit der Applikation essentiell ist.

Said u. a. (2021) identifizieren vier Hauptziele des GUI-Testens: *Funktionalität*, *Zuverlässigkeit*, *Leistung* und *Sicherheit*, die mit vier der acht in Abschnitt 3.1.1 vorgestellten Evaluationsanforderungen korrespondieren. In der Praxis des mobilen App-Testens konzentrieren sich diese Ziele auf bestimmte Aspekte der Benutzeroberfläche und Interaktion. Dazu gehören die Überprüfung der korrekten Darstellung von GUI-Elementen auf verschiedenen Geräten und Bildschirmgrößen sowie die Sicherstellung der korrekten Funktionalität von Touch-Gesten und Interaktionen. Ebenso wichtig sind das Testen der Navigation und des Workflows innerhalb der App, die Überprüfung der Reaktionsfähigkeit und Performance der Benutzeroberfläche sowie die Sicherstellung der Konsistenz des Designs und der Benutzerführung. Diese Aspekte spiegeln die unmittelbaren Bedürfnisse der Endnutzer wider, für die eine funktionale, zuverlässige und intuitive App von primärer Bedeutung ist. Dabei liegt der Fokus häufig auf der Funktionalität und Zuverlässigkeit, gefolgt von Leistungsaspekten, während Sicherheitstests oft eine nachgeordnete Rolle spielen, obwohl sie ebenfalls von Bedeutung sind (Said u. a. 2021).

UI-Tests für mobile Apps können sowohl manuell als auch automatisiert durchgeführt werden, wobei beide Ansätze ihre Stärken haben. Manuelle Tests sind besonders wertvoll für die Bewertung des Benutzererlebnisses und die Entdeckung von Usability-Problemen, da sie die intuitive Interaktion eines realen Benutzers nachahmen. Automatisierte GUI-Tests hingegen ermöglichen eine effiziente und wiederholbare Überprüfung der Funktionalität auf verschiedenen Geräten und unter verschiedenen Bedingungen. Sie eignen sich besonders für Regressionstests (siehe Abschnitt 3.2.1.3) und die Simulation komplexer Szenarien, die manuell schwer zu reproduzieren wären.

Für automatisierte GUI-Tests haben sich verschiedene Ansätze etabliert:

- **Modellbasiertes Testen:** Hierbei wird ein Modell der Applikation erstellt, aus dem Testfälle generiert werden.

- **Zufallsbasiertes Testen (siehe Abschnitt 3.3.4):** Auch als „Monkey Testing“ bekannt, generiert zufällige Eingabesequenzen.
- **Suchbasiertes Testen:** Verwendet Optimierungsalgorithmen, um Testsequenzen zu finden, die mit wenigen Tests eine hohe Wahrscheinlichkeit aufweisen, Fehler zu finden.
- **Symbolische Ausführung:** Analysiert den Quellcode, um Testfälle zu generieren, die alle möglichen Pfade abdecken.

Jeder dieser Ansätze hat Vor- und Nachteile und eignet sich für unterschiedliche Testszenarien (Said u. a. 2021). Modellbasiertes Testen beispielsweise ermöglicht eine systematische Überdeckung der Applikationslogik, während zufallsbasiertes Testen unerwartete Fehler aufdecken kann, die durch strukturierte Tests möglicherweise übersehen würden.

Eine häufig verwendete Methode zur Durchführung automatisierter GUI-Tests ist der Capture-and-Replay-Ansatz (Dolotta u. a. 1976), der im nächsten Abschnitt näher erläutert wird. Dieser Ansatz bietet eine Brücke zwischen manuellen und vollständig automatisierten Tests und ist besonders nützlich für die initiale Erstellung von Testfällen.

### 3.3.2 Capture-and-replay-Tests

Eine besonders relevante Methode für GUI-Tests im Bereich mobiler Applikationen sind Capture-and-Replay-Tests. Capture-and-Replay-Tests kommen zwar nicht nur bei mobilen Applikationen zur Anwendung, sind dort aber besonders weit verbreitet. Diese Methode basiert auf einem zweistufigen Prozess, der zunächst einen potenziellen Testablauf aufzeichnet *(Capture)* und diesen anschließend automatisiert wiederholt *(Replay)* (Dolotta u. a. 1976).

In der *Capture-Phase* werden Benutzerinteraktionen mit der Applikation systematisch aufgezeichnet (C. H. Liu u. a. 2014). Dies umfasst verschiedene Arten von Eingaben, von einfachen Touchgesten bis hin zu komplexeren Interaktionen wie Texteingaben, Navigationsaktionen und Systemereignisse. Gleichzeitig werden die resultierenden Änderungen in der Benutzeroberfläche dokumentiert, oft mittels Screenshots oder anderer Repräsentationen des GUI-Zustands zu definierten Zeitpunkten.

Während der *Replay-Phase* werden die aufgezeichneten Aktionen automatisch reproduziert. Das System führt dabei einen detaillierten Vergleich zwischen den aktuellen Ergebnissen und den während der Aufnahme gespeicherten Zuständen durch (siehe Abschnitt 3.2.2). Dieser Vergleichsprozess ermöglicht die präzise Identifikation von Abweichungen oder Fehlern durch die Gegenüberstellung der erwarteten Soll-Zustände mit den tatsächlichen Ist-Zuständen der Benutzeroberfläche.

Capture-and-Replay-Tests bieten den Vorteil, dass sie die Erstellung mehrschrittiger Testszenarien ohne umfangreiche Programmierkenntnisse ermöglichen. Zudem erlauben sie die präzise Replikation realer Benutzerinteraktionen, einschließlich vielschichtiger Touch-Gesten wie Wischen, Zoomen oder Mehrfinger-Bedienung. Allerdings weisen sie auch Limitationen auf, insbesondere eine hohe Sensitivität gegenüber GUI-Änderungen. Selbst geringfügige Modifikationen der Benutzeroberfläche können zu Fehlschlägen zuvor aufgezeichneter Tests führen, was falsche Fehlermeldungen generieren kann. Darüber hinaus zeigen sie oft eine geringere Flexibilität als programmierte Tests bei der Handhabung dynamischer oder zufälliger GUI-Elemente.

Ein exemplarisches Tool für den Capture-and-Replay-Ansatz bei Android-Applikationen ist der Android GUIRipper (Amalfitano u. a. 2012). Dieses Instrument automatisiert das Testen von Android-Benutzeroberflächen durch die Erstellung einer strukturierten Hierarchie der GUI-Elemente und die Identifikation potenzieller Interaktionen. Der GUIRipper analysiert die aktuelle GUI-Struktur, identifiziert interaktive Elemente, führt Aktionen auf diesen Elementen aus, zeichnet den resultierenden GUI-Zustand auf und wiederholt diesen Prozess für neue, unentdeckte Zustände. Dabei speichert er neu entdeckte interaktive Komponenten in einem „*Planner*“, der für die Auswahl künftiger Interaktionen zuständig ist. Dieser Prozess wird fortgesetzt, bis keine interaktiven Elemente mehr im Planner vorhanden sind, was auf einen vollständigen Test der App hindeutet.

Capture-and-Replay-Tests bilden oft die Grundlage für weiter automatisierte Prüftechniken, indem sie die durchgeführten Tests als Espresso-Tests (Negara, Esfahani und Buse 2019) speichern. Diese spezialisierten Tests für Android-Applikationen werden im folgenden Abschnitt näher erläutert und stellen eine wichtige Komponente im Instrumentarium für das Testen mobiler Apps dar.

### 3.3.3 Espresso-Tests

Espresso hat sich als spezialisiertes Testframework für Android etabliert, das speziell für GUI-Tests entwickelt wurde (Negara, Esfahani und Buse 2019; Google 2024a). Als integraler Bestandteil der *Android Testing Support Library*[25] bietet Espresso eine leistungsfähige API zur Simulation von Benutzerinteraktionen und zur Überprüfung von GUI-Elementen. Eine der Hauptstärken von Espresso liegt in seiner Fähigkeit, Testaktionen automatisch mit dem GUI-Thread der Applikation zu synchronisieren (Negara, Esfahani und Buse 2019).

[25] Android Testing Support Library: Framework zum Testen von Android Apps (`https://android.github.io/android-test/`).

Diese Eigenschaft erhöht nicht nur die Zuverlässigkeit der Tests, sondern trägt auch wesentlich zur Reduzierung von Flaky Tests (siehe Abschnitt 3.2.4.1) bei (Coppola, Morisio und Torchiano 2017). Dadurch adressiert Espresso effektiv einige der zentralen Herausforderungen beim Testen mobiler Applikationen, insbesondere im Umgang mit Netzwerkanfragen, Animationen und asynchronen Operationen.

Espresso-Tests lassen sich als Gray-Box-Tests charakterisieren (siehe Abschnitt 3.2.1.2). Sie kombinieren die Möglichkeit des Zugriffs auf den internen Applikationszustand mit der systematischen Überprüfung des Oberflächenverhaltens. Diese Kombination erlaubt eine detaillierte Prüfung der GUI-Funktionalität, ohne dass ein umfassendes Verständnis des gesamten Applikationscodes erforderlich ist (Linares-Vásquez u. a. 2017). Dadurch eignen sich Espresso-Tests besonders gut für Entwickler und Tester, die sich auf die Benutzererlebnis und die korrekte Funktionsweise der GUI konzentrieren möchten.

Die Architektur der Espresso API basiert auf drei Kernkomponenten: *ViewMatchers* zur Lokalisierung von View-Elementen, *ViewActions* zur Durchführung von Aktionen und *ViewAssertions* zur Verifikation des View-Zustands. Diese Struktur ermöglicht einen intuitiven und leicht verständlichen Aufbau von Tests, der eng an der tatsächlichen Benutzerinteraktion orientiert ist.

Der Espresso Test Recorder (ETR) in Android Studio ermöglicht die Aufzeichnung von GUI-Tests durch manuelle Bedienung der App und generiert automatisch Espresso-Testcode (Negara, Esfahani und Buse 2019; Google 2024d). Dies kombiniert den Capture-and-Replay-Ansatz mit der Präzision manuell erstellter Tests. Zur Veranschaulichung der Struktur und Funktionsweise eines Espresso-Tests soll folgendes Beispiel dienen:

```java
@Test
public void testLoginButton() {
    onView(withId(R.id.username))
        .perform(typeText("user123"));
    onView(withId(R.id.password))
        .perform(typeText("pass123"));
    onView(withId(R.id.loginButton))
        .perform(click());
    onView(withId(R.id.welcomeMessage))
        .check(matches(isDisplayed()));
}
```

**Code 3.5:** Espressotestbeispiel

Dieser Test demonstriert die typische Struktur eines Espresso-Tests: Zunächst werden GUI-Elemente lokalisiert, dann werden Aktionen auf diesen Elementen ausgeführt, und schließlich wird der resultierende Zustand verifiziert. In diesem konkreten Fall wird ein Login-Vorgang simuliert und anschließend überprüft, ob eine Willkommensnachricht angezeigt wird.

### 3.3.4 Zufälliges Testen

Zufälliges Testen (Random Testing) stellt einen alternativen Ansatz zum Testen mobiler Applikationen dar, der sich durch seine Einfachheit und Skalierbarkeit auszeichnet. Diese Methode ist dadurch charakterisiert, dass Testeingaben zufällig aus dem gesamten Eingabebereich einer Software generiert werden, ohne dabei die interne Struktur des Programms oder bestimmte Testkriterien zu berücksichtigen (Hamlet 1994).

Im Kontext mobiler Applikationen manifestiert sich zufälliges Testen in der Generierung zufälliger Sequenzen von GUI-Ereignissen, die Berührungen, Gesten oder Systemeingaben umfassen können. Das primäre Ziel dieses Ansatzes ist es, durch diese zufälligen Interaktionen möglichst viele Zustände und Pfade der Applikation zu erreichen und dabei potenzielle Fehler oder Abstürze aufzudecken. Diese Methode kann besonders effektiv sein, um unerwartete Fehler zu entdecken, die bei systematischen Testansätzen möglicherweise übersehen werden (Patel u. a. 2018).

Ein Beispiel für die Implementierung des zufälligen Testens im Android-Ökosystem ist das Monkey-Tool, das Teil des Android SDK ist (Patel u. a. 2018; Google 2023). Monkey generiert und injiziert pseudo-zufällige Benutzereingaben in eine Android-Applikation, basierend auf einer vordefinierten Wahrscheinlichkeitsverteilung für verschiedene Ereignistypen. Die Effektivität dieses scheinbar simplen Ansatzes wurde in einer vergleichenden Studie demonstriert, in der Monkey die höchste durchschnittliche Anweisungsüberdeckung unter den verglichenen Testtools erreichte (Choudhary, Gorla und Orso 2015).

Trotz seiner Stärken weist zufälliges Testen auch einige Einschränkungen auf. Eine wesentliche Herausforderung besteht darin App-Teile zu erreichen, die bestimmte Eingabesequenzen erfordern. Zudem bietet es keine Garantie für eine systematische Überdeckung aller möglichen App-Zustände.

Dennoch stellt es aufgrund seiner Einfachheit, Skalierbarkeit und der erzielten Codeüberdeckung eine relevante Methode für das Testen mobiler Applikationen dar.

### 3.3.5 Werkzeuge

Die Entwicklung und das Testen von Android-Applikationen werden durch eine Vielzahl spezialisierter Werkzeuge unterstützt. Diese Tools decken verschiedene Aspekte des Testprozesses ab, von der automatischen Testgenerierung über die Messung der Codeüberdeckung bis hin zur Bereitstellung konsistenter Testumgebungen.

#### 3.3.5.1 Evosuite

Evosuite implementiert einen suchbasierten Ansatz zur automatischen Testgenerierung und setzt die in Abschnitt 3.2.1 diskutierten Prinzipien des strukturorientierten Testens um (G. Fraser und Arcuri 2013). Es nutzt genetische Algorithmen, um Tests zu erzeugen, die eine hohe Codeüberdeckung erreichen (G. Fraser und Arcuri 2013).

Der Prozess umfasst folgende Schritte:

1. Analyse des Quellcodes
2. Generierung initialer Testfälle
3. Bewertung der Tests durch eine Fitnessfunktion
4. Iterative Verbesserung der Tests durch Selektion, Kreuzung und Mutation

Evosuite eignet sich besonders für die Erstellung von Unit-Tests (siehe Abschnitt 3.2.1.4) und kann die in Abschnitt 3.2.3 diskutierte Testüberdeckung signifikant verbessern (G. Fraser und Arcuri 2013). Ein wesentlicher Vorteil liegt in der Fähigkeit, Randfälle zu identifizieren, die bei manueller Testerstellung möglicherweise übersehen würden. Allerdings erfordert der hohe Rechenaufwand eine sorgfältige Abwägung zwischen Testqualität und verfügbaren Ressourcen, insbesondere bei umfangreichen mobilen Applikationen.

#### 3.3.5.2 JUnit

JUnit ist ein Framework für automatisierte Tests in Java[26]. Die Version 4 des Frameworks führte Annotationen ein, die die Testentwicklung stark vereinfachten und den modernen Testprozess für Java-basierte Anwendungen, einschließlich Android-Apps, maßgeblich geprägt haben (Tahchiev u. a. 2010).

Das Framework basiert auf dem xUnit-Architekturmuster und implementiert den in Abschnitt 3.2.2.2 beschriebenen strukturierten Aufbau von Testfällen. Kernelemente der JUnit 4 Architektur sind:

- **@Test**: Markiert Methoden als Testfälle.
- **@Before/@After**: Definiert Setup- und Cleanup-Code für jeden einzelnen Test.
- **@BeforeClass/@AfterClass**: Definiert einmalige Setup- und Cleanup-Operationen für die gesamte Testklasse.
- **Assert-Methoden**: Bieten verschiedene Möglichkeiten zur Überprüfung von Testergebnissen.

Im Kontext mobiler Applikationen wird JUnit häufig in Kombination mit Android-spezifischen Erweiterungen wie AndroidJUnitRunner[27] verwendet. Diese Integration ermöglicht die Ausführung von Tests sowohl auf der Java Virtual Machine (JVM) als auch auf Android-Geräten oder Emulatoren.

[26] JUnit ist ein Framework zur Erstellung von Java Tests (`https://junit.org/junit4/`)

[27] AndroidJUnitRunner ist eine Erweiterung um JUnit Tests in Android auszuführen (`https://developer.android.com/training/testing/instrumented-tests/androidx-test-libraries/runner`)

Ein besonderer Vorteil von JUnit liegt in seiner Integration mit Entwicklungsumgebungen und Build-Tools wie Android Studio und Maven, was eine nahtlose Einbindung in den Entwicklungsprozess ermöglicht. Die Kombination mit anderen Werkzeugen wie Mockito für die Simulation von Abhängigkeiten und Jacoco für die Messung der Testüberdeckung macht JUnit zu einem zentralen Baustein im Testprozess für mobile Applikationen.

#### 3.3.5.3 Jacoco

Jacoco, ein Akronym für *Java Code Coverage*, hat sich als Standard-Tool zur Messung der Codeüberdeckung für Java-Anwendungen etabliert (M. R. Hoffmann 2007; Lingampally, Gupta und Jalote 2007). Es analysiert den Java-Bytecode während der Testausführung und erstellt detaillierte Berichte über verschiedene Überdeckungsmetriken. Die Funktionsweise basiert auf der Instrumentierung des Bytecodes zur Erkennung der verwendeten Codestellen, wodurch Informationen über die Ausführung jeder Anweisung, Verzweigung und Methode gesammelt werden.

Ein besonderer Vorteil von Jacoco liegt in der Visualisierung in HTML der Überdeckungsmetriken, die eine schnelle Identifizierung von Bereichen mit geringer Überdeckung ermöglicht.

#### 3.3.5.4 Containerisierung

Containerisierungstechnologien wie Docker[28] bieten Möglichkeiten zur Standardisierung von Entwicklungs- und Testumgebungen.

Im Kontext mobiler Applikationen adressiert dies die Herausforderungen der Geräte- und Betriebssystemfragmentierung durch die Bereitstellung isolierter, reproduzierbarer Umgebungen.

[28] Docker: Open-Source-Software zur Automatisierung der Bereitstellung von containerisierten Anwendungen (`https://www.docker.com/`).

Potenzielle Vorteile der Containerisierung für den Testprozess mobiler Applikationen umfassen:

- Reproduzierbarkeit von Testumgebungen.
- Vereinfachung des Installations-Prozesses.
- Möglichkeit zur parallelen Ausführung multipler Testinstanzen mit unterschiedlichen Parametern.

Sie erlaubt zudem die Nutzung einer einheitlichen, leicht einzurichtenden Testumgebung, die auch komplexere, KI-basierte, Testmethoden unterstützt.

#### 3.3.5.5 Defects4J

Defects4J ist eine Datenbank und ein Framework für Softwarefehler, die speziell für die reproduzierbare Forschung im Bereich des Softwaretestens entwickelt wurde (Just, Jalali und Ernst 2014). Die in Defects4J vorhandenen Fehler sind in Open-Source-Programmen im Produktiveinsatz aufgetreten und wurden manuell isoliert, um unabhängig von anderen Änderungen zu sein.
Das Framework stellt 854 (Stand Mai 2025) reale Fehler aus fünf Open-Source-Java-Programmen bereit, die jeweils durch mindestens einen Testfall nachgewiesen werden können.

Die zentrale Bedeutung von Defects4J liegt in drei Aspekten:

- Isolierte Fehler: Jeder Fehler wurde manuell von nicht verwandten Änderungen wie Features oder Refactorings getrennt.
- Reproduzierbarkeit: Jeder Fehler wird durch mindestens einen Testfall nachgewiesen.

- Einheitliche Schnittstelle: Das Framework bietet eine standardisierte API für den Zugriff auf fehlerhafte und korrigierte Programmversionen.

Das Framework ermöglicht durch seine Testausführungskomponente die Integration verschiedener Testgenerierungs-, Testausführungs- und Codeüberdeckungswerkzeuge. Die containerisierte Ausführungsumgebung gewährleistet dabei die Reproduzierbarkeit der Tests über verschiedene Systeme hinweg.

Die durch Defects4J bereitgestellte Sammlung realer Fehler und die standardisierte Testinfrastruktur machen es zu einem wichtigen Werkzeug für die empirische Evaluation von Testmethoden, insbesondere im Kontext automatisierter Testgenerierung.

# 4 Grundlagen Maschinelle Lernverfahren

Künstliche Intelligenz (KI) bezeichnet ein Teilgebiet der Informatik, das sich mit der Entwicklung und Implementierung von computerbasierten Systemen befasst, die Aufgaben ausführen können, welche üblicherweise menschliche Intelligenz erfordern. Diese Systeme zeichnen sich durch ihre Fähigkeit aus, aus Daten zu lernen, Muster zu erkennen und adaptive Entscheidungen zu treffen.
Die ISO/IEC 24030:2024 unterscheidet zwischen KI als System und als Ingenieurdisziplin:

Als System beschreibt sie KI als die Fähigkeit, Wissen in Form von Modellen zu erwerben, zu verarbeiten, zu erstellen und anzuwenden, um definierte Aufgaben durchzuführen. Als Ingenieurdisziplin umfasst KI die Entwicklung und Erforschung dieser Systeme (ISO/IEC 24030:2024). Diese technische Definition wird durch die KI-Verordnung der Europäischen Union (European Union 2024) ergänzt, die KI-Systeme als maschinenbasierte Systeme charakterisiert, die mit unterschiedlichen Autonomiegraden Vorhersagen, Empfehlungen oder Entscheidungen generieren können, welche physische oder virtuelle Umgebungen beeinflussen. Zentrale Merkmale von KI-Systemen umfassen die Verarbeitung mehrdimensionaler Eingabedaten, die Fähigkeit zur Mustererkennung und Generalisierung, die Adaption des Verhaltens basierend auf Erfahrungen sowie die autonome Entscheidungsfindung innerhalb definierter Parameter.
Ein zentraler Teilbereich der KI ist Machine Learning (ML), das an der Schnittstelle von Statistik und Informatik angesiedelt ist.

Dieses Kapitel bietet einen Überblick über die Grundlagen des ML sowie dessen in der Abbildung 4.1 dargestellten Teilmengen. Besonderer Fokus liegt in diesem Kontext auf Large Language Models (Große Sprachmodelle, LLMs) – einer Untergruppe des Natural Language Processing (Natürliche Sprachverarbeitung, NLP) zur Erstellung von White-Box-Tests – und Reinforcement Learning (Bestärkendes Lernen, RL) für Black-Box-Tests .

## 4.1 Einführung in Maschinelles Lernen

ML konzentriert sich auf die Entwicklung von Algorithmen und Modellen, die aus Daten lernen, um Vorhersagen zu treffen oder Entscheidungen zu fällen (A. Zhang u. a. 2023; T. Mitchell 1997; Carbonell, Michalski und T. M. Mitchell 1983). Samuel (1959) definierte ML als Forschungsfeld, das Computern die Fähigkeit verleiht, Aufgaben auszuführen und sich zu verbessern, ohne dass die spezifischen Lösungsschritte explizit programmiert werden müssen. Stattdessen werden die Systeme durch die Analyse von Beispieldaten und die Erkennung von Mustern in die Lage versetzt, eigenständig Lösungsstrategien zu entwickeln.

ML-Techniken ermöglichen es Computern, Muster in Daten zu erkennen und daraus Erkenntnisse zu gewinnen, wodurch sie ihre Leistung durch Erfahrung verbessern können (Das u. a. 2015; LeCun, Bengio und G. Hinton 2015). Diese Fähigkeit zur Mustererkennung und daraus Anpassungen abzuleiten macht ML besonders wertvoll für die Verarbeitung und Analyse von großen Datenbeständen (Big Data) (Das u. a. 2015).

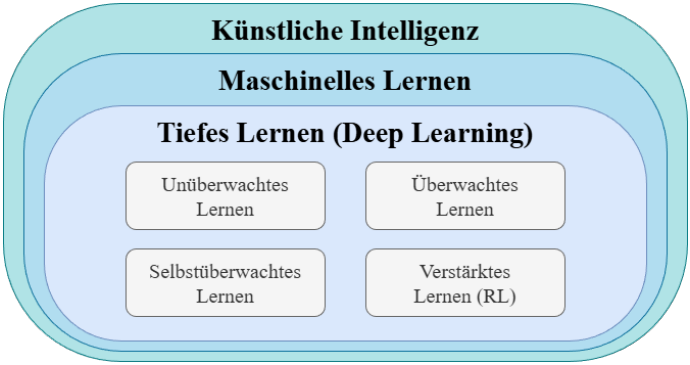



**Abbildung 4.1:** Übersicht über die Einteilung maschineller Lernverfahren

Wie in Abbildung 4.1 dargestellt, lassen sich ML-Methoden grob in folgende Kategorien einteilen:

- **Überwachtes Lernen (Supervised Learning)**: Training mit annotierten Datenpaaren, bei dem der Algorithmus lernt, eine Eingabe einer bekannten Ausgabe zuzuordnen.
- **Unüberwachtes Lernen (Unsupervised Learning)**: Lernen aus nicht annotierten Daten, wobei der Algorithmus selbstständig Muster und Strukturen in den Daten erkennt.
- **Semiüberwachtes Lernen (Semi-supervised Learning)**: Kombination aus annotierten und nicht annotierten Daten.
- **Selbstüberwachtes Lernen (Self-supervised Learning)**: Der Algorithmus extrahiert automatisch Lernziele und deren zugehörige Zielwerte aus den vorhandenen Daten selbst. Ein Beispiel hierfür ist das Vorhersagen des nächsten Wortes in einem Text basierend auf dem vorherigen Kontext, wobei der ursprüngliche Text sowohl die Eingabe als auch die korrekte Ausgabe liefert.

- **Reinforcement Learning (Bestärkendes Lernen, RL)**: Lernen durch Interaktion mit einer Umgebung, wobei der Agent durch Belohnungen und Bestrafungen sein Verhalten verbessert. Mit Agent ist hierbei das ML-Modell gemeint, das in einer Umgebung agiert und durch Interaktion mit dieser lernt und im Gegensatz selbstständig Entscheidungen trifft und daraufhin Aktionen ausführt.

Diese Kategorien unterscheiden sich in der Art der verwendeten Daten und den Lernzielen. Im Folgenden werden die wichtigsten Ansätze detailliert betrachtet.

### 4.1.1 Überwachtes Lernen (Supervised Learning)

Überwachtes Lernen ist eine zentrale Methode im maschinellen Lernen, bei der ein Modell mithilfe von annotierten Trainingsdaten (*„gelabelt“*) konditioniert wird (LeCun, Bengio und G. Hinton 2015; A. Zhang u. a. 2023). Diese Methode zielt darauf ab, die Beziehung zwischen Eingabevariablen (Features) und den entsprechenden Ausgabevariablen (*„Labels“*) abzubilden.

Im Kern des überwachten Lernens steht das Konzept der Schätzung der bedingten Wahrscheinlichkeitsverteilung $P(Y|X)$. Diese Verteilung beschreibt die Wahrscheinlichkeit für eine bestimmte Ausgabevariable $Y$ unter der Bedingung, dass die Eingabevariablen $X$ beobachtet wurden (Goodfellow, Bengio und Courville 2016).

Der Trainingsprozess umfasst die Anpassung der internen Modellparameter, die aus zwei Hauptkomponenten bestehen: Gewichte, die die Stärke der Verbindungen zwischen den Verarbeitungseinheiten des Modells bestimmen, und Bias-Terme, die als konstante Verschiebungswerte fungieren. Diese Parameter werden so angepasst, dass die Wahrscheinlichkeit der korrekten Vorhersagen für die Trainingsdaten maximiert wird. Dies entspricht der Minimierung der Diskrepanz zwischen den tatsächlich beobachteten Ausgabevariablen und den vom Modell vorhergesagten Ausgabevariablen.

Die annotierten Datenpunkte dienen als Referenz und Validierung für das Modell, um Muster und Beziehungen zwischen den Eingabevariablen und ihren entsprechenden Ausgabevariablen zu erkennen. Ein wesentlicher Prozess im überwachten Lernen ist das Training des Modells. Während des Trainings lernt das Modell, die Beziehung zwischen den Eingabevariablen und den Ausgabevariablen zu verstehen.

Ein grundlegendes Beispiel für überwachtes Lernen ist die lineare Regression, die den Zusammenhang zwischen Eingabe- und Ausgabevariablen durch eine lineare Funktion modelliert:

$$\hat{\mathbf{y}} = X\mathbf{w} + \mathbf{b}. \tag{4.1}$$

Hierbei ist $\hat{\mathbf{y}} \in \mathbb{R}^n$ der Vektor der vorhergesagten Ausgabevariablen, wobei $n$ die Anzahl der Datenpunkte bezeichnet. $\mathbf{w} \in \mathbb{R}^d$ ist der Gewichtsvektor, wobei $d$ die Anzahl der Merkmale bezeichnet.
$X \in \mathbb{R}^{n \times d}$ ist die Matrix der Eingabevariablen mit $n$ Datenpunkten in den Zeilen und $d$ Merkmalen in den Spalten. Der Bias-Term $\mathbf{b} \in \mathbb{R}^n$ ermöglicht es dem Modell, die Vorhersagen entlang der y-Achse zu verschieben und somit auch lineare Zusammenhänge abzubilden, deren Graph nicht durch den Koordinatenursprung verläuft (Goodfellow, Bengio und Courville 2016).

Die lineare Regression wird durch die Minimierung einer Verlustfunktion optimiert (A. Zhang u. a. 2023) und bildet die Grundlage des überwachten Lernens. Als Verlustfunktion wird üblicherweise der mittlere quadratische Fehler zwischen den vorhergesagten Werten $\hat{\mathbf{y}}$ und den tatsächlichen Ausgabevariablen $\mathbf{y}$ verwendet:

$$L(\mathbf{w}) = (\mathbf{y} - X\mathbf{w})^T(\mathbf{y} - X\mathbf{w}) \tag{4.2}$$

wobei $w^T$ die Transposition der Matrix $w$ bezeichnet, die für die Berechnung des quadratischen Fehlers erforderlich ist.

Diese Verlustfunktion hat mehrere vorteilhafte Eigenschaften (Su, Yan und Tsai 2012):

- Sie ist differenzierbar, was eine analytische Lösung ermöglicht.
- Durch die Quadrierung werden größere Abweichungen stärker bestraft.
- Positive und negative Abweichungen werden symmetrisch behandelt.

Die Minimierung dieser Verlustfunktion führt zur analytischen Lösung für den Gewichtsvektor $\mathbf{w}^*$, die als Normal-Gleichung bekannt ist:

$$\mathbf{w}^* = (X^T X)^{-1} X^T \mathbf{y} \tag{4.3}$$

Diese Lösung besitzt die BLUE-Eigenschaft (Best Linear Unbiased Estimator). Dies bedeutet, dass unter allen linearen und erwartungstreuen Schätzern dieser Schätzer die geringste Varianz aufweist (Su, Yan und Tsai 2012).
Ein Schätzer gilt als erwartungstreu, wenn sein Erwartungswert dem wahren Parameterwert entspricht. Für große Datensätze wird in der Praxis häufig statt dieser analytischen Lösung eine iterative Approximation mittels Stochastic Gradient Descent (SGD) verwendet (A. Zhang u. a. 2023), da die Berechnung der Inversen $(X^T X)^{-1}$ mit steigender Datenmenge aufwendiger zu berechnen ist.

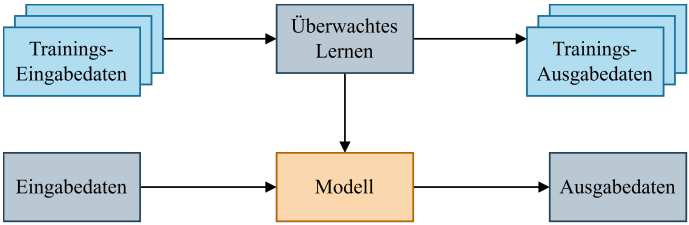



**Abbildung 4.2:** Überwachtes Lernen - Lernprozess mit Trainingsdatensatz und annotierten Daten (vgl. A. Zhang u. a. 2023)

Überwachtes Lernen findet breite Anwendung in Bereichen wie Bild- und Spracherkennung, medizinischer Diagnose und Finanzmarktanalyse (Boyarshinov 2005). Die Vorteile dieser Methode liegen in der Präzision und gleichzeitig effizienten Implementierung bei klar definierten Problemen, ihrer Fähigkeit zur Generalisierung und der einfachen Interpretierbarkeit der Ergebnisse.

Die Effektivität hängt aber stark von der Qualität und Quantität der verfügbaren Trainingsdaten ab.

Ein klassisches Anwendungsbeispiel von überwachtem Lernen ist die Klassifizierung von E-Mails als Spam oder Nicht-Spam (Bratko u. a. 2006). Hierbei lernt das Modell anhand einer Menge von E-Mails, die als Spam oder Nicht-Spam annotiert sind, die charakteristischen Merkmale von Spam-E-Mails zu erkennen. Nach dem Training kann das Modell neue und bisher unbekannte E-Mails analysieren und vorhersagen, ob diese Spam sind oder nicht.

Während statistische Methoden bei mathematisch klar definierten Problemen oft effizienter sind, zeigt sich in aktuellen ML-Anwendungen ein Trend zu Deep Learning (DL)-basierten Ansätzen (siehe Abschnitt 4.3.1). Diese sind besser geeignet, mehrdimensionale und weniger strukturierte Probleme zu lösen, was durch die zunehmende Verfügbarkeit von Rechenleistung praktisch umsetzbar geworden ist.

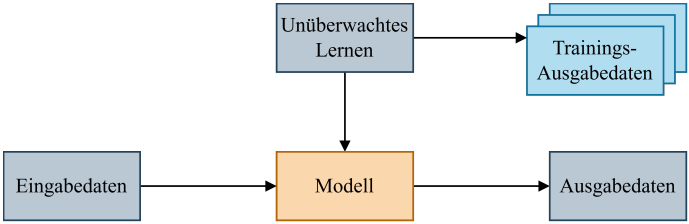



**Abbildung 4.3:** Unüberwachtes Lernen - Lernprozess ohne annotierte Daten (vgl. A. Zhang u. a. 2023)

### 4.1.2 Unüberwachtes Lernen (Unsupervised Learning)

Unüberwachtes Lernen (Unsupervised Learning) stellt die andere zentrale Methode des maschinellen Lernens dar und befasst sich mit der Analyse und Interpretation von nicht annotierten Daten (Becker und Plumbley 1996; G. Hinton und Sejnowski 1999). Im Gegensatz zum überwachten Lernen gibt es hier keine bekannten und annotierten Ausgabevariablen oder Labels, anhand derer das Modell trainiert wird. Stattdessen liegt der Schwerpunkt auf der Identifikation von Mustern, Strukturen oder Zusammenhängen direkt aus den Daten (Hastie, Tibshirani und J. H. Friedman 2009).

Das Modell schätzt hierbei die Wahrscheinlichkeitsverteilung $P(\mathbf{X})$ der Eingabedaten. Diese Verteilung beschreibt, wie häufig bestimmte Datenpunkte oder Datenmuster auftreten, ohne dass eine Zuordnung zu vordefinierten Ausgabevariablen $Y$ erforderlich ist. Beispielsweise könnte in einem Bildverarbeitungssystem die Verteilung erfassen, welche Pixelmuster besonders häufig vorkommen, oder in einem System zur Kundenanalyse, welche Kombinationen von Kundeneigenschaften typisch sind. In unüberwachten Lernprozessen werden Algorithmen eingesetzt, um diese verborgenen Strukturen in den Daten zu finden. Diese Methoden sind besonders nützlich, um Einsichten in Daten zu gewinnen, deren Beziehungen oder Klassifikationen im Vorfeld nicht klar definiert sind.

Die wichtigsten Methoden des unüberwachten Lernens umfassen:

- **Clusteranalyse:** Gruppierung ähnlicher Datenpunkte zur Identifikation natürlicher Gruppierungen.
- **Dimensionsreduktion:** Verringerung der Anzahl der Merkmale bei gleichzeitiger Beibehaltung wesentlicher Informationen zur effizienteren Datenverarbeitung.
- **Assoziationsregel-Lernen:** Entdeckung von Beziehungen zwischen Variablen in großen Datensätzen zur Aufdeckung versteckter Zusammenhänge.

Die Herausforderung bei unüberwachtem Lernen besteht darin, dass die Ergebnisse oft schwerer zu interpretieren sind als beim überwachten Lernen, da es keine eindeutigen Labels gibt, die als Orientierung dienen könnten. Dies erfordert eine sorgfältige Analyse und oft auch eine nachträgliche Bewertung durch Experten.

Unüberwachtes Lernen findet in verschiedenen Anwendungsbereichen wichtige praktische Verwendung (Hastie, Tibshirani und J. H. Friedman 2009). In der Kundensegmentierung ermöglicht es die Gruppierung von Kunden basierend auf ihrem Kaufverhalten, demografischen Merkmalen und Präferenzen, was Unternehmen die Entwicklung zielgerichteter Marketingstrategien ermöglicht.

Ein weiteres bedeutendes Anwendungsgebiet ist die Anomalieerkennung in Netzwerken oder Finanzsystemen, wo ungewöhnliche Muster oder Ausreißer frühzeitig identifiziert werden können, um potenzielle Sicherheitsbedrohungen oder Betrugsversuche aufzudecken (Becker und Plumbley 1996). Besonders wertvoll ist unüberwachtes Lernen auch in der explorativen Datenanalyse, wo es bei der Entdeckung und Visualisierung unbekannter Muster hilft, insbesondere wenn keine vordefinierten Kategorien existieren.

### 4.1.3 Semiüberwachtes Lernen (Semi-supervised Learning)

Semiüberwachtes Lernen stellt eine Brücke zwischen überwachtem und unüberwachtem Lernen dar, indem es die Vorteile beider Ansätze kombiniert (Engelen und Hoos 2020). Diese Methode nutzt eine große Menge nicht annotierter Daten zusammen mit einer kleineren Menge annotierter Daten. Der Einsatz nicht annotierter Daten ist besonders wertvoll, da das Sammeln gelabelter Daten oft aufwendig und kostspielig ist (X. Zhu 2008).

Aus probabilistischer Sicht lässt sich semiüberwachtes Lernen durch die Wahrscheinlichkeitsverteilung $P(\mathbf{X})$ für nicht annotierte Daten und die bedingte Wahrscheinlichkeitsverteilung $P(\mathbf{Y}|\mathbf{X})$ für gelabelte Daten beschreiben (Engelen und Hoos 2020; X. Zhu 2008). Eine zentrale Annahme ist, dass $P(\mathbf{X})$ Informationen über $P(\mathbf{Y}|\mathbf{X})$ enthält, was die Nutzung nicht annotierter Daten zur Verbesserung des Modells ermöglicht (Goodfellow, Bengio und Courville 2016; Engelen und Hoos 2020).

Verschiedene Annahmen formalisieren die Interaktion zwischen $P(\mathbf{X})$ und $P(\mathbf{Y}|\mathbf{X})$ (Engelen und Hoos 2020):

- **Glättungsannahme (smoothness assumption)**: Ähnliche Eingabevariablen sollten zu ähnlichen Ausgabevariablen führen. Dies bedeutet, dass die Vorhersagefunktion in Bereichen mit hoher Datendichte stetig differenzierbar sein sollte, also keine abrupten Sprünge oder Unstetigkeiten aufweist.
- **Niedrigdichteannahme (low-density assumption)**: Die Entscheidungsgrenze sollte durch Bereiche niedriger Dichte im Eingaberaum verlaufen. Dies impliziert, dass Klassenübergänge in Regionen mit wenigen Datenpunkten stattfinden.

- **Mannigfaltigkeitsannahme (manifold assumption)**: Obwohl die Daten in einem hochdimensionalen Raum repräsentiert sind (beispielsweise ein Bild mit tausenden von Pixeln), können die relevanten Variationen oft durch deutlich weniger Dimensionen und Parameter beschrieben werden. Diese Parameter spannen eine niedrigdimensionale, gekrümmte Fläche (Mannigfaltigkeit) im hochdimensionalen Pixelraum auf. Die Erkennung und Nutzung solcher niedrigdimensionalen Strukturen ermöglicht eine effizientere Verarbeitung und bessere Generalisierung.
- **Cluster-Annahme (cluster assumption)**: Wenn Punkte im selben Cluster liegen, gehören sie wahrscheinlich zur selben Klasse. Dies erlaubt es, Informationen von gelabelten auf ungelabelte Datenpunkte innerhalb eines Clusters zu übertragen.

Diese Annahmen bilden die Grundlage für semi-überwachte Lernalgorithmen und tragen zur Verbesserung der Modellleistung bei (Engelen und Hoos 2020).

Ein anschauliches Beispiel für semiüberwachtes Lernen findet sich in der natürlichen Sprachverarbeitung (Natural Language Processing (NLP)). Hier wird dieser Ansatz durch die Kombination von unüberwachtem Pre-Training und überwachtem Fein-Tuning beim Training von Transformer-Modellen eingesetzt (Radford, Narasimhan u. a. 2018). Dies wird in den Abschnitten 4.4.4, 4.4.6 und insbesondere 4.4.6.2 noch weiter vertieft. Diese Methode steigert die Effizienz des Lernprozesses, indem sie die in nicht annotierten Daten enthaltenen Informationen nutzt, um die Leistung des Modells in der überwachten Lernphase zu verbessern.

### 4.1.4 Selbstüberwachtes Lernen (Self-supervised Learning)

Selbstüberwachtes Lernen ist eine Methode des ML, die ebenfalls die Vorteile von überwachtem und unüberwachtem Lernen kombiniert. Im Gegensatz zum semi-überwachten Lernen nutzt es aber keine annotierten Daten, sondern generiert automatisch und selbstständig Pseudoannotationen bzw. Trainingssignale (Albelwi 2022; X. Liu u. a. 2021).

Der Kerngedanke besteht darin, aus den inhärenten Strukturen der Daten selbst ein Verbesserungssignal im Training zu konstruieren. Dies ermöglicht die Nutzung großer ungelabelter Datensätze ohne den aufwendigen Prozess manueller Annotation. Ein typisches Beispiel ist das Masked Language Model (Maskiertes Sprachmodell, MLM) (siehe Abschnitt 4.13), wie es in BERT (Devlin, Chang und Toutanova 2019) verwendet wird.

Selbstüberwachtes Lernen hat sich besonders in der Verarbeitung natürlicher Sprache und Bildverarbeitung als effektiv erwiesen.

## 4.2 Neuronale Netze

Neuronale Netze stellen einen fundamentalen Ansatz des maschinellen Lernens dar und bilden die technische Grundlage für DL. Während Lernverfahren wie überwachtes oder unüberwachtes Lernen die methodische Herangehensweise beschreiben, bilden neuronale Netze die konkrete technische Implementierung dieser Methoden.

Diese Netze sind von der biologischen Struktur des menschlichen Gehirns inspiriert und bestehen aus einer Vielzahl künstlicher „Neuronen“ (Knoten), die in Schichten angeordnet sind und Informationen verarbeiten können. Dennoch ist das menschliche Gehirn weitaus komplexer. Moderne Large Language Models (Große Sprachmodelle, LLMs) wie GPT-3 verfügen über etwa 175 Milliarden Parameter, während das menschliche Gehirn schätzungsweise 86 Billionen synaptische Verbindungen aufweist (Brown u. a. 2020; Herculano-Houzel 2009). Diese Gegenüberstellung verdeutlicht sowohl die bereits erreichte Komplexität künstlicher neuronaler Netze als auch das noch bestehende Potenzial für weitere Entwicklungen.

Die Funktionsweise neuronaler Netze basiert auf dem Zusammenspiel verschiedener Komponenten und Mechanismen. Zentral sind der Feed-Forward-Mechanismus zur Informationsverarbeitung und die Aktivierungsfunktionen.

Das Training dieser Netze erfolgt durch den Backpropagation-Algorithmus, der die schrittweise Anpassung der Netzwerkparameter ermöglicht. Diese Elemente werden in den folgenden Abschnitten detailliert erläutert und in ihren funktionalen Zusammenhang gestellt.

### 4.2.1 Grundlagen und Architektur

Ein Neural Network (NN) lässt sich mathematisch als gerichteter Graph $G = (V, A)$ darstellen, wobei $V$ die Menge der Knoten (Neuronen) und $A \subseteq V \times V$ die Menge der gerichteten Kanten repräsentiert. Die Knoten sind, wie in Abbildung 4.5 illustriert, in mehreren aufeinanderfolgenden Schichten angeordnet und über gewichtete Kanten miteinander verbunden. Jedes Neuron verarbeitet eingehende Informationen und leitet sie entlang der Kanten weiter. Der gesampte Prozess ist in Abbildung 4.4 abgebildet.

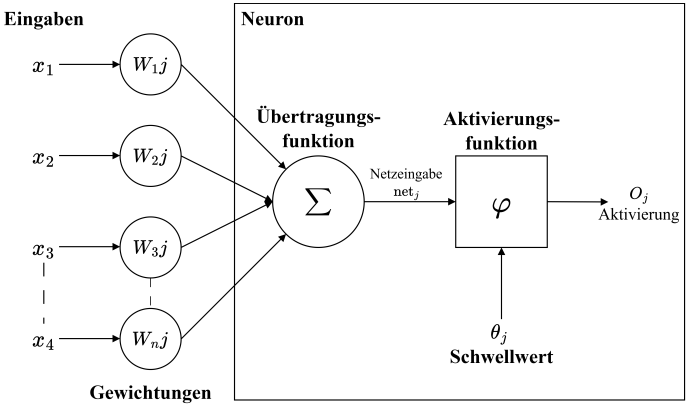



**Abbildung 4.4:** Aufbau eines Neural Networks

Die Architektur eines NN besteht aus drei grundlegenden Schichttypen: Die Eingabeschicht (in Abbildung 4.5 grün dargestellt) nimmt die zu verarbeitenden Daten auf und führt sie dem Netzwerk zu.
Diese Daten werden dann durch eine oder mehrere verborgene Schichten (grau dargestellt) verarbeitet. Jedes Neuron in diesen Schichten führt zunächst eine gewichtete Summierung seiner Eingangssignale durch und addiert einen Bias-Wert, der als zusätzlicher Parameter die Aktivierungsschwelle des Neurons beeinflusst.

Auf diese Summe wird dann eine Aktivierungsfunktion (siehe Abschnitt 4.2.2) angewendet. Durch diese Art der Signalverarbeitung können die verborgenen Schichten mehrdimensionale Muster in den Daten erkennen. Die finale Ausgabeschicht (blau dargestellt) erzeugt schließlich das Ergebnis der Verarbeitung in der gewünschten Form.

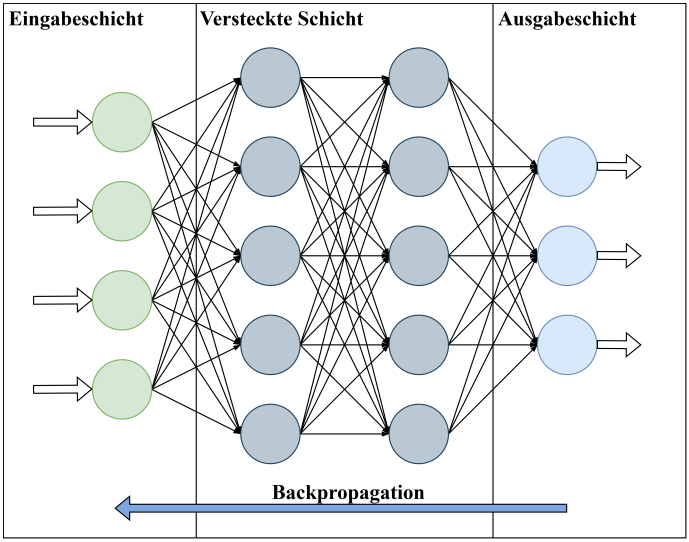



**Abbildung 4.5:** Neuronales Feed-Forward-Netz mit zwei versteckten Schichten

Tiefe neuronale Netze (siehe Abschnitt 4.3.1) haben gegenüber einfachen neuronalen Netzen mehrere verborgene Schichten (Anzahl $> 0$) (L. Deng, G. Hinton und Kingsbury 2013; Montavon, Samek und Müller 2018). In einem Feed-Forward-Netzwerk fließen Informationen ausschließlich von der Eingabe- zur Ausgabeschicht, ohne Rückkopplungen (Goodfellow, Bengio und Courville 2016).

Die gewichtete Summe für alle Neuronen einer Schicht, die vollständig mit der vorherigen Schicht verbunden ist, lässt sich mathematisch wie folgt ausdrücken:

$$\mathbf{z} = \mathbf{W}\mathbf{a} + \mathbf{b} \tag{4.4}$$

Hierbei bezeichnet $\mathbf{z}$ den Vektor der gewichteten Summen, $\mathbf{W}$ die Gewichtsmatrix, $\mathbf{a}$ den Aktivierungsvektor der vorherigen Schicht und $\mathbf{b}$ den Bias-Vektor. Diese mathematische Formulierung bildet die Grundlage für die Berechnungen und Lernprozesse in neuronalen Netzen, die in den folgenden Abschnitten detaillierter betrachtet werden.

### 4.2.2 Aktivierungsfunktionen

Aktivierungsfunktionen sind ein essentieller Bestandteil neuronaler Netze. Sie führen Nichtlinearität in die Netzwerkausgabe ein und ermöglichen es dem Netzwerk, komplexe Beziehungen zwischen Eingangs- und Ausgangsvariablen zu modellieren (Dubey, Singh und Chaudhuri 2022; A. Zhang u. a. 2023). Die Ausgabevariable eines Neurons $i$ wird berechnet, indem die gewichtete Summe $net_i$ durch die Aktivierungsfunktion geleitet wird:

$$o_i = \text{activation}(net_i) \tag{4.5}$$

Im Folgenden werden drei zentrale Aktivierungsfunktionen detailliert vorgestellt.

**Rectified Linear Unit (ReLU):**

$$\text{ReLU}(net_i) = \max(0, net_i) = \begin{cases} net_i, & \text{wenn } net_i \geq 0 \\ 0, & \text{sonst} \end{cases} \tag{4.6}$$

Rectified Linear Unit (ReLU) ist eine der am häufigsten verwendeten Aktivierungsfunktionen in tiefen neuronalen Netzwerken (Goodfellow, Bengio und Courville 2016; Nair und Geoffrey E. Hinton 2010). Ihre Popularität begründet sich in ihrer Einfachheit und rechnerischen Effizienz. ReLU reduziert die Komplexität der Berechnungen erheblich und erzielt gleichzeitig gute Leistungen in verschiedenen Anwendungen (Jakub und Nica 2023; Jarrett u. a. 2009; Glorot, Bordes und Bengio 2011).

Die ReLU-Funktion gibt den Eingabewert unverändert weiter, wenn dieser positiv ist, und setzt negative Werte auf null. Ein Nachteil dieser Funktion ist das „dying ReLU"-Problem, bei dem Neuronen für alle Eingabevariablen inaktiv werden können, wenn sie in einen Zustand geraten, in dem sie nur noch negative Werte ausgeben (L. Lu u. a. 2020).

**Sigmoidfunktion:**

$$\text{sigmoid}(net_i) = \frac{1}{1 + \exp(-net_i)} \tag{4.7}$$

Die Sigmoidfunktion bildet Eingabewerte auf den Bereich (0, 1) ab und liefert eine stetig differenzierbare, kontinuierliche Ausgabe.
Diese Eigenschaft macht sie besonders geeignet für die Ausgangsschicht von Modellen, die Wahrscheinlichkeiten vorhersagen müssen. Ein typisches Beispiel sind binäre Klassifikationsaufgaben, bei denen die Ausgabe als Wahrscheinlichkeit für die Zugehörigkeit zu einer von zwei Klassen interpretiert werden kann.

Ein Nachteil der Sigmoidfunktion ist das Problem der verschwindenden Gradienten bei sehr großen oder sehr kleinen Eingabewerten, was das Training tiefer Netzwerke erschweren kann (Hochreiter und Schmidhuber 1997).

**Tangens Hyperbolicus (tanh):**

$$\tanh(net_i) = \frac{\exp(net_i) - \exp(-net_i)}{\exp(net_i) + \exp(-net_i)} \tag{4.8}$$

Die tanh-Funktion bildet Eingabevariablen auf den Bereich (-1, 1) ab und ist nullzentriert. Diese Eigenschaft macht sie für viele neuronale Netze vorteilhaft, da sie eine symmetrische Behandlung positiver und negativer Werte ermöglicht (Kalman und Kwasny 1992). Ähnlich wie die Sigmoidfunktion kann auch die tanh-Funktion bei extremen Eingabewerten zu verschwindenden Gradienten führen.

Die Wahl der Aktivierungsfunktion hängt von der spezifischen Aufgabe und der Architektur des neuronalen Netzes ab. Während ReLU oft in versteckten Schichten verwendet wird, finden Sigmoid und tanh häufig Anwendung in Ausgabeschichten oder in speziellen Architekturen wie Recurrent Neural Networks (RNNs). RNNs sind Netzwerke, die durch rückgekoppelte Verbindungen zwischen Neuronen Informationen über Zeitschritte hinweg speichern können, was sie besonders für die Verarbeitung sequentieller Daten wie Text oder Zeitreihen geeignet macht.

### 4.2.3 Lernprozess

Der Lernprozess in einem NN umfasst mehrere Schritte: den **Forward Pass**, die **Berechnung der Verlustfunktion**, die **Backpropagation** und die **Parameteranpassung**. Diese Schritte werden iterativ durchgeführt, um die Leistung des Netzes kontinuierlich zu verbessern.

#### Forward Pass

Im Forward Pass werden die Eingabedaten durch das Netzwerk geleitet. Jedes Neuron berechnet die gewichtete Summe seiner Eingangssignale, addiert einen Bias-Wert und wendet die Aktivierungsfunktion an. Dieser Prozess setzt sich durch alle Schichten des Netzwerks fort, bis die Ausgabeschicht erreicht ist.

**Berechnung der Verlustfunktion**

Nach dem Forward Pass wird die Verlustfunktion berechnet, die den Unterschied zwischen den vorhergesagten Ausgabevariablen des Netzwerks und den tatsächlichen Ausgabevariablen quantifiziert. Das Ziel des Trainings besteht darin, diese Verlustfunktion zu minimieren.

Die allgemeine Form der Verlustfunktion eines neuronalen Netzes kann durch den durchschnittlichen Fehler über alle Trainingsdaten ausgedrückt werden (Z. Zhang und Sabuncu 2018; Goodfellow, Bengio und Courville 2016):

$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} L(x_i, y_i, \theta) \tag{4.9}$$

Hierbei bezeichnet $n$ die Anzahl der Trainingsdaten, $\theta$ die Modellparameter und $L(x_i, y_i, \theta)$ den Fehler für ein einzelnes Eingabe-/Ausgabevariablenpaar $(x_i, y_i)$.

**Backpropagation**

Die Backpropagation ist der zentrale Prozess, durch den neuronale Netze aus ihren Fehlern lernen (Rumelhart, Geoffrey E. Hinton und Williams 1986). Dabei werden die Gradienten der Verlustfunktion bezüglich aller Netzwerkparameter (Gewichte und Biases) berechnet. Diese Gradienten zeigen eine Richtung an, in der die Parameter angepasst werden können, um den Fehler zwischen tatsächlicher und vorhergesagter Ausgabe zu reduzieren. Der Gradient zeigt lediglich eine lokale Verbesserungsmöglichkeit an und führt nicht notwendigerweise direkt zum globalen Minimum der Verlustfunktion.

Der Backpropagation-Algorithmus basiert auf der Kettenregel der Differentiation, die es ermöglicht, den Gradienten einer zusammengesetzten Funktion zu berechnen. Da ein neuronales Netz aus mehreren hintereinander geschalteten Schichten besteht, wird die Kettenregel verwendet, um den Einfluss jedes Parameters auf den Gesamtfehler zu bestimmen. Der Algorithmus arbeitet sich von der Ausgabeschicht rückwärts durch das Netzwerk.

Der Prozess umfasst drei wesentliche Schritte:

1. Berechnung der Gradienten in der Ausgabeschicht
2. Rekursive Berechnung der Gradienten in den versteckten Schichten durch Anwendung der Kettenregel
3. Aktualisierung aller Gewichte basierend auf den berechneten Gradienten

Die durch Backpropagation berechneten partiellen Ableitungen der Verlustfunktion nach den Netzwerkparametern beschreiben den Gradienten des Fehlers für ein einzelnes Trainingspaar. Für die Anpassung des Gesamtmodells wird der Durchschnitt der Gradienten über alle Trainingspaare in einem Batch (einer Teilmenge der Trainingsdaten) berechnet. Dieser gemittelte Gradient wird dann für die Parameteranpassung verwendet.

**Anpassung der Parameter**

Die Parameteranpassung erfolgt nach der Gradientenberechnung durch Backpropagation, typischerweise nach der Verarbeitung eines Batches (einer Teilmenge der Trainingsdaten).

Die grundlegende Update-Regel für die Parameteranpassung lautet:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L(\theta_t) \tag{4.10}$$

wobei $\theta_t$ die aktuellen Parameter, $\alpha$ die Lernrate, die die Schrittweite der Anpassung bestimmt, und $\nabla_\theta L(\theta_t)$ der Gradient der Verlustfunktion bezüglich der Parameter ist.

Für die Anpassung der Parameter haben sich verschiedene Methoden etabliert (Goodfellow, Bengio und Courville 2016):

- *Mini-Batch-Optimierung:* Der Gradient wird für kleine Teilmengen der Trainingsdaten berechnet, was häufigere Parameter-Updates ermöglicht.
- *Stochastic Gradient Descent:* Ein Spezialfall der Mini-Batch-Optimierung mit einem Trainingsbeispiel pro Batch.
- *Adaptive Optimierungsverfahren:* Algorithmen wie Adam (Adaptive Moment Estimation) passen die Lernrate für jeden Parameter individuell an (Kingma und Ba 2015). Diese Anpassung ermöglicht es dem Anpassungsprozess, lokale Minima der Verlustfunktion effektiver zu überwinden.

Das Training wird beendet, wenn die Änderung der Verlustfunktion einen definierten Schwellenwert unterschreitet oder eine festgelegte Anzahl von Durchläufen durch den Trainingsdatensatz (Epochen) erreicht ist. Um eine Überanpassung des Modells an die Trainingsdaten („*Overfitting*“) zu vermeiden, wird der Fehler auf separaten Validierungsdaten überwacht (siehe Abschnitt 4.2.3).

### 4.2.4 Augmentierung

Neuronale Netze verfügen über viele Parameter und damit eine hohe Modellkapazität, die es ihnen ermöglicht, Muster in Daten zu erkennen (A. Zhang u. a. 2023). Diese Kapazität führt jedoch häufig zu Überanpassung (Overfitting) auf den Trainingsdaten, was die Generalisierungsfähigkeit auf ungesehenen Daten beeinträchtigt.

Eine Regularisierungstechnik ist eine Methode zur Kontrolle der Modellkomplexität, die das Überanpassen eines maschinellen Lernmodells an die Trainingsdaten verhindert und damit seine Generalisierungsfähigkeit auf ungesehene Daten verbessert (Goodfellow, Bengio und Courville 2016). Neben den in Abschnitt 4.2.3 beschriebenen Regularisierungstechniken stellt die Datenaugmentierung eine besonders effektive Methode dar (Shorten und Khoshgoftaar 2019), da sie den Trainingsdatensatz künstlich erweitert und dem Modell hilft, invariante Merkmale zu erlernen.

Datenaugmentierung bezeichnet die systematische Erweiterung des Trainingsdatensatzes durch kontrollierte Variationen existierender Daten. Bei der Bildverarbeitung umfasst dies üblicherweise geometrische Transformationen wie Rotation, Skalierung oder Spiegelung sowie photometrische Anpassungen wie Helligkeits- und Kontraständerungen. Diese künstlichen Variationen vergrößern nicht nur effektiv den Trainingsdatensatz, sondern verbessern auch die Robustheit[29] des Modells gegenüber natürlichen Variationen in den Eingabedaten (A. Zhang u. a. 2023).

Im Kontext des GUI-Testings ist diese Robustheit besonders relevant, da GUI-Elemente in verschiedenen Größen, Ausrichtungen und unter verschiedenen Beleuchtungsbedingungen korrekt erkannt werden müssen.

[29] Robustheit bezeichnet in diesem Kontext die Fähigkeit eines Modells, auch bei leichten Veränderungen der Eingabedaten stabile und korrekte Vorhersagen zu treffen.

## 4.3 Erweiterte Methoden des maschinellen Lernens

Dieses Kapitel behandelt erweiterte Techniken des maschinellen Lernens, die über die grundlegenden Ansätze hinausgehen und den aktuellen Stand der Forschung und Anwendung repräsentieren (A. Zhang u. a. 2023).

Diese Methoden erweitern die in den vorherigen Abschnitten beschriebenen Grundlagen, um Problemstellungen mit hoher kombinatorischer Komplexität oder nichtlinearen Abhängigkeiten zu bewältigen.

Im Fokus stehen:

- Deep Learning (Tiefes Lernen, DL), das durch mehrschichtige neuronale Netzwerke die Modellierung hochdimensionaler Daten und nichtlinearer Beziehungen ermöglicht (LeCun, Bengio und G. Hinton 2015).
- Transfer Learning, eine Methode zur systematischen Übertragung gelernter Repräsentationen zwischen verwandten Aufgaben (S. J. Pan und Q. Yang 2010).
- Maschinelles Sehen, ein Teilgebiet zur automatisierten Analyse und Interpretation visueller Daten (Jain, Kasturi, Schunck u. a. 1995).
- Mixture of Experts (MoE), ein Ansatz zur effizienten Modellierung durch spezialisierte Teilmodelle (Shazeer u. a. 2017).

Diese erweiterten Methoden bauen auf den Grundlagen des maschinellen Lernens auf und ermöglichen die Verarbeitung größerer Datenmengen sowie die Modellierung vielschichtiger Zusammenhänge. Sie finden insbesondere im Kontext von LLMs und der Analyse hochdimensionaler Daten Anwendung.

### 4.3.1 Deep Learning

Deep Learning (Tiefes Lernen, DL) ist eine Spezialisierung des maschinellen Lernens, die auf tiefen neuronalen Netzwerken basiert. Im Gegensatz zu den in Abschnitt 4.2 beschriebenen einfachen neuronalen Netzen verfügen DL-Modelle über mehrere versteckte Schichten (Hidden Layers). Die Tiefe eines neuronalen Netzes bezieht sich auf die Anzahl der versteckten Schichten zwischen Eingabe- und Ausgabeschicht. Bereits Netzwerke mit mehr als einer versteckten Schicht können als „tief“ bezeichnet werden. Moderne Architekturen weisen jedoch oft Dutzende oder sogar Hunderte von Schichten auf. Diese Architektur ermöglicht die Modellierung von Beziehungen in den Daten, wie beispielsweise die gleichzeitige Berücksichtigung von Farbe, Form und Textur bei der Bilderkennung oder die Analyse von Grammatik und Kontext in der Sprachverarbeitung (Goodfellow, Bengio und Courville 2016; LeCun, Bengio und G. Hinton 2015).

Ein zentrales Merkmal des DL ist die Fähigkeit, mit jeder Schicht zunehmend abstraktere Merkmale zu extrahieren. In der Bildverarbeitung beispielsweise erkennen die ersten Schichten einfache Kanten und Texturen, während tiefere Schichten darauf aufbauend komplexere Muster wie Gesichtszüge oder Objekte identifizieren. Diese Fähigkeit ermöglicht die Verarbeitung von Daten mit vielen Dimensionen (z.B. Bilder mit Millionen von Pixeln) oder ohne offensichtliche Struktur (wie Texte in natürlicher Sprache) (Goodfellow, Bengio und Courville 2016). Zwei spezialisierte Architekturen haben sich für spezifische Anwendungsgebiete als besonders effektiv erwiesen:

- **Convolutional Neural Network (Faltendes Neuronales Netz, CNN):** Diese Netzwerkarchitektur verarbeitet mehrdimensionale, räumlich strukturierte Daten wie Bilder durch Faltungsoperationen. Dies ermöglicht die effiziente Erkennung lokaler Muster unabhängig von ihrer Position im Bild, was Convolutional Neural Networks (CNNs) besonders für die Bildverarbeitung und das maschinelle Sehen qualifiziert (Krizhevsky, Sutskever und Geoffrey E Hinton 2012).
- **Recurrent Neural Network (Rekurrentes Neuronales Netz, RNN):** Diese Architektur ist besonders für die Verarbeitung von Datensequenzen wie Text, Sprache oder Zeitreihendaten geeignet. Durch spezielle Verbindungen zwischen Neuronen können RNNs Informationen aus vorherigen Elementen einer Sequenz (z.B. frühere Wörter in einem Satz oder vergangene Messwerte) für die Analyse des aktuellen Elements nutzen (Hochreiter und Schmidhuber 1997).

Zusätzlich zu diesen Architekturen hat sich die Transformer-Architektur (siehe Abschnitt 4.4.4) als besonders effektiv für die Verarbeitung natürlicher Sprache und verwandter Aufgaben erwiesen.

Für das Training tiefer neuronaler Netze sind spezielle Techniken erforderlich, um mit der hohen Anzahl an Parametern umzugehen:

- **Optimierung:** Die Anpassung der Modellparameter erfolgt durch Verfahren wie SGD und Adam, die den Fehler zwischen vorhergesagten und tatsächlichen Ausgabewerten minimieren. Die große Anzahl an Parametern in tiefen Netzen erfordert besonders effiziente Anpassungsstrategien (Goodfellow, Bengio und Courville 2016).
- **Regularisierung:** Um zu verhindern, dass das Modell die Trainingsdaten lediglich auswendig lernt statt allgemeine Muster zu erkennen, werden Techniken wie Dropout eingesetzt. Dropout deaktiviert zufällig ausgewählte Neuronen während des Trainings, was die Robustheit des Netzes erhöht (Srivastava u. a. 2014).

- **Batch Normalization:** Diese Technik standardisiert die Ausgaben jeder Schicht, sodass sie einen Mittelwert von 0 und eine Standardabweichung von 1 aufweisen. Dies stabilisiert den Lernprozess, da die Eingaben nachfolgender Schichten in einem kontrollierten Wertebereich bleiben (Ioffe und Szegedy 2015).

Diese technischen Aspekte des Deep Learning sind für das Verständnis der grundlegenden Funktionsweise wichtig, spielen aber für die weiteren Betrachtungen in dieser Arbeit eine untergeordnete Rolle.

Die Architektur tiefer neuronaler Netze kann je nach Anwendungsfall stark variieren. Die Wahl der Anzahl und Art der Schichten sowie der Verbindungen zwischen diesen hat einen erheblichen Einfluss auf die Leistungsfähigkeit und Effizienz des Modells (He u. a. 2016). Die Entwicklung besserer Architekturen für spezifische Aufgaben ist ein aktives Forschungsgebiet im DL.

Im Kontext des automatisierten Testens mobiler Applikationen bietet DL vielversprechende Möglichkeiten. CNNs können beispielsweise zur Analyse von Bildschirminhalten und zur Erkennung von GUI-Elementen eingesetzt werden, während RNNs und Transformer-basierte Modelle für die Verarbeitung von Testsequenzen und die Generierung von Testfällen genutzt werden können. Die Fähigkeit des DL, komplexe Muster in großen Datenmengen zu erkennen, macht es zu einem leistungsfähigen Werkzeug für die Automatisierung und Verbesserung von Testprozessen.

### 4.3.2 Transfer Learning

Transfer Learning ermöglicht die Übertragung von Wissen zwischen verschiedenen Domänen und Aufgaben (S. J. Pan und Q. Yang 2010). Formal lässt sich dies durch eine Quelldomäne $\mathcal{D}_S$ mit zugehöriger Aufgabe $\mathcal{T}_S$ und eine Zieldomäne $\mathcal{D}_T$ mit Aufgabe $\mathcal{T}_T$ beschreiben ($S$ für *Source* und $T$ für *Target*). Das Ziel ist es, die Leistung bei der Zielaufgabe $\mathcal{T}_T$ durch die Nutzung des in $\mathcal{D}_S$ und $\mathcal{T}_S$ erworbenen Wissens zu verbessern. Dies ist besonders wertvoll, wenn in der Zieldomäne nur begrenzte Trainingsdaten verfügbar sind.

Der Prozess des Transfer Learnings umfasst typischerweise folgende Schritte (A. Zhang u. a. 2023):

1. **Vortraining:** Ein neuronales Netzwerk wird auf einem umfangreichen Quelldatensatz trainiert.
2. **U̇bertragung:** Die gelernten Merkmale oder Parameter werden in ein Modell für die Zielaufgabe integriert.
3. **Anpassung:** Das Modell wird auf den Zieldatensatz feinabgestimmt, wobei die übertragenen Merkmale als Basis dienen.

Ein wesentlicher Vorteil des Transfer Learnings liegt in der Wiederverwendung gelernter Merkmale. In der Bildverarbeitung beispielsweise lernen die ersten Schichten eines neuronalen Netzes häufig grundlegende visuelle Merkmale wie Kanten oder Texturen, die domänenübergreifend nützlich sind (Yosinski u. a. 2014). Durch die Nutzung dieser vortrainierten Merkmale können auch bei kleinen Zieldatensätzen gute Ergebnisse erzielt werden, da weniger aufgabenspezifische Trainingsdaten benötigt werden.

Es lassen sich vier grundlegende Kategorien des Wissenstransfers unterscheiden (Weiss, Khoshgoftaar und D. Wang 2016):

1. **Instance-based Transfer:** Dieser Ansatz passt die Gewichtung einzelner Trainingsbeispiele aus der Quelldomäne an, um sie für das Training in der Zieldomäne nutzbar zu machen. Dies ist besonders effektiv, wenn die grundlegenden Zusammenhänge in beiden Domänen ähnlich sind, wie zum Beispiel bei der Übertragung von Bilderkennungsmodellen zwischen verschiedenen Arten von Fotografien.

2. **Feature-based Transfer:** Diese Kategorie umfasst zwei Ansätze zur Übertragung von Merkmalen:

   - *Asymmetrische Anpassung:* Die gelernten Merkmale aus der Quelldomäne werden gezielt transformiert, um sie an die Eigenschaften der Zieldomäne anzupassen. Ein Beispiel wäre die Anpassung eines für Farbbilder trainierten Modells an Schwarz-Weiß-Bilder.

   - *Gemeinsame Merkmalsräume:* Es wird eine Darstellung gesucht, die für beide Domänen gleichermaßen gültig ist. Dies ermöglicht beispielsweise die Übertragung von Wissen zwischen verschiedenen Sprachen in der maschinellen Übersetzung.

3. **Parameter-based Transfer:** Diese Methoden übertragen die gelernten Parameter (Gewichte) direkt zwischen Modellen. Ein typisches Beispiel ist die Verwendung vortrainierter Bilderkennungsmodelle als Ausgangspunkt für neue, verwandte Aufgaben.

4. **Relational-based Transfer:** Dieser Ansatz überträgt Wissen basierend auf logischen Beziehungen zwischen den Domänen, wird aber aufgrund seiner Komplexität seltener eingesetzt.

In dieser Arbeit findet Transfer Learning in zwei Kontexten Anwendung. Es wird zum einen zur Feinabstimmung von LLMs auf spezifische Testgenerierungsaufgaben verwendet, wie in Abschnitt 4.4.6.2 beschrieben. Zum anderen kommt es im Bereich des RL zum Einsatz, wo vortrainierte Bildverarbeitungsmodelle die Objekterkennung in der GUI-Analyse unterstützen (S. J. Pan und Q. Yang 2010).

Transfer Learning ermöglicht nicht nur eine effizientere Nutzung begrenzter Trainingsdaten, sondern kann auch die Generalisierungsfähigkeit der Modelle verbessern. Diese Technik bildet die Grundlage für viele moderne Anwendungen in der Computer Vision, der Verarbeitung natürlicher Sprache und anderen Domänen, wo große vortrainierte Modelle als Ausgangspunkt für spezifische Anwendungen dienen.

### 4.3.3 Maschinelles Sehen

Maschinelles Sehen ist ein Teilgebiet der KI und des ML, das sich mit der automatischen Extraktion, Analyse und Interpretation von Informationen aus Bilddaten befasst. Es ermöglicht Computersystemen, visuelle Sensordaten zu verarbeiten und daraus strukturierte Informationen zu gewinnen. Im Kontext des automatisierten Testens mobiler Applikationen ist insbesondere die Bilderkennung von Bedeutung, da sie die Identifikation und Lokalisation von GUI-Elementen auf der Benutzeroberfläche ermöglicht.

Die moderne Bilderkennung basiert hauptsächlich auf DL-Methoden, die durch groß angelegte Datensätze wie PASCAL VOC (Everingham u. a. 2012) und ImageNet (J. Deng u. a. 2009) ermöglicht wurden (A. Zhang u. a. 2023). Der initiale PASCAL VOC-Datensatz stellte mit 20 annotierten Objektklassen einen wichtigen Meilenstein dar, während der nachfolgende ImageNet-Datensatz mit 1000 Klassen und über einer Million Bildern das Training größerer Modelle ermöglichte. Der Microsoft COCO-Datensatz (Lin u. a. 2014) erweiterte dies zusätzlich um präzise pixelgenaue Annotationen, die insbesondere für die Objektsegmentierung relevant sind.

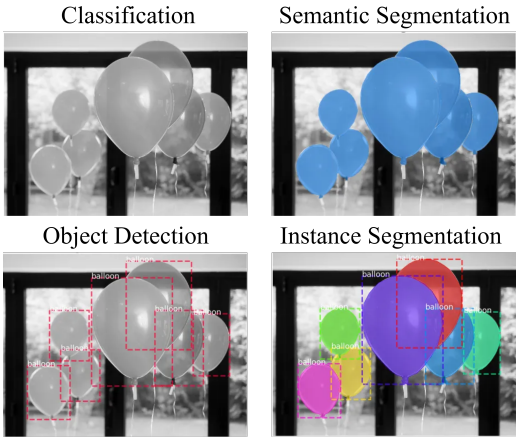



**Abbildung 4.6:** Taxonomie der Bilderkennungsaufgaben im maschinellen Sehen (Abdulla 2017)

Die Bilderkennungsaufgaben unterteilen sich in drei grundlegende Kategorien. Bei der *Klassifizierung* erfolgt die Zuweisung einer spezifischen Klasse zum gesamten Bild, ohne einzelne Bildbereiche separat zu betrachten. Die *Objekterkennung* erweitert dies um die räumliche Komponente, indem sie mehrere Objekte innerhalb eines Bildes durch Begrenzungsboxen lokalisiert und klassifiziert. Die *Segmentierung* als detaillierteste Form der Bildanalyse unterteilt sich in drei Arten:

- **Semantische Segmentierung:** Weist jedem Pixel eine Klassenbezeichnung zu, ohne zwischen Instanzen derselben Klasse zu unterscheiden.
- **Instanzsegmentierung:** Erweitert die semantische Segmentierung durch zusätzliche Unterscheidung einzelner Objektinstanzen innerhalb derselben Klasse.
- **Panoptische Segmentierung (Kirillov u. a. 2019):** Kombiniert beide Ansätze durch instanzbasierte Segmentierung für zählbare Objekte und semantische Segmentierung für Hintergrundelemente.

### YOLO

YOLO (You Only Look Once) repräsentiert eine Familie einstufiger Objekterkennungsalgorithmen, die sich durch ihre Effizienz in der Echtzeitverarbeitung auszeichnen (Redmon u. a. 2016).

Im Gegensatz zu zweistufigen Ansätzen wie R-CNN, die zunächst Regionsvorschläge generieren und diese anschließend klassifizieren, formuliert YOLO die Objekterkennung als direktes Regressionsproblem (Redmon u. a. 2016). Diese direkte Herangehensweise führt zu deutlich schnelleren Verarbeitungszeiten, da das Bild nur einmal durch das neuronale Netz geleitet werden muss. Während R-CNN-basierte Modelle typischerweise mehrere Sekunden pro Bild benötigen, erreicht YOLO Verarbeitungsgeschwindigkeiten in Echtzeit bei vergleichbarer Erkennungsgenauigkeit (Redmon u. a. 2016).
YOLO wird kontinuierlich weiterentwickelt und ist aktuell in Version 11[30].

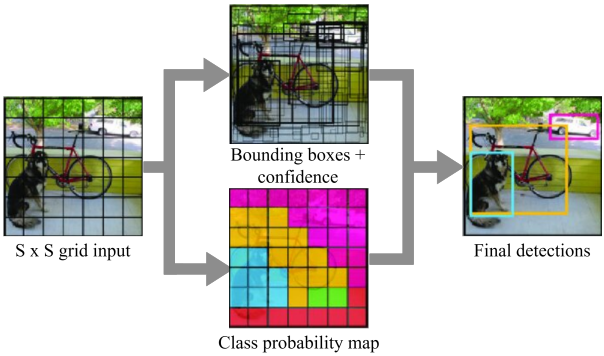



**Abbildung 4.7:** YOLO-Architektur mit Gitterstruktur und Begrenzungsboxen (Redmon u. a. 2016)

[30] YOLOv11 von Ultralytics: Stand Dezember 2024 die aktuellste Version, die auf der YOLO Architektur aufbaut (`https://github.com/ultralytics/ultralytics`)

Die Architektur basiert auf einem Fully Convolutional Network (FCN) (Q. Chen, Xu und Koltun 2017) und unterteilt das Eingabebild in ein regelmäßiges Raster aus $S \times S$ Zellen. Jede Zelle ist für die Erkennung von Objekten verantwortlich, deren Zentrum in diese Zelle fällt. Das Netzwerk erzeugt für jede Zelle drei Arten von Vorhersagen:

- Begrenzungsboxen: Rechteckige Rahmen, die die Position und Größe erkannter Objekte markieren. Diese werden ausgehend von vordefinierten Ankerboxen berechnet - Standardformen verschiedener Größen und Proportionen, die als Ausgangspunkt für die Objekterkennung dienen.
- Konfidenzwerte: Ein Maß für die Sicherheit der Erkennung. Dieser Wert kombiniert zwei Faktoren:
  - Die Wahrscheinlichkeit, dass ein Objekt vorhanden ist.
  - Den IoU-Wert (Intersection over Union), der die Überlappung zwischen vorhergesagter und tatsächlicher Objektposition angibt. Ein IoU von 1 bedeutet perfekte Übereinstimmung, 0 bedeutet keine Überlappung.
- Klassenwahrscheinlichkeiten: Für jedes erkannte Objekt wird die Wahrscheinlichkeit der Zugehörigkeit zu jeder möglichen Objektklasse (wie Person, Auto, Hund etc.) berechnet.

Im Kontext des GUI-Testings ermöglicht Transfer Learning (siehe Abschnitt 4.3.2) die Nutzung vortrainierter YOLO-Modelle zur effizienten Erkennung von GUI-Elementen. Die hohe Verarbeitungsgeschwindigkeit macht YOLO besonders geeignet für die Echtzeitanalyse von Benutzeroberflächen, was den RL-Agenten ermöglicht, GUI-Elemente schnell zu erkennen und mit ihnen zu interagieren.

### 4.3.4 Mixture of Experts

Mixture of Experts (MoE) ist eine Architektur im maschinellen Lernen, bei der mehrere spezialisierte Teilmodelle („Experten“) zusammenarbeiten, um komplexe Aufgaben wie Sprachübersetzung, Bildklassifizierung oder Code-Generierung zu lösen (Shazeer u. a. 2017; Baldacchino u. a. 2016).

Ein Router-Netzwerk weist Eingabevariablen den am besten geeigneten Experten zu. Diese Architektur ermöglicht eine effektive Modellierung von mehrdimensionalen Zusammenhängen durch die Aufteilung des Eingaberaums in spezialisierte Bereiche.

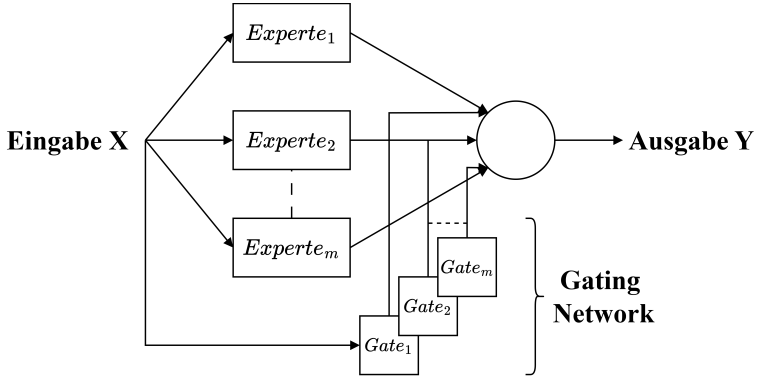



**Abbildung 4.8:** Mixture of Experts (MoE)-Architektur mit Router-Netzwerk und spezialisierten Experten (vgl. Baldacchino u. a. 2016)

Im Kontext von LLMs wurde dieser Ansatz zur Sparse Mixture of Experts (SMoE) weiterentwickelt. SMoE-Modelle aktivieren für jede Eingabe nur eine Teilmenge der verfügbaren Experten, was die Berechnungseffizienz erheblich steigert (Shazeer u. a. 2017). Der entscheidende Vorteil liegt darin, dass die Modellkapazität signifikant erhöht werden kann, ohne dass die Berechnungskosten proportional ansteigen. Dies wird durch die sparsame Aktivierung der Experten erreicht - für jede Eingabe wird nur ein kleiner Teil der Experten tatsächlich verwendet.

Ein aktuelles Beispiel für die SMoE-Architektur ist das Mixtral 8x7B Modell (Jiang, Sablayrolles, Roux u. a. 2024). Die Architektur basiert auf seinem Vorgänger Mistral 7B (Jiang, Sablayrolles, Mensch u. a. 2023), führt aber einen wesentlichen Unterschied ein:

Jede Schicht besteht aus acht Feed-Forward-Blöcken (Experten), wobei ein Router-Netzwerk für jedes Token (die kleinste Verarbeitungseinheit im Modell, beispielsweise ein Wort oder Wortteil vektorisiert) dynamisch zwei dieser Experten auswählt.

Die ausgewählten Experten können sich bei jedem Zeitschritt ändern, wodurch das Modell Zugriff auf insgesamt 47 Milliarden Parameter hat, während pro Inferenz nur etwa 13 Milliarden Parameter aktiv genutzt werden (Jiang, Sablayrolles, Roux u. a. 2024). Die spezielle Architektur von MoE-Modellen ermöglicht eine dynamische Expertenauswahl sowie eine effiziente Parameternutzung durch dünnbesetzte (sparse) Aktivierung.

Durch die Implementierung von Expert Parallelism (EP) wird eine parallele Verarbeitung erreicht (Shazeer u. a. 2017), die durch spezialisierte Kernel für die GPU-Ausführung zusätzlich verbessert wird. Diese technischen Innovationen führen zu einer signifikanten Leistungsverbesserung der Modelle. Die Kombination von EP adressiert außerdem das Load Balancing zwischen den Experten.
Die effiziente Parameternutzung und Parallelisierung ermöglichen es, dass leistungsfähige LLMs auch auf Consumer-Hardware betrieben werden können, wie sie in Abschnitt 4.6.3 beschrieben wird.

Die SMoE-Architektur stellt einen vielversprechenden Ansatz für die Entwicklung effizienter und leistungsfähiger LLMs dar. Bei vergleichbarer Rechenleistung können MoE-Modelle eine um Größenordnungen höhere Modellkapazität erreichen als traditionelle Architekturen (Shazeer u. a. 2017). Die Fähigkeit, große Modelle ressourceneffizient zu betreiben, macht sie besonders interessant für den Einsatz in ressourcenbeschränkten Umgebungen, wie sie beim lokalen Betrieb von KI-gestützten Testsystemen typisch sind.

## 4.4 Large Language Models

Large Language Models (Große Sprachmodelle, LLMs) sind eine bedeutende Entwicklung im Bereich der KI. Diese Modelle, die auf der in Abschnitt 4.4.4 beschriebenen Transformer-Architektur basieren, verarbeiten und generieren natürlichsprachliche Sequenzen verschiedener Längen und linguistischer Strukturen (Vaswani u. a. 2017; Brown u. a. 2020).
LLMs sind neuronale Netze mit Parameterzahlen im Milliardenbereich, die auf umfangreichen Textdatensätzen wie „The Pile“ (Gao u. a. 2020) trainiert werden.

Ihre primäre Aufgabe besteht in der Modellierung der bedingten Wahrscheinlichkeit von Token-Sequenzen, was je nach Architektur durch verschiedene Ansätze wie autoregressive Vorhersage oder maskiertes Lernen realisiert wird. Diese Ansätze werden in den Abschnitten 4.4.3.2 und 4.4.3.3 erläutert.

Neben der Textverarbeitung werden LLMs in verschiedenen Domänen eingesetzt. Im Gebiet des maschinellen Sehens unterstützen sie die Bilderkennung und -beschreibung, während sie bei der Programmierung Codegenerierung und -analyse ermöglichen (Radford, J. Wu, Amodei u. a. 2019).

Die Fähigkeiten eines LLM werden durch drei zentrale Faktoren bestimmt: die Anzahl der Modellparameter, die Größe des Trainingsdatensatzes und die Dauer des Trainings (Kaplan u. a. 2020). Mit steigenden Werten dieser Faktoren verbessert sich typischerweise die Modellleistung, jedoch erhöht sich gleichzeitig der Bedarf an Rechenkapazität. Die aktuelle Forschung konzentriert sich daher auf die Entwicklung effizienter Trainings- und Inferenzmethoden, die in den folgenden Abschnitten behandelt werden:

- Vorverarbeitung und Tokenisierung (Abschnitt 4.4.2).
- Technische Grundlagen der Modellarchitektur (Abschnitt 4.4.3).
- Training und Feinabstimmung (Abschnitt 4.4.6).
- Leistungsbewertung und Metriken (Abschnitt 4.4.9).

Diese Aspekte bilden die Basis für den Einsatz von LLMs im automatisierten Testen mobiler Applikationen.

### 4.4.1 Datenbeschaffung nach CRISP-DM

Die Beschaffung von Trainingsdaten für LLMs orientiert sich am Cross-Industry Standard Process for Data Mining (CRISP-DM)-Prozessmodell (Berry und Linoff 2004; Wirth und Hipp 2000). Dieses Modell strukturiert den Prozess der Datenanalyse in sechs iterative Phasen, wie in Abbildung 4.9 dargestellt.

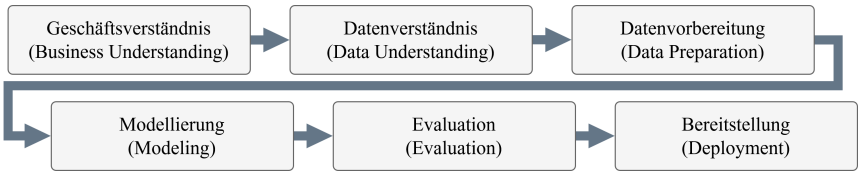



**Abbildung 4.9:** Ablauf des Prozessmodells CRISP-DM (vgl. Berry und Linoff 2004)

Für die Datenbeschaffung sind insbesondere die ersten beiden Phasen maßgeblich.

In der Phase des *Geschäftsverständnisses* werden die projektspezifischen Anforderungen und Ziele definiert. Im Kontext des automatisierten Testens umfasst dies die Definition der Qualitätskriterien für Trainingsdaten sowie die Spezifikation der erforderlichen Merkmale für das Training von LLMs. Diese Phase legt damit die Grundlage für alle nachfolgenden Schritte.

Die Phase des *Datenverständnisses* beinhaltet die systematische Identifikation und Analyse potenzieller Datenquellen, wie beispielsweise existierender Codebeispiele. Diese Phase umfasst die Bewertung der Datenqualität anhand der zuvor definierten Kriterien sowie die Analyse der Datenverteilung und -repräsentativität. Die Qualität der Trainingsdaten beeinflusst direkt die Leistung der resultierenden Modelle.

Der zyklische Charakter des CRISP-DM-Modells ermöglicht eine kontinuierliche Verfeinerung des Prozesses. Die Ergebnisse jeder Phase können zu einer Neubewertung und Anpassung vorheriger Phasen führen. Diese Rückkopplungsschleifen ermöglichen eine iterative Verbesserung der Datenbeschaffung und -aufbereitung, die in den folgenden Abschnitten detailliert beschrieben werden.

### 4.4.2 Vorverarbeitung für LLMs

Die Vorverarbeitung transformiert Rohdaten in ein für LLMs verarbeitbares Format (Chowdhary 2020; A. Zhang u. a. 2023). Dieser Prozess umfasst vier Hauptkomponenten: *Datenaufteilung*, *Tokenisierung*, *Datenbereinigung* und *Anpassung der Sequenzlängen*.

#### 4.4.2.1 Datenaufteilung (Data Splitting)

Die Aufteilung des Datensatzes in Trainings- und Validierungsdaten ermöglicht die Messung der Modellgeneralisierung (A. Zhang u. a. 2023). Der Trainingsdatensatz dient der Parameteranpassung, während der Validierungsdatensatz die Berechnung des Generalisierungsfehlers durch Vergleich von Trainings- und Validierungsverlust ermöglicht (Bengio u. a. 2003) und damit ein Qualitätsmaß der Modelle ermöglicht.

#### 4.4.2.2 Tokenisierung

Die Tokenisierung zerlegt Text in kleinere, berechenbare Einheiten (Song u. a. 2021). Diese Tokens können Wörter, Teilwörter oder Satzzeichen sein. Abbildung 4.10 zeigt die drei grundlegenden Strategien:

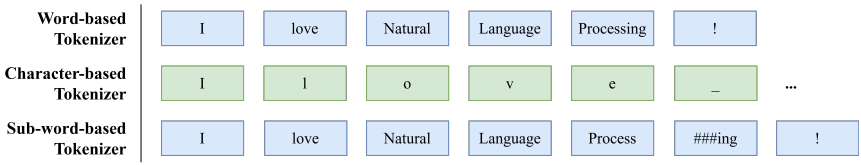



**Abbildung 4.10:** Strategien der Tokenisierung (vgl. Huggingface 2024)

Die *wortbasierte Tokenisierung* zerlegt Text in vollständige Wörter, was bei großen Vokabularen zu Einschränkungen führt. Die *zeichenbasierte Tokenisierung* behandelt jeden Buchstaben als Token, erzeugt jedoch lange Sequenzen. Die *Teilwort-basierte Tokenisierung*, implementiert durch Algorithmen wie Byte-Pair Encoding (Bytepaar-Kodierung, BPE) (Sennrich, Haddow und Birch 2016) oder WordPiece (Song u. a. 2021), bietet einen Kompromiss zwischen Vokabulargröße und Sequenzlänge.

Der Tokenizer wandelt die erzeugten Tokens über eine Lookup-Tabelle in numerische Eingabevektoren um, die das Modell verarbeiten kann. Die Lookup-Tabelle enthält eine feste Zuordnung von Tokens zu numerischen Werten.


**Abbildung 4.11:** Tokenisierte Darstellung des Satzes: „Testen muss nicht aufwendig sein."

Beispielsweise wird der in Abbildung 4.11 gezeigte Satz mit dem subwortbasierten Tokenizer *tiktoken*[31] mit 33 Zeichen in 9 Tokens zerlegt.

#### 4.4.2.3 Datenaufbereitung und -bereinigung

Die Datenaufbereitung basiert auf den in Abschnitt 3.1.1 beschriebenen ISO-25012-Qualitätsmetriken (ISO und IEC 2008). Die Qualität der Eingabedaten beeinflusst die Modellleistung direkt. Der Prozess kann folgende Schritte umfassen:

- **Entfernen irrelevanter Informationen und Formatierungen:** Nicht relevante Daten wie HTML-Tags oder Metadaten werden entfernt.
- **Entfernen niedrigqualitativer Daten:** Um die Gesamtqualität des Trainingsdatensatzes zu erhöhen werden Kriterien für Daten von niedriger Qualität festgelegt und diese entfernt.
- **Normalisierung des Textes:** Der Text wird standardisiert (z.B. Kleinschreibung, Entfernung von Akzenten), um die Konsistenz zu gewährleisten.
- **Behandlung von Sonderzeichen und Zahlen:** Sonderzeichen und Zahlen werden kontextabhängig verarbeitet, um Rauschen zu minimieren.

[31] Tiktoken ist ein Tokenizer für OpenAI Modelle (`https://github.com/openai/tiktoken`)

- **Anonymisierung sensibler Informationen:** Persönliche Daten werden entfernt oder ersetzt, um die Privatsphäre zu schützen.
- **Datenbereinigung:** Es erfolgt eine Prüfung auf Inkonsistenzen, Duplikate und fehlende Daten.

#### 4.4.2.4 Eingabeformate und Sequenzlängen

LLMs erfordern feste Eingabelängen, was spezifische Vorverarbeitungsschritte notwendig macht:

- **Chunking:** Längere Texte werden in Segmente aufgeteilt, die der maximalen Eingabelänge des Modells entsprechen.
- **Padding:** Kürzere Sequenzen werden mit speziellen Padding-Tokens aufgefüllt, um eine einheitliche Länge zu erreichen.
- **Concatenation:** Mehrere Eingaben werden zusammengeführt und durch ein spezielles Zeichen getrennt, das das Ende der Einzeleingaben anzeigt.
- **Truncation:** Sequenzen, die die maximale Länge überschreiten, werden gekürzt.
- **Spezielle Tokens:** Tokens wie „[CLS]“ für Klassifizierung und „[SEP]“ als Separator werden eingeführt, um spezifische Aufgaben zu unterstützen.
- **Maskieren von Tokens:** Für bestimmte Trainingstechniken, insbesondere bei Masked Language Model (MLM) (Devlin, Chang und Toutanova 2019), werden einzelne Tokens vor dem Netzwerk „versteckt“, sodass das Modell diese Tokens „erraten“ muss.

Die genaue Gestaltung der Vorverarbeitungspipeline hängt von den Zielen des Modells, den Eigenschaften der Daten und den spezifischen Herausforderungen der Aufgabe ab (Pengfei Liu u. a. 2023).

### 4.4.3 Technische Grundlagen

Die technischen Grundlagen von LLMs basieren auf fortgeschrittenen Konzepten der Sprachmodellierung und des DL. Aufbauend auf den in Abschnitt 4.1 eingeführten Grundlagen des maschinellen Lernens werden hier die spezifischen Konzepte erläutert, die LLMs das Verstehen und Generieren natürlicher Sprache ermöglichen.

#### 4.4.3.1 Grundlegende Konzepte der Sprachmodellierung

Sprachmodellierung bezeichnet den Prozess der Schätzung von Wahrscheinlichkeitsverteilungen von Wortsequenzen. LLMs nutzen dieses Konzept, um die bedingte Wahrscheinlichkeit einer Sequenz von Tokens innerhalb natürlicher Texte zu schätzen (Scao u. a. 2022; A. Zhang u. a. 2023).

Die fundamentale Aufgabe eines Sprachmodells lässt sich mathematisch wie folgt darstellen:

$$p(\mathbf{x}) = p(x_1, \ldots, x_n) = \prod_{i=1}^{n} p(x_i \mid x_{<i}) \tag{4.11}$$

Hierbei repräsentiert $\mathbf{x}$ die Sequenz von Tokens, und $x_i$ ist das Token an Position $i$, das basierend auf den vorherigen Tokens $x_{<i}$ vorhergesagt wird. Diese Modellierung wird als autoregressiv bezeichnet, da die Vorhersage jedes Elements rekursiv von den vorherigen Elementen abhängt - analog zu autoregressiven Prozessen in der Zeitreihenanalyse, bei denen ein Wert durch eine gewichtete Summe seiner Vorgänger bestimmt wird. In der Sprachmodellierung bedeutet dies, dass jedes Token auf Basis der vollständigen Historie der vorangegangenen Tokens vorhergesagt wird.

Diese autoregressive Modellierung bildet die Grundlage für zwei zentrale Architekturen moderner Sprachmodelle:

#### 4.4.3.2 Causal Language Models

Causal Language Models (Kausale Sprachmodelle, CLMs) sind autoregressive Modelle, die Text in seiner natürlichen Leserichtung verarbeiten. Bei westlichen Sprachen bedeutet dies eine Verarbeitung von links nach rechts, wobei jedes neue Token auf Basis aller vorherigen Tokens vorhergesagt wird. Bekannte Implementierungen dieses Ansatzes finden sich in der GPT-Modellfamilie (Radford, J. Wu, Child u. a. 2019).

Sie weisen jedem Token in einer Sequenz eine Wahrscheinlichkeit zu, indem sie die Kette der vorherigen Tokens berücksichtigen. Abbildung 4.12 zeigt die Architektur eines kausalen Decoders, der für die sequentielle Tokenverarbeitung verwendet wird.

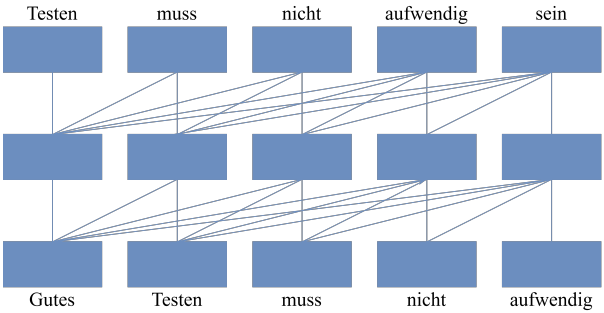



**Abbildung 4.12:** Architektur eines kausalen Decoders für die sequentielle Tokenverarbeitung. Alle Tokens werden sequentiell vorhergesagt und sind mit den vorherigen Tokens verbunden (vgl. Y. Liu u. a. 2024).

Die Wahrscheinlichkeit einer gesamten Tokensequenz wird durch Anwendung der Kettenregel der Wahrscheinlichkeitsrechnung ermittelt:

$$P(\mathbf{x}) = P(x_1) \times P(x_2|x_1) \times \ldots \times P(x_n|x_1, \ldots, x_{n-1}) \tag{4.12}$$

Diese Formel verdeutlicht den sequentiellen Charakter: Die Wahrscheinlichkeit des ersten Tokens $P(x_1)$ wird unabhängig berechnet, während jedes folgende Token auf Basis aller vorherigen Tokens vorhergesagt wird. Beispielsweise hängt die Wahrscheinlichkeit für das zweite Token $P(x_2|x_1)$ vom ersten Token ab, die des dritten Tokens $P(x_3|x_1, x_2)$ von den ersten beiden Tokens, und so weiter.

Diese Architektur wird von State-of-the-Art Modellen (SOTA) wie die GPT-Serie (Brown u. a. 2020; OpenAI u. a. 2023; Ouyang u. a. 2022; Radford, Narasimhan u. a. 2018; Radford, J. Wu, Amodei u. a. 2019), GPT-NeoX (Black u. a. 2022), LLaMA (Touvron, Lavril u. a. 2023; Touvron, Martin u. a. 2023) und BLOOM (Scao u. a. 2022) verwendet und eignet sich besonders für generative Aufgaben wie Code- und Testfallgenerierung.

#### 4.4.3.3 Masked Language Models

Masked Language Models (Maskierte Sprachmodelle, MLMs) implementieren einen grundlegend anderen Ansatz als Causal Language Models (CLMs) zur Verarbeitung von Text. Anstatt Tokens sequentiell vorherzusagen, verwenden sie einen bidirektionalen Ansatz: Einzelne Tokens werden gezielt maskiert (ausgeblendet) und das Modell lernt, diese maskierten Tokens anhand des vollständigen umgebenden Kontexts vorherzusagen.

Diese Vorgehensweise ähnelt dem menschlichen Prozess des Lückentextlesens. Wie ein Mensch ein fehlendes Wort in einem Satz aus dem Gesamtzusammenhang erschließen kann, lernt das Modell, maskierte Tokens basierend auf ihrem beidseitigen Kontext zu rekonstruieren.

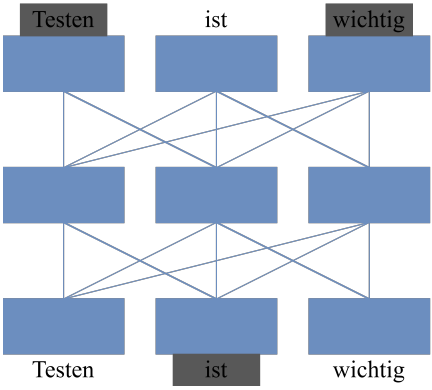



**Abbildung 4.13:** Bidirektionale Verarbeitung in MLMs mit maskierten Tokens (vgl. Thoppilan u. a. 2022; Y. Liu u. a. 2024)

Das bekannteste Beispiel für diesen Ansatz ist BERT (Devlin, Chang und Toutanova 2019), das sich durch seine bidirektionale Verarbeitung besonders für Aufgaben eignet, die ein tiefes Verständnis des Textkontexts erfordern. Dazu gehören die Textklassifikation zur Einordnung von Texten in vorgegebene Kategorien, Named Entity Recognition (NER) zur Erkennung und Klassifizierung benannter Entitäten wie Personen oder Organisationen sowie die Sentiment-Analyse zur Bestimmung der emotionalen Ausrichtung eines Textes.

Die Wahl zwischen CLM und MLM wird durch die spezifischen Anforderungen der Aufgabe bestimmt. Während sich CLMs durch ihre sequentielle Verarbeitung besonders für generative Aufgaben wie Text- oder Code-Erzeugung eignen, zeigen MLMs durch ihre bidirektionale Verarbeitung Stärken bei analytischen Aufgaben, die ein umfassendes Kontextverständnis erfordern.

### 4.4.4 Transformer-Architektur

Die Transformer-Architektur, eingeführt von Vaswani u. a. (2017), markiert einen Paradigmenwechsel in der Verarbeitung sequentieller Daten. Im Folgenden basieren die Erläuterungen der Transformer-Architektur auf Vaswani u. a. (2017). Im Gegensatz zu früheren Ansätzen wie RNNs (Chung u. a. 2014) oder Long Short-Term Memory (LSTM) (Hochreiter und Schmidhuber 1997) basiert die Transformer-Architektur auf Aufmerksamkeitsmechanismen (Attention) und verzichtet komplett auf rekurrente oder konvolutionale Schichten. Diese Aufmerksamkeitsmechanismen ermöglichen die direkte Modellierung von Beziehungen zwischen beliebigen Positionen einer Sequenz, unabhängig von deren Abstand.

Ein wesentlicher Vorteil der Transformer-Architektur liegt in ihrer Fähigkeit zur parallelen Verarbeitung von Eingabesequenzen (Vaswani u. a. 2017). Während RNNs Sequenzen zwangsläufig Element für Element verarbeiten müssen, kann ein Transformer durch seinen Self-Attention-Mechanismus quasi alle Elemente gleichzeitig betrachten. Dies führt zu einer erheblichen Effizienzsteigerung sowohl im Training als auch in der Inferenz. Für die automatische Generierung von Testfällen ist diese Eigenschaft besonders wertvoll, da sie die effiziente Verarbeitung langer Codesequenzen ermöglicht und das Erkennen von Abhängigkeiten zwischen verschiedenen Codeteilen unterstützt.

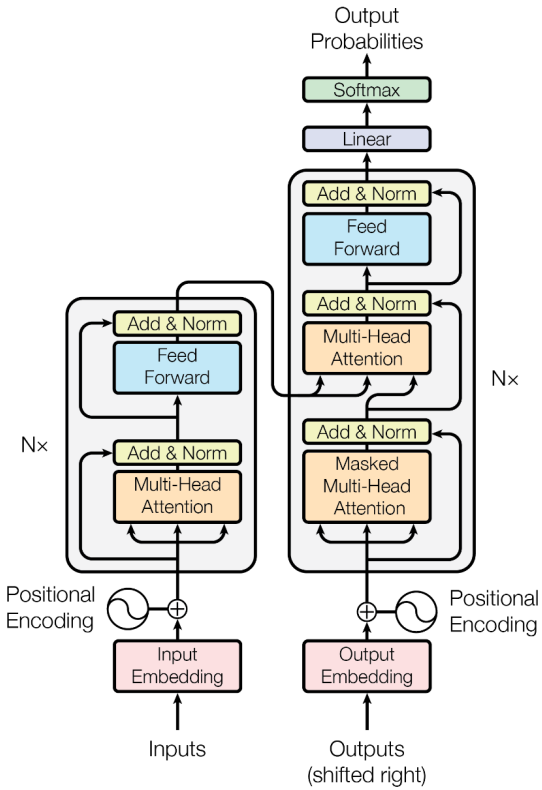



**Abbildung 4.14:** Transformer Architektur (Vaswani u. a. 2017)

Die in Abbildung 4.14 zu sehende Beschreibung der Transformer-Architektur basiert auf (Vaswani u. a. 2017).

#### 4.4.4.1 Encoder-Decoder Struktur

Die Transformer-Architektur verwendet eine Encoder-Decoder-Struktur für die Umwandlung von Eingabesequenzen in Ausgabesequenzen (Vaswani u. a. 2017). Diese Struktur, die bereits bei früheren ML-Ansätzen wie RNNs (siehe Abschnitt 4.2) eingesetzt wurde, übersetzt im Kontext der Testfallgenerierung Quellcode und Spezifikationen in entsprechende Testfälle. Ein typisches Beispiel ist die Umwandlung einer Sequenz von Benutzerinteraktionen in einer mobilen App in ausführbare Testfälle.

Der Encoder verarbeitet die Eingabesequenz durch mehrere identische Schichten. Jede dieser Schichten besteht aus zwei Hauptkomponenten:

- Einem Multi-Head Self-Attention-Mechanismus (siehe Abschnitt 4.4.4.3), der Beziehungen zwischen allen Elementen der Eingabe analysiert.
- Einem Feed-Forward-Netzwerk, ähnlich dem in Abschnitt 4.2.1 beschriebenen, das diese Informationen weiterverarbeitet.

Der Decoder ähnelt in seiner Struktur dem Encoder, enthält aber eine zusätzliche Attention-Schicht, die auf die Ausgabe des Encoders zugreift. Eine wichtige Besonderheit des Decoders ist die Maskierung: Sie stellt sicher, dass bei der Generierung eines Elements nur die bereits generierten Elemente berücksichtigt werden. Diese Eigenschaft ist essentiell für die autoregressive Generierung von Sequenzen, wie sie in Abschnitt 4.4.3.2 für CLMs beschrieben wurde.

#### 4.4.4.2 Self-Attention Mechanismus

Der Self-Attention-Mechanismus bildet das Kernstück der Transformer-Architektur und ermöglicht das Verständnis von Zusammenhängen in Sequenzen. Anders als bei RNNs (siehe Abschnitt 4.2), die Informationen sequentiell verarbeiten, kann Self-Attention direkte Beziehungen zwischen allen Elementen einer Sequenz herstellen.

Der Mechanismus funktioniert, indem er für jedes Element (Token) in der Sequenz drei Vektoren erzeugt:

- Einen „Query"-Vektor, der das aktuelle Token repräsentiert.
- „Key"-Vektoren für alle Token, die mit der Query verglichen werden.
- „Value"-Vektoren, die die eigentlichen Informationen der Token enthalten.

Die Aufmerksamkeit (Attention) wird durch einen als „Scaled Dot-Product Attention" bezeichneten Prozess berechnet:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{4.13}$$

Dabei repräsentieren die Matrizen $Q$, $K$ und $V$ die entsprechenden Vektoren aller Token. Der Skalierungsfaktor $\frac{1}{\sqrt{d_k}}$ wird zur numerischen Stabilisierung verwendet. Diese Berechnung ermöglicht es dem Modell, relevante Teile der Eingabe stärker zu gewichten als weniger relevante.

#### 4.4.4.3 Multi-Head Attention

Durch die parallele Ausführung mehrerer unabhängiger Attention-Operationen erweitert Multi-Head Attention den beschriebenen Self-Attention-Mechanismus (Vaswani u. a. 2017; Brown u. a. 2020).

Jeder dieser „Attention-Heads“ kann sich dabei auf unterschiedliche Aspekte der Eingabesequenz fokussieren. Ähnlich wie verschiedene Filter in CNNs (siehe Abschnitt 4.3.1) unterschiedliche Merkmale erkennen, spezialisiert sich jeder Attention-Head auf bestimmte Beziehungsmuster in den Daten.

Der Mechanismus transformiert die Eingabe zunächst in mehrere parallele Repräsentationen, wobei jeder Head seine eigenen trainierbaren Parameter besitzt. Diese parallelen Verarbeitungspfade ermöglichen es dem Modell, verschiedene Arten von Beziehungen gleichzeitig zu erfassen. Beispielsweise könnte ein Head syntaktische Strukturen fokussieren, während ein anderer semantische Zusammenhänge analysiert.

Die Ausgaben aller Heads werden anschließend zusammengeführt und durch eine lineare Transformation in die finale Ausgabe umgewandelt. Diese Kombination verschiedener Aufmerksamkeitsperspektiven erhöht die Modellkapazität erheblich, ohne die Berechnungseffizienz wesentlich zu beeinträchtigen.

#### 4.4.4.4 Positionsweise Feed-Forward-Schichten

Neben den Attention-Schichten verwendet der Transformer positionsweise Feed-Forward-Network (FFNs) im Encoder und Decoder. Diese Schichten, die den in Abschnitt 4.2 beschriebenen neuronalen Netzen ähneln, verarbeiten jedes Token einzeln und unabhängig voneinander. Sie bestehen aus zwei linearen Transformationen, zwischen denen eine ReLU-Aktivierungsfunktion (siehe Abschnitt 4.2.2) liegt:

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{4.14}$$

Die trainierbaren Parameter $W_1$, $W_2$, $b_1$ und $b_2$ ermöglichen dem Modell, komplexe nicht-lineare Transformationen der Attention-Ausgaben zu lernen.

#### 4.4.4.5 Residual Connections und Layer Normalization

Das Training tiefer neuronaler Netze wird durch zwei wichtige Mechanismen stabilisiert und beschleunigt. Die Residual Connections ermöglichen einen direkten Informationsfluss, indem sie die ursprüngliche Eingabe zu ihrer transformierten Version addieren He u. a. (2016). Diese *Skip-Connections*, wie sie auch genannt werden, ermöglichen es Informationen, tiefere Schichten des Netzes direkt zu erreichen, ohne durch alle Zwischenschichten propagiert werden zu müssen.

Ergänzend dazu stabilisiert die Layer Normalization das Training durch Normalisierung der Aktivierungen. Dieser Prozess standardisiert die Aktivierungen jeder Schicht durch Berechnung des Mittelwerts und der Standardabweichung, was zu einer verbesserten Trainingsgeschwindigkeit und -stabilität führt. Die Normalisierung erfolgt durch trainierbare Parameter, die es dem Modell ermöglichen, eine bessere Skalierung für jede Schicht zu lernen.

Diese Kombination reduziert nicht nur die Abhängigkeit von der Initialisierung der Gewichte, sondern beschleunigt auch den Trainingsprozess erheblich, was gerade für Modelle mit vielen Parametern wichtig ist.

#### 4.4.4.6 Eingabeembeddings und Positionsembeddings

Der Transformer nutzt ein komplexes System von Embeddings zur Verarbeitung von Text und Code (Vaswani u. a. 2017; Devlin, Chang und Toutanova 2019). Ähnlich wie in anderen NLP-Modellen wandeln Eingabeembeddings zunächst die Token in hochdimensionale Vektoren um (siehe Abschnitt 4.4.2), die ihre semantische Bedeutung repräsentieren.

Da der Transformer jedoch keine inhärente Sequenzverarbeitung wie RNNs besitzt, werden zusätzlich Positionsinformationen durch spezielle Positionsembeddings kodiert. Diese basieren auf sinusförmigen Funktionen mit unterschiedlichen Frequenzen für verschiedene Dimensionen des Embeddings (Vaswani u. a. 2017).

Diese mathematische Struktur ermöglicht es dem Modell, sowohl absolute als auch relative Positionen der Token effizient zu erfassen und zu verarbeiten. Die Verwendung trigonometrischer Funktionen gewährleistet dabei, dass das Modell auch bei unterschiedlichen Sequenzlängen stabile Positionsinformationen erhält.

Diese Kombination aus semantischen Eingabeembeddings und geometrisch kodierten Positionsinformationen ermöglicht es dem Transformer, sowohl die Bedeutung als auch die strukturelle Anordnung der Token zu verarbeiten.

### 4.4.5 Quantisierung (Quantization)

Quantisierung ist ein Verfahren zur Reduzierung des Speicher- und Rechenbedarfs von LLMs, indem die numerische Präzision der Modellparameter verringert wird. Die ursprünglich hochpräzisen Modellparameter (typischerweise 32-bit Floating-Point) werden in ein Format mit niedrigerer Präzision, wie 8-bit Integer oder 4-bit, umgewandelt (Dettmers, Pagnoni u. a. 2023). Diese Konvertierung ermöglicht eine erhebliche Reduzierung des Speicherbedarfs und kann gleichzeitig die Verarbeitungsgeschwindigkeit verbessern, da weniger Daten verarbeitet werden müssen (Dettmers, Lewis u. a. 2022).

Ein besonders effektiver Ansatz für die Quantisierung von LLMs ist die Verwendung des 4-bit Normal Float (NF4) Datentyps. Im Gegensatz zu herkömmlichen 4-bit Formaten bietet NF4 eine optimierte Darstellung für normalverteilte Daten, wie sie typischerweise in den Gewichten neuronaler Netze vorkommen (Dettmers, Pagnoni u. a. 2023). Dies ermöglicht es, deutlich größere Modelle für Inferenz und Freinabstimmung (siehe Abschnitt 4.4.6.2) zu nutzen, ohne signifikante Leistungseinbußen zu erleiden (Dettmers, Pagnoni u. a. 2023).

Die praktischen Vorteile dieser Quantisierungstechnik sind vielfältig: Der reduzierte Speicherbedarf erlaubt die Nutzung größerer Modelle auf Hardware mit begrenztem Arbeitsspeicher. Die verringerte Datenmenge führt zu schnelleren Vorhersagen und einem geringeren Energieverbrauch. Dies erweitert den Kreis potentieller Nutzer erheblich, da auch Consumer-Hardware für den Betrieb von LLMs genutzt werden kann.

Um die Modellqualität bei der Quantisierung zu erhalten, sind spezielle Techniken erforderlich. Die Quantized Low-Rank Adaptation (QLoRA)-Methode beispielsweise kombiniert Quantisierung mit Feinabstimmung-Strategien, wodurch hochwertige und gleichzeitig ressourceneffiziente LLMs erzeugt werden können (Dettmers, Pagnoni u. a. 2023). Diese Kombination von Techniken ist besonders wichtig für die lokale Ausführung von LLMs, da sie die Zugänglichkeit und Anwendbarkeit dieser Modelle auf einer breiteren Palette von Hardware ermöglicht. Dadurch können auch Consumer-GPUs, wie in Abschnitt 4.6.3 beschrieben, für das Training und die Inferenz großer Sprachmodelle genutzt werden.

### 4.4.6 Training von LLMs

Das Training von LLMs folgt einem zweistufigen Prozess, bestehend aus einem initialen Vortraining auf umfangreichen Datensätzen und einer anschließenden Feinabstimmung für spezifische Aufgaben. Dieser Ansatz ermöglicht es den Modellen, zunächst ein fundamentales Verständnis für Sprache und Textmuster zu entwickeln, welches später für konkrete Anwendungen spezialisiert werden kann.

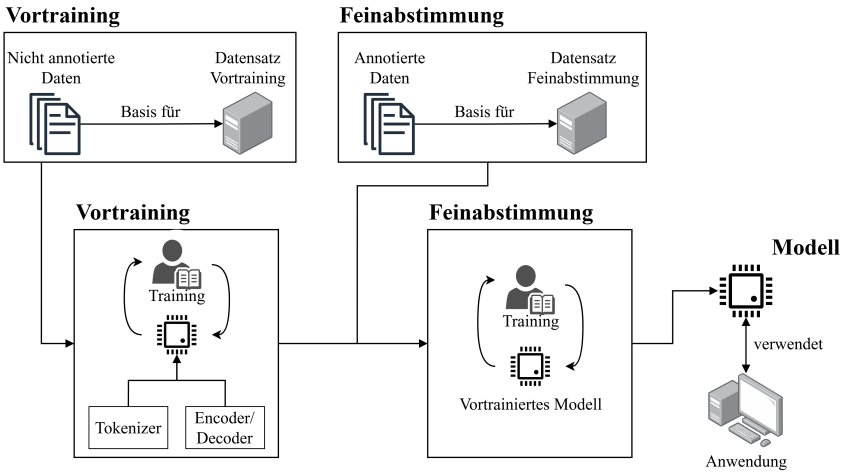



**Abbildung 4.15:** Trainingsprozess eines LLMs

Abbildung 4.15 beschreibt den Trainingsprozess, der sich der in eine Vortraining- („Pretraining“) und eine Feinabstimmungsphase („Finetuning“) aufteilt.

#### 4.4.6.1 Vortraining auf großen Datensätzen

Die Grundlage leistungsfähiger Sprachmodelle bildet das Vortraining auf umfangreichen Datensätzen wie „The Pile“ (Gao u. a. 2020). Dieser Prozess basiert auf den Prinzipien des selbstüberwachten Lernens (siehe Abschnitt 4.1.4) mit einem nicht annotierten Datensatz. Dabei nutzt er die zwei in Abschnitt 4.4.3 vorgestellten Ansätze. Das MLM, bei dem zufällig ausgewählte Tokens maskiert und vom Modell basierend auf dem Kontext vorhergesagt werden (siehe Abschnitt 4.4.3.3), sowie das CLM, das das nächste Token einer Sequenz vorhersagt (siehe Abschnitt 4.4.3.2). Diese Methoden ermöglichen ein effektives Training ohne explizit annotierte Daten, indem die inhärente Struktur der Sprache als implizite Annotation genutzt wird (Fu u. a. 2023).

Das unüberwachte Vortraining lässt sich mathematisch durch die Log-Likelihood-Funktion auf einem nicht annotierten Datensatz von $U$ Tokens beschreiben (Radford, Narasimhan u. a. 2018; Scao u. a. 2022):

$$L_1(U) = \sum_{i=1}^{n} \log P(u_i \mid u_{i-n_{\text{ctx}}}, \ldots, u_{i-1}; \theta) \tag{4.15}$$

Hierbei ist $n_{\text{ctx}}$ die Größe des Kontextfensters (context length), und $P$ stellt die durch ein neuronales Netzwerk mit Parametern $\theta$ modellierte bedingte Wahrscheinlichkeit dar. Das Training erfolgt durch Backpropagation mit Adam-Optimierung oder SGD, wie in Abschnitt 4.2.3 beschrieben.

Die verwendeten Datensätze erreichen enorme Größen – die GPT-Modelle von EleutherAI wurden auf dem 825 GiB großen The Pile-Datensatz trainiert (Scao u. a. 2022), während die BLOOM-Modelle von BigScience den 1,6 TB umfassenden ROOTS Corpus verwendeten (Laurençon u. a. 2022). Modernere Modelle wie LLaMA 3 geben die Größe des Trainingsdatensatzes in Token an und wurden auf bis zu 15 Billionen Token trainiert (Llama Team, AI @ Meta 2024).

#### 4.4.6.2 Feinabstimmung und Anpassung

Die Feinabstimmung, eine spezialisierte Form des Transfer Learnings (siehe Abschnitt 4.3.2), passt die vortrainierten Modelle an spezifische Aufgaben an. Sie stellt eine zentrale Phase im Training großer Sprachmodelle dar und verbessert deren Fähigkeit, spezifische Aufgaben mit hoher Genauigkeit auszuführen. Die überwachte Feinabstimmung auf einem gelabelten Datensatz $LD$ wird mathematisch durch folgende Funktion beschrieben (Pengfei Liu u. a. 2023; Radford, Narasimhan u. a. 2018):

$$L_2(LD) = \sum_{(x,y)} \log P(y \mid x_1, \ldots, x_n; \theta) \tag{4.16}$$

Dabei repräsentiert $x_1, \ldots, x_n$ eine Sequenz von Tokens und $y$ die entsprechende Beschriftung für die Zielaufgabe.

Das Hauptziel der überwachten Feinabstimmung besteht darin, das während des Vortrainings erworbene allgemeine Wissen zu verfeinern und an die spezifischen Anforderungen verschiedener nachgelagerter Aufgaben anzupassen bei gleichzeitig sinkenden Ressourcenanforderungen.

Eine besondere Rolle nehmen die sogenannten Basis-Modelle (Foundation Models) ein, die auf einer breiten, domänenübergreifenden Datenbasis trainiert wurden (Bommasani u. a. 2022). Diese Modelle zeichnen sich durch ihre Flexibilität aus: Sie können durch gezielte Feinabstimmung oder Few-Shot Learning effizient an spezifische Anwendungsbereiche angepasst werden, während sie von ihrer breiten Wissensbasis profitieren (Brown u. a. 2020).

Im Kontext dieser Arbeit könnte $y$ typischerweise einen Unit-Test für eine zu testende Code-Einheit darstellen. Die Verlustfunktionen $L_1(U)$ und $L_2(LD)$ werden während des Trainings und der Feinabstimmung minimiert, um die Modellausgaben schrittweise an die gewünschten Ausgabevariablen anzupassen.

#### 4.4.6.3 Feinabstimmung auf synthetischen Daten

Ein Ansatz zur Verbesserung kleiner Modelle ist die Verwendung synthetischer Daten für die Feinabstimmung (Llama Team, AI @ Meta 2024). Diese Methode, die beispielsweise bei den Phi-Modellen (Abdin u. a. 2024) erfolgreich eingesetzt wurde, kombiniert LLM-basierte Filterung von Webdaten mit LLM-generierten synthetischen Trainingsdaten.

Diese Methode erlaubt es kleineren Sprachmodellen, Leistungen zu erzielen, die typischerweise nur bei deutlich größeren Modellen beobachtet wurden.

#### 4.4.6.4 Optimierungstechniken für das Training

Mit der zunehmenden Größe von Transformer-Modellen werden effiziente Trainings- und Feinabstimmungsmethoden immer wichtiger (Dettmers, Lewis u. a. 2022). Die folgenden Techniken ermöglichen das ressourceneffiziente Training und die Anpassung großer Modelle:

**Parameter-Efficient Fine-Tuning**

Parameter-Efficient Fine-Tuning (Parameter-effizientes Feintuning, PEFT) optimiert den Feinabstimmungsprozess durch die Anpassung einer kleinen Anzahl zusätzlicher Parameter, während der Großteil der vortrainierten Modellparameter unverändert bleibt (Hu u. a. 2021). Dies ermöglicht eine effiziente Anpassung der Modelle bei reduzierten Hardwareanforderungen. Eine wichtige Parameter-Efficient Fine-Tuning (PEFT)-Methode ist Low-Rank Adaptation (LoRA).

**Low-Rank Adaptation**

Low-Rank Adaptation (LoRA) ergänzt trainierbare Rang Zerlegungsmatrizen parallel zu den bestehenden Gewichtsmatrizen eines vortrainierten Modells (Hu u. a. 2021). Das Grundprinzip basiert auf der Beobachtung, dass die Anpassungen während der Feinabstimmung von sich aus in einem niedrigdimensionalen Unterraum liegen, also eine natürlich niedrige Dimensionalität aufweisen.

Für eine ursprüngliche Gewichtsmatrix $W$ der Dimension $d \times k$ (wobei $d$ und $k$ typischerweise mehrere tausend Einträge umfassen) wird eine effiziente Aktualisierung $\Delta W = BA$ eingeführt. Diese Aktualisierung verwendet zwei kleinere Matrizen: Matrix $B$ der Dimension $d \times r$ und Matrix $A$ der Dimension $r \times k$, wobei $r$ den Rang der Zerlegung bezeichnet und deutlich kleiner als $d$ und $k$ gewählt wird (typischerweise 8 oder 16).

Während des Trainings bleiben die ursprünglichen Gewichte $W$ eingefroren, und nur die deutlich kleineren Matrizen $A$ und $B$ werden trainiert. Diese Rang-Zerlegung reduziert die Anzahl der trainierbaren Parameter um den Faktor 10.000 und die GPU-Speicheranforderungen um den Faktor 3 (Hu u. a. 2021), während die Modellqualität weitgehend erhalten bleibt.

**Quantized Low-Rank Adaptation**

Quantized Low-Rank Adaptation (QLoRA) kombiniert die in Abschnitt 4.4.5 beschriebene Quantisierung mit LoRA (Dettmers, Pagnoni u. a. 2023). Während LoRA die Anzahl der zu trainierenden Parameter reduziert, ermöglicht die Quantisierung eine effiziente Speichernutzung durch Komprimierung der Modellparameter. QLoRA führt dafür drei wesentliche Erweiterungen ein:

- 4-bit NormalFloat (NF4): Ein spezieller Datentyp für die Speicherung von Modellgewichten. Klassische 4-bit Formate teilen den Wertebereich in 16 gleichmäßige Intervalle ein. NF4 hingegen passt diese Intervalle an die statistische Verteilung der Gewichte an. In Bereichen, wo viele Gewichte liegen (nahe Null bei einer Normalverteilung), werden die Intervalle feiner gewählt als in den Randbereichen. Dies minimiert den Informationsverlust bei der Komprimierung.
- Doppelte Quantisierung: Auch die Skalierungsfaktoren, die bei der ersten Quantisierung entstehen, werden nochmals quantisiert. Dies ist möglich, da diese Faktoren selbst wieder ähnliche statistische Eigenschaften aufweisen. Diese zweite Quantisierung spart zusätzlichen Speicherplatz, ohne die Modellqualität signifikant zu beeinträchtigen.
- Verbessertes Speichermanagement: Moderne NVIDIA-GPUs erlauben es, Arbeitsspeicher (RAM) wie GPU-Speicher zu nutzen. QLoRA nutzt dies, um Trainingsdaten und Modellparameter dynamisch zwischen RAM und GPU-Speicher zu verschieben. Bei Speicherspitzen während des Trainings werden Daten temporär in den RAM ausgelagert, statt das Training abzubrechen.

Diese Kombination ermöglicht die Feinabstimmung sehr großer Modelle auf schwächerer Hardware. So kann ein 65B-Parameter-Modell auf einer einzelnen 48GB GPU feinabgestimmt werden (Dettmers, Pagnoni u. a. 2023).

### 4.4.7 Verwendete Basismodelle

Die für das Training in Abschnitt 7.3.1.2 verwendeten Basismodelle gehören zur Llama 3- und Gemma-Familie (Llama Team, AI @ Meta 2024; Zhao u. a. 2024; Team u. a. 2024). Llama 3 ist ein Open-Source-Sprachmodell, das in drei Größenvarianten mit 8B, 70B und 405B Parametern verfügbar ist. Das Modell wurde auf einem Korpus von etwa 15 Billionen multilingualen Tokens trainiert und zeigt besondere Stärken in der Code-Generierung und dem mathematischen Schlussfolgern. Die in Abschnitt 4.4.9 beschriebenen Benchmarks belegen dies durch gute Ergebnisse bei HumanEval (89,0%) und MBPP (88,6%) (Llama Team, AI @ Meta 2024).

Als spezialisiertes Modell für die Code-Generierung wurde zusätzlich CodeGemma eingesetzt (Zhao u. a. 2024). Dieses basiert auf der Gemma-Architektur und wurde spezifisch auf über 500 Milliarden Token Code-Daten trainiert. CodeGemma zeichnet sich besonders durch seine Fähigkeiten im Code-Vervollständigen (Fill-in-the-Middle) und der Repository-übergreifenden Kontextverarbeitung aus. Diese Eigenschaften sind für die in Abschnitt 6.2 beschriebene White-Box-Test-Komponente von besonderer Bedeutung.

Die Kombination dieser Modelle ermöglicht eine effektive Implementierung der automatisierten Testgenerierung. Während Llama 3 durch seine große Modellkapazität und breites Verständnis von Code und natürlicher Sprache die Grundlage bildet, ergänzt CodeGemma diese Fähigkeiten durch seine Spezialisierung auf Code-bezogene Aufgaben. Die Modelle behalten dabei ihre Fähigkeiten zur natürlichsprachlichen Verarbeitung bei, was für die Generierung verständlicher und gut dokumentierter Tests essentiell ist, wie in Abschnitt 4.4.6 ausgeführt wird.

Die gewählten Modellarchitekturen unterstützen zudem die in Abschnitt 4.4.5 beschriebenen Quantisierungstechniken, was eine effiziente Implementierung auch auf Consumer-Hardware ermöglicht.

### 4.4.8 Prompting

Prompting ist eine zentrale Technik zur Nutzung von LLMs. Es bezeichnet die Erstellung und Optimierung von textbasierten Anweisungen oder Anfragen, um das Verhalten des LLMs in eine gewünschte Richtung zu lenken, ohne das Modell selbst zu verändern (Marvin u. a. 2024; Faragó 2023). Im Gegensatz zu den in Abschnitt 4.4.6 beschriebenen Trainingsmethoden werden beim Prompting keine Modellparameter verändert.

Ein Prompt besteht typischerweise aus mehreren Komponenten (Marvin u. a. 2024):

- **Anweisung:** Eine klare Beschreibung der Aufgabe, die das LLM ausführen soll.
- **Kontext:** Hintergrundinformationen, die dem LLM relevantes Wissen zur Verfügung stellen.
- **Eingabedaten:** Spezifische Informationen, auf die sich die Antwort beziehen soll.
- **Ausgabeindikatoren:** Vorgaben zum gewünschten Format oder Typ der Ausgabe.

Eine sorgfältige, auf den Anwendungsfall abgestimmte, Kombination dieser Elemente ist entscheidend für die Effektivität eines Prompts (Marvin u. a. 2024).

Um effektive Prompts zu erstellen, sollten diese präzise und explizit formuliert werden, um Mehrdeutigkeiten zu vermeiden und zielgerichtete Ausgaben zu erhalten (Busch u. a. 2023; Marvin u. a. 2024). Die Verwendung von Satzzeichen und Absätzen zur Strukturierung kann die Qualität der Ausgabe verbessern. Zudem sollten Prompts vorzugsweise auf Englisch formuliert werden, da dies oft zu besseren Ergebnissen führt (Mondal, Bappon und Roy 2024). Die Positionierung wichtiger Anweisungen am Anfang oder Ende des Prompts kann aufgrund des Primacy- und Recency-Effekts die Effektivität weiter steigern.

Verschiedene Prompting-Techniken haben sich als besonders effektiv erwiesen, stellen aber keine abschließende Liste dar (Sahoo u. a. 2024):

### N-Shot Prompting

N-Shot Prompting ist eine Methode bei der dem Prompt n Beispiele hinzugefügt, um dem LLM das gewünschte Ausgabeformat zu verdeutlichen (Touvron, Lavril u. a. 2023; Brown u. a. 2020):

```
Generiere einen Testfall für eine Funktion ``sortiere()'',
↪  die eine Liste von Zahlen aufsteigend sortiert.
Der Testfall soll eine Eingabeliste und die erwartete
↪  Ausgabe enthalten.

Beispiel 1:

    Eingabe: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
    Erwartete Ausgabe: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

Generiere nun einen neuen Testfall:
```

**Prompt 4.1:** N-Shot Prompt Beispiel

Das Modell „lernt“ dann anhand dieser Beispiele, schneller den passenden Kontext zu verwenden und die Aufgabe für neue Instanzen zu lösen. Die Beispiele dienen als Konditionierung für die nachfolgenden Aufgaben, bei denen das Modell eine Antwort generieren soll. Few-Shot Prompting kann die Robustheit der Ausgaben erhöhen (Brown u. a. 2020; Faragó 2023).

Diese Methode ist besonders nützlich, wenn nur begrenzte Trainingsdaten verfügbar sind.

**Zero-Shot Prompting**, als grundlegendste Form des N-Shot Prompting, versucht, das Modell ohne spezifische Beispiele zur Lösung einer Aufgabe zu bringen:

```
Generiere einen Testfall für eine Funktion `sortiere()',
↪  die eine Liste von Zahlen aufsteigend sortiert.

Der Testfall soll eine Eingabeliste und die erwartete
↪  Ausgabe enthalten.
```

**Prompt 4.2:** Zero-Shot Prompt Beispiel

Diese Methode testet die Generalisierungsfähigkeit des Modells am stärksten. Erst Modelle nach GPT-3, die nach Mai 2020 rausgekommen sind, sind überhaupt in der Lage längere Prompting Strategien zu verarbeiten (Reynolds und McDonell 2021; J. Ye u. a. 2023).

### Chain-of-Thought

Chain-of-Thought (CoT) teilt die zu lösende Aufgabe in mehrere Schritte auf, ähnlich dem menschlichen Gedankenprozess beim Lösen komplexer Probleme (Faragó 2023; Wei u. a. 2023):

```
Generiere einen Testfall für eine Funktion `sortiere()`,
↪  die eine Liste von Zahlen aufsteigend sortiert.

Gehe dabei wie folgt vor:


    1. Erstelle eine unsortierte Liste von Zahlen.
    2. Identifiziere besondere Elemente (z.\,B. Duplikate,
    ↪  negative Zahlen).
    3. Sortiere die Liste manuell.
    4. Formuliere den Testfall mit Eingabe und erwarteter
    ↪  Ausgabe.

Denke Schritt für Schritt über die Aufgabe nach und erkläre
↪  dein Vorgehen.
```

**Prompt 4.3:** Chain-of-Thought Prompt Beispiel

Chain-of-Thought (CoT) Prompting kann durch Hinzufügen von Anweisungen wie „Nimm einen tiefen Atemzug und führe dies Schritt für Schritt aus“ implementiert werden (C. Yang u. a. 2024; Wei u. a. 2023).

Der Ansatz ermöglicht es LLMs, Probleme in Teilschritte zu zerlegen und diese schrittweise zu lösen. Dies verbessert sowohl die Robustheit als auch die Genauigkeit der Modellausgaben (C. Yang u. a. 2024; Wei u. a. 2023).

Beispielsweise konnte das PaLM 540B Modell mit CoT Prompting auf dem GSM8K Benchmark für mathematische Textaufgaben eine Genauigkeit von 58% erreichen, verglichen mit 18% bei Standard-Prompting (Wei u. a. 2023). Die Effektivität von CoT Prompting korreliert stark mit der Modellgröße (siehe Abschnitt 4.4.3) - bei kleineren Modellen kann es sogar zu einer Verschlechterung der Leistung führen (Wei u. a. 2023).

### Take a Step Back

Take a Step Back (TaS) stellt eine Erweiterung des CoT Ansatzes dar (Zheng u. a. 2024). Diese Technik fügt einen Abstraktionsschritt vor der eigentlichen Problemlösung ein:

```
Bevor wir einen Testfall für eine Sortierfunktion
↪  erstellen, sollten wir
zunächst einen Schritt zurück machen und das große Ganze
↪  betrachten:

    1. Was sind die wichtigsten Prinzipien beim Testen von
    ↪  Sortierfunktionen?
    2. Welche Arten von Eingaben sind besonders relevant
    ↪  für Sortieralgoritmen?

Nachdem du diese Fragen beantwortet hast, generiere einen
↪  Testfall für die
Funktion 'sortiere()', die eine Liste von Zahlen
↪  aufsteigend sortiert.
```

**Prompt 4.4:** Take a Step Back Prompt Beispiel

Dieser Prozess gliedert sich in zwei Phasen: Zunächst erfolgt die Abstraktion, bei der eine allgemeinere Frage formuliert wird, die auf übergeordnete Konzepte abzielt. Anschließend folgt die Reasoning-Phase, in der das eigentliche Problem basierend auf den identifizierten Konzepten gelöst wird.

Take a Step Back (TaS) Prompting zielt darauf ab, die Leistung von Language Models bei komplexen Aufgaben zu verbessern, indem es ihnen ermöglicht, relevantes Hintergrundwissen zu aktivieren, bevor sie mit der detaillierten Problemlösung beginnen. Ähnlich wie bei CoT Prompting scheint die Effektivität dieser Methode mit der Modellgröße zu korrelieren, wobei größere Modelle tendenziell stärker von dieser Technik profitieren (Zheng u. a. 2024). TaS Prompting ergänzt das Spektrum der Prompting-Techniken um einen Ansatz, der besonders auf die Aktivierung von Hintergrundwissen und die strukturierte Herangehensweise an komplexe Probleme abzielt.

Die folgenden zwei Techniken können mit allen vorher genannten Prompting-Methoden kombiniert werden, um die Effektivität weiter zu steigern.

### Role Prompting

Role Prompting weist dem LLM eine spezifische Rolle zu, in der es die Anweisung ausführen soll (Spasić und Janković 2023; Kong u. a. 2024):

```
Du bist ein erfahrener Software-Tester, spezialisiert auf
↪  Algorithmen.

Deine Aufgabe ist es, einen aussagekräftigen Testfall für
↪  eine Funktion 'sortiere()' zu erstellen, die eine Liste
↪  von Zahlen aufsteigend sortiert.

Nutze dein Fachwissen, um einen Testfall zu generieren, der
↪  mögliche Fehler aufdecken könnte.
```

**Prompt 4.5:** Role Prompt Beispiel

Dies fördert kontextspezifischere und zielgerichtetere Antworten, da das Modell aus der Perspektive eines Experten für die gegebene Aufgabe „denkt“.

### Prompt Templates

Prompt Templates dienen der Strukturierung von Prompts und enthalten Platzhalter für spezifische Daten (Spasić und Janković 2023; Kong u. a. 2024). Sie ermöglichen eine konsistente Erstellung von Prompts und erleichtern die Anpassung an verschiedene Anwendungsfälle. In der Testgenerierung können Templates die zu testende Funktion und ihre Spezifikation enthalten. Sie können auch verwendet werden, um bestimmte Prompt-Muster wie CoT Prompting zu standardisieren, was zur Verbesserung der Robustheit und Vorhersagbarkeit der LLM-Ausgaben beiträgt:

```
Funktion: $FUNKTION_NAME
Beschreibung: $FUNKTIONS_BESCHREIBUNG
Eingabetyp: $EINGABETYP
Ausgabetyp: $AUSGABETYP
Besondere Bedingungen: $BEDINGUNGEN
Generiere einen Testfall für die oben beschriebene
↪  Funktion.
Der Testfall sollte folgende Elemente enthalten:

Eingabedaten
Erwartete Ausgabe
Kurze Erklärung, warum dieser Testfall wichtig ist

Testfall:
```

**Prompt 4.6:** Prompt Template Beispiel

Dieses Template könnte dann wie folgt mit spezifischen Daten gefüllt werden:

```
Funktion: sortiere()
Beschreibung: Sortiert eine Liste von Zahlen in
↪  aufsteigender Reihenfolge
Eingabetyp: Liste von ganzen Zahlen
Ausgabetyp: Sortierte Liste von ganzen Zahlen
Besondere Bedingungen:
Die Funktion soll auch mit negativen Zahlen und Duplikaten
↪  umgehen können
Generiere einen Testfall für die oben beschriebene
↪  Funktion.
Der Testfall sollte folgende Elemente enthalten:

    1. Eingabedaten
    2. Erwartete Ausgabe
    3. Kurze Erklärung, warum dieser Testfall wichtig ist

Testfall:
```

**Prompt 4.7:** Prompt Template Beispiel mit Daten

Die Verwendung solcher Templates ermöglicht es, schnell und konsistent Prompts für verschiedene Funktionen oder Szenarien zu erstellen, wobei die grundlegende Struktur beibehalten wird.

Die Wahl der Prompting-Methode und die sorgfältige Gestaltung der Prompts können einen erheblichen Einfluss auf die Leistung des Modells haben. Faktoren wie die Reihenfolge und Verteilung der Beispiele, das Format der Prompts und die Komplexität der Aufgabe spielen eine wichtige Rolle (Min u. a. 2022).

### 4.4.9 Leistungsbewertung und Metriken für LLMs

Die Leistung von LLMs wird anhand verschiedener Benchmarks und Metriken bewertet. Diese Metriken helfen, die Fähigkeiten eines Modells zur Textgenerierung quantitativ zu bewerten und Vergleiche zwischen verschiedenen Modellen zu ermöglichen. Die Bewertung von LLMs im Kontext der Testgenerierung stellt eine besondere Herausforderung dar, da existierende Metriken nicht speziell für diesen Anwendungsfall entwickelt wurden (Tufano, S. K. Deng u. a. 2022; H. Wang u. a. 2023).

#### 4.4.9.1 Accuracy, PPL und BLEU

*Accuracy* ist eine gängige Evaluierungsmetrik im NLP für Klassifikationsaufgaben wie Textklassifikation oder Stimmungsanalyse (A. Zhang u. a. 2023). Sie ist definiert als das Verhältnis der Anzahl der korrekten Vorhersagen zur Gesamtzahl der getroffenen Vorhersagen:

$$\text{Accuracy} \coloneqq \frac{\text{Number of correct predictions}}{\text{Number of total predictions}} \tag{4.17}$$

Da sowohl die Code- als auch die Testgenerierung in die Kategorie der Textgenerierungsaufgaben fallen, ist die Accuracy-Metrik für die Bewertung dieser Modelle nicht geeignet, da diese eine komplexere Bewertung benötigen. Stattdessen wird die Perplexity (Perplexität) (PPL) zur Bewertung der Modellleistung bei Textgenerierungsaufgaben bevorzugt (Huggingface 2022).

*Perplexity (Perplexität, PPL)* ist eine zentrale Evaluierungsmetrik zur Bewertung der Vorhersagequalität von Sprachmodellen (A. Zhang u. a. 2023). Sie misst die *Unsicherheit* eines Modells bei der Vorhersage des nächsten Wortes (Tokens) in einem Text. Je niedriger die Perplexität, desto sicherer ist das Modell in seinen Vorhersagen und desto besser kann es den Text modellieren.

Als Beispiel dient folgender unvollständiger Satz: „Softwaretests kosten _____“. Ein gutes Sprachmodell soll das fehlende Wort vorhersagen. Es würde dem Wort „*Geld*“ eine hohe Wahrscheinlichkeit zuweisen, während es „*Eis*“ als sehr unwahrscheinlich einstufen würde. Die Perplexität quantifiziert diese Vorhersagefähigkeit über alle möglichen Wörter und den gesamten Text hinweg.

Mathematisch wird die PPL als exponentieller Term der mittleren negativen Log-Likelihood berechnet:

$$\mathrm{PPL}(W) = \exp\left(-\frac{1}{n}\sum_{i=1}^{n}\log P(w_i \mid w_{i-1}, \ldots, w_1)\right) \tag{4.18}$$

Dabei ist $n$ die Anzahl der Wörter im Text und $P(w_i \mid w_{i-1}, \ldots, w_1)$ die Wahrscheinlichkeit, die das Modell dem tatsächlich auftretenden Wort $w_i$ zuweist, nachdem es die vorherigen Wörter $w_{i-1}, \ldots, w_1$ gesehen hat.

Die Exponentialfunktion (exp) wird verwendet, um die Log-Wahrscheinlichkeiten wieder in den ursprünglichen Wahrscheinlichkeitsraum zu transformieren. Ein niedrigerer PPL-Wert bedeutet dabei eine bessere Vorhersagefähigkeit des Modells.

Die PPL wird auf einem separaten Validierungsdatensatz berechnet, der während des Trainings nicht verwendet wurde. Dies stellt sicher, dass die Metrik die echte Generalisierungsfähigkeit des Modells misst und nicht nur seine Fähigkeit, die Trainingsdaten auswendig zu lernen.

1. Ein PPL-Wert von 1 bedeutet perfekte Vorhersagen. Das Modell war sich bei jedem Wort zu 100% sicher.

2. Ein PPL-Wert von 2 bedeutet, dass das Modell im Durchschnitt zwischen zwei gleichwahrscheinlichen Alternativen entscheiden musste.

3. Höhere PPL-Werte zeigen an, dass das Modell bei seinen Vorhersagen zunehmend unsicher war. Die PPL wächst exponentiell mit der Unsicherheit des Modells.

In dieser Arbeit dient die PPL als zentrale quantitative Metrik zur Bewertung der Modellqualität nach der Feinabstimmung. Sie ermöglicht einen objektiven Vergleich verschiedener Modellversionen und Trainingsstrategien.

Der *BLEU-Score* (Bilingual Evaluation Understudy), ursprünglich für die Evaluierung maschineller Übersetzungen entwickelt (Papineni u. a. 2002), vergleicht n-Gramme in der vorhergesagten Sequenz mit n-Grammen in der Zielsequenz und berechnet die Präzision dieser Übereinstimmungen. Während ein höherer BLEU-Score eine größere Ähnlichkeit zwischen generierter und Zielsequenz anzeigt, hat diese Metrik Schwierigkeiten, semantische Merkmale von Code zu erfassen (M. Chen u. a. 2021). Dies bedeutet, dass der BLEU-Score möglicherweise nicht in der Lage ist, die Bedeutung und Funktionalität des generierten Codes im Vergleich zur Referenzlösung angemessen zu bewerten. Selbst wenn der generierte Code eine hohe Ähnlichkeit in Bezug auf die n-Gramme aufweist, kann er semantisch oder funktional von der Referenzlösung abweichen.

#### 4.4.9.2 CodeBLEU

CodeBLEU (Ren u. a. 2020) wurde als spezifische Metrik für die Bewertung von generiertem Code entwickelt. Im Gegensatz zum klassischen BLEU-Score berücksichtigt CodeBLEU die besonderen Eigenschaften von Programmcode. Die Metrik kombiniert vier Komponenten:

1. Die lexikalische Ähnlichkeit (n-Gramm-basiert wie beim BLEU-Score)
2. Die syntaktische Ähnlichkeit basierend auf abstrakten Syntaxbäumen
3. Die semantische Ähnlichkeit durch Analyse der Datenflusspfade

4. Die Ähnlichkeit der Schlüsselwort-Verwendung

Diese mehrdimensionale Bewertung ermöglicht eine differenziertere Beurteilung der Codequalität als traditionelle Metriken. Experimente haben gezeigt, dass CodeBLEU besser mit menschlichen Bewertungen übereinstimmt als bisherige Ansätze (Ren u. a. 2020).

#### 4.4.9.3 HumanEval-Datensatz und Pass@k-Metrik

Der HumanEval-Datensatz wurde entwickelt, um die Problemlösungsfähigkeiten von LLMs systematisch zu evaluieren. Er besteht aus 164 von Experten erstellten Programmieraufgaben, wobei jede Aufgabe eine Funktionssignatur, einen Docstring, einen Funktionskörper und durchschnittlich 7,7 Unit-Tests enthält (M. Chen u. a. 2021).

Die bewusst manuelle Erstellung dieser Aufgaben durch Experten stellt sicher, dass die Aufgaben nicht aus existierenden Code-Repositories stammen und somit eine echte Evaluierung der Modellleistung ermöglichen. Dies ist wichtig, da viele LLMs auf umfangreichen Code-Datenbanken wie GitHub trainiert werden. Der Pile-Datensatz beispielsweise enthält zu etwa 7,5% GitHub-Daten (Gao u. a. 2020).

Die Aufgaben im HumanEval-Datensatz sind so konzipiert, dass sie Fähigkeiten wie Sprachverständnis, logisches Denken, Algorithmen und Grundkenntnisse in Mathematik evaluieren. Der öffentlich verfügbare Datensatz ermöglicht einen standardisierten Vergleich verschiedener Modelle.

Zur Bewertung der funktionalen Korrektheit der generierten Lösungen wird die Pass@k-Metrik verwendet (Kulal u. a. 2019). Diese Metrik basiert auf einem statistischen Ansatz. Für jede Aufgabe werden $n$ verschiedene Lösungsvorschläge generiert (wobei $n \geq k$). Ein Problem gilt als gelöst, wenn mindestens einer dieser $n$ Vorschläge alle zugehörigen Unit-Tests besteht.

Die Pass@k-Metrik berechnet sich für jede Aufgabe nach folgender Formel:

$$\text{Pass@k} = \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{4.19}$$

Dabei bezeichnet $n$ die Gesamtzahl der generierten Lösungen, $k$ die Anzahl der zufällig ausgewählten Lösungen und $c$ die Anzahl der korrekten Lösungen unter den $n$ Versuchen. Der Erwartungswert $\mathbb{E}$ wird verwendet, da die Auswahl der $k$ Lösungen aus den $n$ Versuchen zufällig erfolgt. Die Binomialkoeffizienten $\binom{n-c}{k}$ und $\binom{n}{k}$ beschreiben dabei die möglichen Kombinationen der Lösungsauswahl.

Der Gesamtanteil der gelösten Probleme wird als Maß für die Leistung des Modells verwendet (M. Chen u. a. 2021).
Untersuchungen mit dem auf Code spezialisierten GPT-Modell Codex zeigen einen systematischen Zusammenhang zwischen Modellgröße und Leistung. Größere Modelle erreichen konsistent bessere Pass@k-Werte, wobei die Verbesserung einer logarithmischen Funktion der Modellgröße folgt (M. Chen u. a. 2021).

Die Sampling-Temperatur stellt einen wichtigen Parameter bei der Anwendung der Pass@k-Metrik und allgemein der Verwendung von LLMs. Die Temperatur steuert die Zufälligkeit bei der Tokengenerierung durch das Modell. Ein Wert von $1,0$ führt zu kreativen, aber möglicherweise weniger präzisen Ausgaben, während ein Wert von 0 stets die wahrscheinlichste Ausgabe wählt. Höhere Temperaturwerte führen zu vielfältigeren Lösungsvorschlägen, was die Wahrscheinlichkeit erhöht, mindestens eine korrekte Lösung zu generieren.

## 4.5 Reinforcement Learning

Reinforcement Learning (Bestärkendes Lernen, RL) ist ein Teilgebiet des maschinellen Lernens, das sich fundamental von den in Abschnitt 4.1.1 und 4.1.2 beschriebenen Ansätzen unterscheidet (Barto, R. Sutton und C. Watkins 1989; R. S. Sutton und Barto 2018). Während überwachtes Lernen mit vordefinierten Eingabe-Ausgabe-Paaren und unüberwachtes Lernen mit der Entdeckung von Datenstrukturen arbeitet, lernt ein RL-Agent durch direkte Interaktion mit seiner Umgebung. Der Agent verbessert seine Entscheidungsstrategie basierend auf einem Belohnungssignal, das die Qualität seiner Aktionen bewertet (Barto, R. Sutton und C. Watkins 1989).

**Interaktionszyklus**

Die Interaktion zwischen Agent und Umgebung erfolgt in diskreten Zeitschritten, wie in Abbildung 4.16 dargestellt. In jedem Zeitschritt nimmt der Agent zunächst den aktuellen Zustand $S_t$ der Umgebung wahr. Basierend auf dieser Wahrnehmung wählt er eine Aktion $A_t$ gemäß seiner aktuellen Strategie (Policy) $\pi$. Nach Ausführung der Aktion beobachtet der Agent eine Belohnung $R_{t+1}$ sowie einen neuen Zustand $S_{t+1}$. Diese sequentielle Entscheidungsfindung ermöglicht es dem Agenten, aus den Konsequenzen seiner Aktionen zu lernen und seine Strategie kontinuierlich zu verbessern.

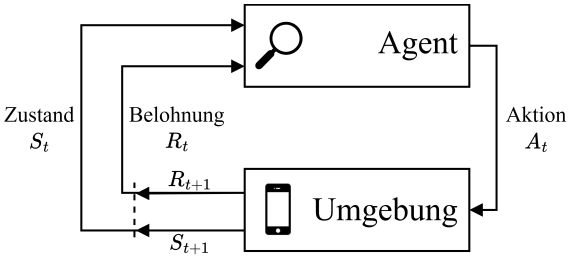



**Abbildung 4.16:** Interaktionszyklus im RL

Der Interaktionszyklus spiegelt dabei die fundamentale Struktur des Lernprozesses wider, bei dem der Agent durch wiederholtes Ausprobieren und Beobachten der Resultate sein Verhalten anpasst.

**Markov-Entscheidungsprozesse**

Die mathematische Grundlage des RL bilden Markov Decision Processes (Markow-Entscheidungsprozesse, MDPs) (Bellman 1957; Bertsekas 2012). Ein MDP formalisiert die Entscheidungsfindung in Situationen, in denen die Ergebnisse teilweise zufällig und teilweise unter der Kontrolle eines Entscheidungsträgers stehen.

Die Struktur eines Markov Decision Processes (MDP) wird durch fünf Kernelemente definiert:

- Der **Zustandsraum** $\mathcal{S}$ umfasst alle möglichen Konfigurationen der Umgebung. Im Kontext des GUI-Testings repräsentiert dies beispielsweise alle möglichen Bildschirmzustände einer Applikation.
- Der **Aktionsraum** $\mathcal{A}$ beschreibt die Menge aller dem Agenten im Zustand $s$ zur Verfügung stehenden Aktionen. Dies entspricht bei mobilen Apps den verschiedenen Interaktionsmöglichkeiten wie Tippen, Wischen oder Scrollen.
- Die **Übergangsfunktion** $P(s'|s, a)$ gibt die Wahrscheinlichkeit an, mit der die Ausführung einer Aktion $a$ in Zustand $s$ zu einem neuen Zustand $s'$ führt.
- Die **Belohnungsfunktion** $R(s, a, s')$ bewertet die unmittelbare Belohnung für den Übergang von Zustand $s$ zu Zustand $s'$ durch Ausführung der Aktion $a$. Bei der Beschreibung von RL-Algorithmen wird die zum Zeitpunkt $t$ von der Umgebung gegebene Belohnung als $R_t$ notiert (R. S. Sutton und Barto 2018).
- Der **Diskontierungsfaktor** $\gamma \in [0, 1]$ gewichtet die Bedeutung zukünftiger Belohnungen im Verhältnis zu unmittelbaren Belohnungen.

### Markov-Eigenschaft

Eine zentrale Eigenschaft von MDPs ist die Markov-Eigenschaft, die das Fundament für die mathematische Behandlung von RL-Problemen bildet (R. S. Sutton und Barto 2018).
Die Markov-Eigenschaft besagt, dass der nächste Zustand $S_{t+1}$ und die Belohnung $R_{t+1}$ ausschließlich vom aktuellen Zustand $S_t$ und der gewählten Aktion $A_t$ abhängen, nicht aber von der Historie vorheriger Zustände und Aktionen.
Im Kontext des GUI-Testings bedeutet dies, dass die Reaktion einer mobilen App auf eine Benutzereingabe nur vom aktuellen Bildschirmzustand abhängt, nicht aber davon, wie dieser Zustand erreicht wurde.

Diese Eigenschaft ist von fundamentaler Bedeutung für die praktische Anwendbarkeit von RL, da sie die Komplexität des Lernproblems erheblich reduziert. Statt die gesamte Historie von Zuständen und Aktionen zu berücksichtigen, kann der Agent seine Entscheidungen ausschließlich auf Basis des aktuellen Zustands treffen.

### Zielfunktion und Belohnung

Das übergeordnete Ziel eines RL-Agenten besteht in der Maximierung der erwarteten kumulierten diskontierten Belohnung (Return) (R. S. Sutton und Barto 2018).
Im Kontext des GUI-Testings könnte eine solche Belohnung beispielsweise für das Erreichen neuer Bildschirmzustände oder das Aufdecken von Fehlern vergeben werden.

Der Gesamt-Belohnung $G_t$ zum Zeitpunkt $t$ wird definiert als:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{4.20}$$

wobei $R_{t+k+1}$ die unmittelbare Belohnung darstellt, die der Agent nach dem Ausführen einer Aktion zum Zeitpunkt $t + k$ und dem Übergang in den nächsten Zustand erhält.

Der Diskontierungsfaktor $\gamma \in [0, 1]$ bestimmt dabei die relative Gewichtung zukünftiger Belohnungen gegenüber unmittelbaren Belohnungen. Ein kleinerer Wert für $\gamma$ führt zu einer stärkeren Gewichtung unmittelbarer Belohnungen, wie dem sofortigen Auffinden eines Fehlers, während Werte nahe 1 langfristige Ziele, wie eine umfassende Testüberdeckung, stärker berücksichtigen. Ein $\gamma$ von 0 bedeutet, dass der Agent ausschließlich die unmittelbare Belohnung berücksichtigt und zukünftige Belohnungen vollständig ignoriert. Ein Wert von $\gamma$ nahe 1 berücksichtigt langfristige Ziele stärker, kann aber in Umgebungen mit unendlich langen Episoden zu unendlichen Belohnungssummen führen.

**Wertfunktionen**

Zur Bewertung von Zuständen und Aktionen dienen zwei Funktionen (R. S. Sutton und Barto 2018):

Die *Value Function* $V^{\pi}(s)$ bewertet einen Zustand $s$ unter einer Strategie $\pi$ als die erwartete zukünftige Belohnung. Im Kontext des Testens entspricht dies der Einschätzung, wie vielversprechend ein bestimmter Zustand der Anwendung für das Auffinden von Fehlern ist.

Die *Q-Function* $Q^{\pi}(s, a)$ erweitert dieses Konzept, indem sie zusätzlich die Ausführung einer spezifischen Aktion $a$ berücksichtigt. Dies ermöglicht dem Agenten, die Auswirkungen verschiedener Testaktionen wie Klicks oder Swipe-Gesten zu bewerten und die vielversprechendste auszuwählen.

### Bellman-Gleichungen

Die Bellman-Gleichungen (Bellman und Kalaba 1957) beschreiben die mathematische Beziehung zwischen dem Wert eines Zustands und den Werten seiner möglichen Nachfolgezustände. Sie formalisieren das Prinzip, dass der Wert eines Zustands von der unmittelbaren Belohnung und den Werten der erreichbaren Folgezustände abhängt.

Für das GUI-Testing bedeutet dies, dass der Wert eines Bildschirmzustands sowohl von seiner unmittelbaren Relevanz für das Testziel als auch von den daraus erreichbaren weiteren Zuständen bestimmt wird. Die optimale Teststrategie $\pi^*$ maximiert dabei die erwartete Gesamtbelohnung über alle möglichen Zustände der Anwendung.

### Exploration vs. Exploitation

Eine zentrale Herausforderung im RL ist die Balance zwischen *Exploration* und *Exploitation* (Kaelbling, Littman und Moore 1996).

Der Agent muss kontinuierlich zwischen zwei Verhaltensweisen abwägen: der Erkundung neuer, möglicherweise besserer Aktionen (Exploration) und der Nutzung bereits bekannter, erfolgversprechender Aktionen (Exploitation).

Eine zu starke Fokussierung auf Exploitation kann dazu führen, dass der Agent in suboptimalen Strategien verharrt, während übermäßige Exploration die effektive Nutzung bereits erlernten Wissens verhindert.

Diese Abwägung ist besonders relevant für das Testen mobiler Apps, da der Agent einerseits bekannte Funktionspfade gründlich testen, andererseits aber auch bisher unentdeckte Bereiche der Anwendung erkunden soll. Verschiedene Strategien wie $\epsilon$-greedy oder Softmax-Exploration bieten Möglichkeiten, diese Balance zu steuern und an die spezifischen Anforderungen der Testaufgabe anzupassen.

### 4.5.1 Algorithmenklassen im RL

Reinforcement Learning Algorithmen lassen sich in verschiedene Klassen einteilen, die sich in ihrer Herangehensweise an das Problem der optimalen Handlungsstrategiefindung unterscheiden (R. S. Sutton und Barto 2018). Diese Klassen sind nicht strikt getrennt, und es gibt Überschneidungen und Kombinationen. Die Wahl des passenden Algorithmus hängt von den spezifischen Eigenschaften des Problems und den verfügbaren Ressourcen ab (R. S. Sutton und Barto 2018).

#### 4.5.1.1 Value-basierte Algorithmen

Value-basierte Algorithmen bilden den traditionellen Ansatz des RL und konzentrieren sich auf das Erlernen der Value Function $V(s)$ oder der Q-Function $Q(s, a)$ (C. J. C. H. Watkins und Dayan 1992). Diese Funktionen ordnen jedem Zustand bzw. Zustand-Aktions-Paar einen erwarteten zukünftigen Wert zu.
Im Kontext des GUI-Testings bedeutet dies, dass der Algorithmus lernt einzuschätzen, welche Bildschirmzustände und Interaktionen am vielversprechendsten für das Auffinden von Fehlern oder das Erreichen einer hohen Testüberdeckung sind.

Ein grundlegender Algorithmus dieser Klasse ist *Q-Learning* (C. J. C. H. Watkins und Dayan 1992), der eine Q-Funktion iterativ verbessert. Im GUI-Testing-Kontext lernt Q-Learning beispielsweise, dass das Drücken eines „Speichern“-Buttons in einem nicht ausgefüllten Formular wahrscheinlich zu einem Fehlerzustand führt und weist dieser Aktion entsprechend einen hohen Wert zu. Die optimale Teststrategie wählt dann bevorzugt Aktionen mit hohen Q-Werten aus (Curtis 2003).

Zur Gewährleistung einer umfassenden Testüberdeckung wird Q-Learning mit Explorationsstrategien wie $\varepsilon$-greedy kombiniert (R. S. Sutton und Barto 2018). Eine reine Greedy-Strategie würde stets die Aktion mit dem höchsten erwarteten Wert wählen, während die $\varepsilon$-greedy Strategie mit einer Wahrscheinlichkeit von $\varepsilon$ eine zufällige Aktion auswählt und dadurch sicherstellt, dass neben den bereits als effektiv erkannten Testpfaden auch neue, bisher unerforschte Bereiche der Anwendung getestet werden.

#### 4.5.1.2 Policy-basierte Algorithmen

Policy-basierte Algorithmen verfolgen einen direkteren Ansatz, indem sie die Teststrategie $\pi(a|s)$ unmittelbar optimieren, ohne den Umweg über Wertfunktionen (R. S. Sutton und Barto 2018).
Diese Algorithmen eignen sich besonders für das GUI-Testing mobiler Applikationen, da sie effektiv mit kontinuierlichen Aktionsräumen umgehen können, wie sie bei Touch-Gesten mit variablen Koordinaten und Geschwindigkeiten auftreten.

Ein wichtiger Vertreter dieser Klasse ist Proximal Policy Optimization (PPO) (Schulman u. a. 2017), der sich durch besondere Stabilität im Lernprozess auszeichnet. Bei der Anwendung im GUI-Testing verhindert PPO durch seinen Clipping-Mechanismus, dass sich die Teststrategie zu abrupt ändert. Der Clipping-Mechanismus funktioniert dabei wie ein Sicherheitsnetz. Er begrenzt die maximale Änderung der Strategie in jedem Lernschritt auf einen vordefinierten Prozentsatz. Dies ist besonders wichtig, da zu große Änderungen in der Teststrategie dazu führen könnten, dass bereits erfolgreich getestete Funktionalitäten nicht mehr erreicht werden.

Bei der Implementierung von RL-Algorithmen wird zwischen diskreten und stetigen Aktionsräumen unterschieden. Während diskrete Aktionsräume eine endliche Menge möglicher Aktionen umfassen (wie beispielsweise „Klick“ oder „Swipe“), beschreiben stetige Aktionsräume kontinuierliche Wertebereiche (wie die exakten x-y-Koordinaten einer Touch-Geste).
Proximal Policy Optimization (PPO) eignet sich besonders für stetige Aktionsräume, da der Algorithmus durch seinen Clipping-Mechanismus auch bei kontinuierlichen Werten stabile Gradientenaktualisierungen gewährleistet.

Im Vergleich zu anderen Algorithmen wie beispielsweise Soft Actor-Critic (SAC) zeichnet sich PPO durch einen geringeren Speicherbedarf aus, da er ohne separate Target-Netzwerke und große Replay-Buffer auskommt.

#### 4.5.1.3 Hybride Actor-Critic Algorithmen

Actor-Critic-Algorithmen (Konda und Tsitsiklis 1999) verbinden die Vorteile value-basierter und policy-basierter Ansätze. Sie verwenden zwei neuronale Netze:

- einen Actor, der die Teststrategie bestimmt,
- und einen Critic, der diese Strategie bewertet und Verbesserungsvorschläge macht.

Im Kontext des GUI-Testings könnte das Actor-Netzwerk beispielsweise Touch-Gesten auswählen, während das Critic-Netzwerk bewertet, wie effektiv diese Gesten zur Testüberdeckung oder zum Auffinden von Fehlern beitragen.

Deep Deterministic Policy Gradient (DDPG) (Gu u. a. 2016) ist ein Actor-Critic-Algorithmus für kontinuierliche Aktionsräume, der Off-Policy-Lernen ermöglicht. Off-Policy bedeutet, dass der Algorithmus aus aufgezeichneten Erfahrungen lernen kann, die also mit einer anderen Teststrategie erstellt wurden. Das können auch aufgezeichnete, von Menschen durchgeführte Interaktionen sein. Dies erhöht die Effizienz des Lernprozesses, da viele gesammelte Testdaten genutzt werden können. Darunter auch höherwertige, von Menschen generierte Beispiele.

Der Twin Delayed Deep Deterministic Policy Gradient (TD3)-Algorithmus (Fujimoto, Hoof und Meger 2018) verbessert den grundlegenden Deep Deterministic Policy Gradient (DDPG) durch drei wesentliche Erweiterungen. Ein häufiges Problem bei DDPG ist die Überschätzung der Q-Werte, was zu einer fehlerhaften Strategieentwicklung führt.

- Zwei Q-Funktionen (*Twin*) bewerten parallel die Aktionen. Für die Aktualisierung wird der kleinere der beiden Q-Werte verwendet. Dies verhindert eine systematische Überschätzung der Aktionswerte (Überoptimismus). Eine Überschätzung entsteht, wenn der Agent aufgrund von Approximationsfehlern in der Q-Funktion den Wert bestimmter Aktionen zu hoch einschätzt und diese dann bevorzugt auswählt.
- Die verzögerte Aktualisierung (*Delayed*) der Strategiefunktion reduziert die Volatilität im Lernprozess. Die Q-Funktionen werden doppelt so häufig aktualisiert wie die Strategiefunktion. Dies stabilisiert den Lernprozess, da die Q-Funktionen Zeit haben, sich an die aktuelle Strategie anzupassen, bevor diese wieder verändert wird.
- Die Glättung der Zielstrategie (*Target Policy Smoothing*) fügt den vorhergesagten Aktionen Rauschen hinzu. Dies erschwert es der Strategiefunktion, kleine Fehler in den Q-Funktionen auszunutzen, und führt zu robusteren Strategien.

Soft Actor-Critic (SAC) (Haarnoja u. a. 2018) erweitert diesen Ansatz durch die Integration von Entropie-Regularisierung. Der Algorithmus maximiert nicht nur die erwartete Belohnung, sondern auch die Entropie der Strategiefunktion. Die Entropie ist ein Maß für die Zufälligkeit der gewählten Aktionen. Ein Regularisierungsparameter $\alpha$ steuert dabei den Kompromiss zwischen der Maximierung der Belohnung (Exploitation) und der Erhöhung der Entropie (Exploration).

- SAC verwendet, ähnlich wie Twin Delayed Deep Deterministic Policy Gradient (TD3), zwei Q-Funktionen zur Vermeidung von Überoptimismus.
- Anders als bei TD3 wird eine stochastische Strategiefunktion gelernt. Diese gibt für jeden Zustand eine Wahrscheinlichkeitsverteilung über mögliche Aktionen zurück.
- Die stochastische Natur der Strategiefunktion macht eine explizite Exploration durch zusätzliches Rauschen überflüssig, da die Zufälligkeit der Aktionsauswahl bereits für ausreichende Exploration sorgt.
- Während des Testens wird statt einer zufälligen Aktion der Mittelwert der gelernten Verteilung verwendet, um die beste bekannte Aktion auszuführen.

Beide Algorithmen haben sich als besonders effektiv für das Training von Agenten in kontinuierlichen Aktionsräumen erwiesen, wie sie beim GUI-Testing auftreten (Haarnoja u. a. 2018). Die Kombination aus verbesserter Stabilität und effektiver Exploration macht sie zu vielversprechenden Kandidaten für die Automatisierung von Testprozessen (Haarnoja u. a. 2018).

#### 4.5.1.4 Model-basierte Algorithmen

Model-basierte Algorithmen erweitern den RL-Ansatz durch ein explizites Modell der Umgebungsdynamik (R. S. Sutton und Barto 2018).

Dieses Modell ermöglicht es dem Agenten, die Konsequenzen seiner Aktionen vorherzusagen und damit effizientere Lernstrategien zu entwickeln.

Im Kontext des GUI-Testings kann ein solches Modell beispielsweise das erwartete Verhalten der Anwendung auf bestimmte Eingabevariablen vorhersagen und damit die Testüberdeckung verbessern.

Die Wahl zwischen den verschiedenen Algorithmenklassen hängt von mehreren Faktoren ab:

- Der Struktur des Zustandsraums der zu testenden Anwendung.
- Der Art der verfügbaren Interaktionen (diskret vs. kontinuierlich).
- Den spezifischen Anforderungen an die Testüberdeckung.
- Den verfügbaren Rechenressourcen.

In der praktischen Implementierung, wie sie in Abschnitt 4.6.1 mit AndroidEnv realisiert wird, können verschiedene Algorithmen mithilfe von Stable Baseline[32] für unterschiedliche Aspekte des Testprozesses kombiniert werden.

[32] Stable Baseline ist eine Python-Bibliothek, die eine Vielzahl von RL-Algorithmen implementiert und eine einfache und konsistente API für die Entwicklung und Evaluierung von RL-Agenten bereitstellt (Raffin u. a. 2021).

### 4.5.2 Hyperparameter-Anpassung

Die Wahl der Hyperparameter ist ein entscheidender Faktor im Training von RL-Algorithmen (R. S. Sutton und Barto 2018). Hyperparameter sind Einstellungen, die die Funktionsweise des Algorithmus steuern, wie z.B. die Lernrate, die Größe des Aktionsraums oder die Anzahl der Schichten in einem neuronalen Netzwerk. Eine systematische Anpassung dieser Parameter gewährleistet eine effiziente Konvergenz des Lernalgorithmus. Dabei können verschiedene Ansätze wie Grid-Search, Random-Search oder Bayesian Optimization verwendet werden (Goodfellow, Bengio und Courville 2016).

## 4.6 Werkzeuge und Frameworks

Die folgenden Abschnitte stellen die Werkzeuge und Frameworks vor, die in dieser Arbeit für die Implementierung und Evaluierung der vorgestellten Methoden verwendet werden.

### 4.6.1 AndroidEnv

AndroidEnv ist eine offene Python-Bibliothek, die ein Android-Gerät als RL Umgebung bereitstellt (Toyama u. a. 2021). Diese Plattform ermöglicht die Definition individueller Interaktionsmöglichkeiten mit dem Android-Betriebssystem und bietet somit eine flexible Grundlage für die RL-Forschung im Kontext realer Anwendungen.

Die Architektur von AndroidEnv basiert auf der Implementierung der dm_env-API (Muldal u. a. 2019) über einem emulierten Android-Gerät, wobei die Umgebung in Echtzeit läuft und die Simulation unabhängig von den Agenteneingaben fortschreitet. Ein zentrales Merkmal von AndroidEnv ist sein komplexer Aktionsraum. Der native Aktionsraum besteht aus einem hybriden Tupel, das folgende Komponenten umfasst:

- Eine kontinuierliche Position $(x, y) \in [0, 1] \times [0, 1]$ auf dem Bildschirm
- Einen diskreten Aktionstyp *TOUCH*, *LIFT*, *REPEAT*

Diese Struktur ermöglicht die Ausführung unterschiedlicher Gesten wie Tippen, Wischen oder Ziehen und Ablegen, die für die Interaktion mit Android-Apps erforderlich sind. Die Komplexität ergibt sich aus der Notwendigkeit, Sequenzen von Rohaktionen zu sinnvollen Gesten zu kombinieren, um Zustandsänderungen in der Umgebung auszulösen.

Der Beobachtungsraum von AndroidEnv ist ebenfalls umfassend gestaltet. Er umfasst drei Hauptkomponenten (Toyama u. a. 2021):

- pixels: Ein RGB-Bildarray, das den aktuellen Bildschirminhalt repräsentiert.
- timedelta: Die seit der letzten Beobachtung vergangene Zeit.
- orientation: Die Orientierung des Geräts.

Zusätzlich können aufgabenspezifische *„Extras“* bereitgestellt werden, die strukturierte Informationen zur Unterstützung des Lernprozesses enthalten. Diese Extras können schwer aus Rohpixeln extrahierbare Informationen wie Zustände oder Ereignisse beinhalten.

AndroidEnv bietet einen flexiblen Mechanismus zur Definition spezifischer RL-Probleme. Dies umfasst die Festlegung von:

- Episodenabbruchbedingungen
- Belohnungsfunktionen
- Interaktionsmöglichkeiten mit spezifischen Apps

Die Plattform ermöglicht es, eine Vielzahl von Aufgaben zu definieren, die von einfachen GUI-Navigationsaufgaben bis hin zu komplexen Spielen, mit vielen möglichen Interaktionen, reichen können. Zur Anpassung des Beobachtungs- und Aktionsraums stellt AndroidEnv verschiedene Wrapper zur Verfügung. Diese ermöglichen beispielsweise die Diskretisierung des Aktionsraums, die Skalierung der Bildbeobachtungen oder die Anpassung der Umgebungsschnittstelle an verschiedene RL-Frameworks. AndroidEnv eignet sich besonders für die Untersuchung von RL-Forschungsproblemen wie Exploration, hierarchisches RL, Transferlernen oder kontinuierliches Lernen in realitätsnahen Szenarien. Die Plattform bietet eine realistische Umgebung für die Entwicklung von Agenten, die potenziell auf echten Android-Geräten eingesetzt werden können. Dies eröffnet Möglichkeiten für direkte Anwendungen in der realen Welt, wie beispielsweise automatisierte Gerätetests und Qualitätssicherung.

### 4.6.2 Faiss Framework

Faiss ist eine Bibliothek für die effiziente Ähnlichkeitssuche und das Clustering hochdimensionaler Vektoren (Douze u. a. 2024). Die Bibliothek ermöglicht die Verarbeitung von Vektormengen beliebiger Größe, auch wenn diese den verfügbaren Arbeitsspeicher übersteigen.

Die Bibliothek implementiert verschiedene Methoden zur Ähnlichkeitssuche basierend auf L2-Distanz (Euklidische Distanz), Skalarprodukt oder Kosinus-Ähnlichkeit.

Zentrale Funktionen umfassen:

- Komprimierte Vektorrepräsentationen für speichereffiziente Verarbeitung.
- Indizierungsstrukturen für beschleunigte Suche.
- GPU-optimierte Implementierungen für parallele Verarbeitung.

Die Architektur von Faiss basiert auf Indextypen, die Vektormengen speichern und Suchfunktionen bereitstellen. Die verschiedenen Indextypen bieten unterschiedliche Kompromisse zwischen Suchgeschwindigkeit, Suchqualität, Speicherverbrauch und Trainingszeit.

Im Kontext des automatisierten Testens mobiler Applikationen eignet sich Faiss besonders für die effiziente Verarbeitung und Analyse von Feature-Vektoren, die aus der GUI-Analyse extrahiert werden. Die Bibliothek ermöglicht das schnelle Auffinden ähnlicher Bildschirmzustände und unterstützt damit die Erkennung bereits gesehener Bildschirmoberflächen.

### 4.6.3 Copilot+ Computer

Die vorliegende Arbeit fokussiert sich auf die Anwendung lokaler maschineller Lernsysteme. Um eine realistische und einheitliche Betrachtung der Hardwareanforderungen zu gewährleisten, wurde die Spezifikation der Copilot+ PCs von Microsoft als Referenz herangezogen (Microsoft 2024). Diese Spezifikation definiert eine Hardwarekonfiguration, die für die Ausführung aktueller KI-Anwendungen auf lokalen Systemen konzipiert wurde.

Die Mindestanforderungen für Copilot+ PCs umfassen:

- Eine Neural Processing Unit (NPU) mit einer Leistung von mindestens 40 TOPS (Trillion Operations Per Second).
- 16 GB DDR5/LPDDR5 Arbeitsspeicher.
- 256 GB SSD/UFS Speicherkapazität.

Die NPU stellt hierbei eine Schlüsselkomponente dar. Es handelt sich um einen spezialisierten Prozessor, der für die effiziente Verarbeitung KI-intensiver Aufgaben optimiert ist. Die Leistung von 40 TOPS ermöglicht die Ausführung komplexer KI-Operationen in Echtzeit, wie beispielsweise der Verwendung lokaler Sprachmodelle.

Die spezifizierten Hardwareanforderungen sind nicht willkürlich gewählt, sondern resultieren aus den Anforderungen spezifischer KI-gestützter Funktionen, die Microsoft für Copilot+ PCs vorgesehen hat[33].

Gemäß (Qualcomm 2024) ermöglicht diese Hardwarekonfiguration die Ausführung von Modellen mit mindestens 13 Milliarden Parametern. Diese Kapazität ist ausreichend für das in Abschnitt 4.4.7 beschriebene Modell, welches in der vorliegenden Arbeit Anwendung findet.

Die gewählte Spezifikation stellt einen Ausgangspunkt für die Untersuchung lokaler KI-Anwendungen im Kontext des automatisierten Testens mobiler Applikationen dar.

Die in Kapitel 4 beschriebenen Grundlagen des maschinellen Lernens bilden das theoretische Fundament für aktuelle Forschungsansätze im Bereich des automatisierten Testens mobiler Applikationen. Im Folgenden werden verwandte Arbeiten vorgestellt, die diese Techniken praktisch umsetzen und erweitern.

[33] Copilot+ Anforderungen nach Microsoft: `https://support.microsoft.com/en-us/topic/copilot-pc-hardware-requirements-35782169-6eab-4d63-a5c5-c498c3037364`

# 5 Verwandte Arbeiten im Bereich des automatisierten Testens mobiler Applikationen

In diesem Kapitel werden verwandte Arbeiten im Bereich des automatisierten Testens mobiler Applikationen vorgestellt und analysiert, die als Grundlage für das in dieser Arbeit umgesetzte System dienen.
Der Fokus liegt dabei auf Ansätzen, die LLMs für White-Box-Tests und RL für Black-Box-Tests einsetzen. Die Arbeiten werden anhand der in Tabelle 5.1 gezeigten Dimensionen strukturiert, um einen Überblick über den aktuellen Forschungsstand zu bieten.

| **Testmethode** | **White-Box-Tests** | **Black-Box-Tests** |
|---|---|---|
| **Technologie** | Large Language Model | Reinforcement Learning |
| **Anwendung** | Unit-Tests | UI-Tests |

**Tabelle 5.1:** Dimensionen der verwandten Arbeiten

Diese Kategorisierung ermöglicht es, die verschiedenen Ansätze systematisch zu vergleichen und ihre Stärken und Schwächen im Kontext der automatisierten Testgenerierung für mobile Applikationen zu bewerten. Zunächst soll die Dimension White-Box-Tests mit LLM betrachtet werden, die in dieser Arbeit eine besonders große Rolle spielt.

## 5.1 Automatisierte Testgenerierung mit LLMs

LLM haben in jüngster Zeit Fortschritte in der Codegenerierung gezeigt. M. Chen u. a. (2021) zeigen, dass LLM generell in der Lage sind, Code in verschiedenen Programmiersprachen zu generieren und die semantischen Beziehungen zwischen Codestrukturen zu erfassen.
Diese Entwicklung bildet die Grundlage für den Einsatz von LLMs in der automatisierten Testgenerierung.

Ein prominentes Beispiel für die Codegenerierungsfähigkeiten von LLMs ist beispielsweise *GitHub Copilot*.

GitHub Copilot, basierend auf OpenAIs[34] Codex-Modell, generiert Codevorschläge und vollständige Funktionen auf Basis natürlichsprachlicher Beschreibungen (M. Chen u. a. 2021).

Die generelle Fähigkeit zur Codegenerierung zeigt, dass LLMs auch für die Generierung von Softwaretests eingesetzt werden könnten. Beide Aufgaben erfordern ein Verständnis der Programmiersprache, der Codelogik und der erwarteten Funktionalität.

In den folgenden Abschnitten werden verschiedene Ansätze zur automatisierten Testgenerierung mit LLMs für White-Box- und Black-Box-Tests vorgestellt und analysiert. Dabei werden sowohl die Möglichkeiten als auch die Grenzen dieser Technologie im Kontext der Softwarequalitätssicherung betrachtet.

[34] `https://openai.com/` – OpenAI ist ein Unternehmen im Bereich der künstlichen Intelligenz, das vor allem große Sprachmodelle erstellt.

### 5.1.1 White-Box-Tests mit LLMs

White-Box-Tests mit LLM zielen darauf ab, die interne Struktur und Logik des Quellcodes zu analysieren, um Testfälle zu generieren, die eine hohe Codeüberdeckung und Fehleraufdeckung gewährleisten. Tufano, Drain u. a. (2020) und Tufano, S. K. Deng u. a. (2022) haben gezeigt, dass LLM effektiv für die Generierung von Unit-Tests eingesetzt werden können, indem sie den Code analysieren und Testfälle erstellen, die festgelegte Codepfade und -funktionen abdecken. Diese Ansätze wurden inzwischen weiterentwickelt und werden im folgenden vorgestellt.

**AlphaCodium**

Das von Ridnik, Kredo und I. Friedman (2024) entwickelte System demonstriert die Effektivität mehrstufiger Prozesse in der Programmierung. Der codeorientierte iterative Ansatz nutzt LLMs zur systematischen Lösung von Programmieraufgaben.

Der Ansatz umfasst zwei Hauptphasen:

1. Vorverarbeitungsphase: Analyse des Problems in natürlicher Sprache
2. Code-Iterationsphase: Generierung, Ausführung und Verbesserung von Code gegen manuell erstellte, öffentlich verfügbare Tests und KI-generierte Tests

Ein Kernelement von AlphaCodium ist die Erzeugung von zusätzlichen, KI-generierten Tests, da es mutmaßlich leichter ist, zusätzliche Tests zu entwickeln als eine komplett fehlerfreie Code-Lösung zu schreiben.
Dies ermöglicht eine umfassendere Validierung der generierten Lösungen. Die Effektivität von AlphaCodium wurde anhand des *CodeContest* Datensatzes (Y. Li u. a. 2022) evaluiert. Auf dem Validierungsset verbesserte sich die Genauigkeit (pass@5) von GPT-4 von 19% mit einem einzelnen verbesserten Prompt auf 44% unter Verwendung des AlphaCodium-Workflows (Ridnik, Kredo und I. Friedman 2024).

Diese Ergebnisse zeigen ebenfalls das Potenzial des mehrstufigen, iterativen Ansatzes in der LLM-basierten Codegenerierung und Testentwicklung und dienen daher als Basis dieser Arbeit.

**GitHub Copilot**

GitHub Copilot ist ein KI-gestützter Programmierassistent, der Codevorschläge und -vervollständigungen in Echtzeit generiert. Das System basiert auf dem OpenAI Codex-Modell, einem Nachfolger von GPT-3, das für Programmieraufgaben feinabgestimmt wurde. GitHub Copilot verwendet Techniken wie Prompt-Engineering und den Fill-In-the-Middle (FIM) Ansatz zur Generierung kontextrelevanter Codevorschläge. Es kann auch Tests zu bereits geschriebenen Funktionen erstellen.

Empirische Untersuchungen zeigten, dass 45,28% der von GitHub Copilot generierten Tests ausgeführt werden können (El Haji, Brandt und Zaidman 2024).

**TestPilot**

Die umfassende empirische Evaluation von LLMs für die automatisierte Generierung von Unit-Tests führte zur Entwicklung des neuartigen Tools TestPilot (Schäfer u. a. 2024). Der Ansatz zeichnet sich durch den Verzicht auf zusätzliche Feinabstimmung oder manuelle Modellanpassungen aus.
Stattdessen werden sorgfältig gestaltete Prompts verwendet, die die Signatur und Implementierung der zu testenden Funktion sowie Nutzungsbeispiele aus der Dokumentation enthalten (Schäfer u. a. 2024).

Ein besonderer Aspekt der Methode ist ein adaptiver, mehrstufiger Ansatz: Wenn ein generierter Test fehlschlägt, wird das Modell mit dem fehlgeschlagenen Test und der Fehlermeldung erneut aufgefordert, um eine Korrektur vorzunehmen (Schäfer u. a. 2024).
TestPilot wurde auf 25 npm-Paketen mit insgesamt 1.684 API-Funktionen evaluiert. Mit den generierten Tests wurde eine mittlere Anweisungsüberdeckung von 70,2% und eine Zweigüberdeckung von 52,8% erreicht.

Dies übertrifft deutlich die Leistung traditioneller Techniken zur Testgenerierung wie Nessie (Arteca u. a. 2022), bei denen nur 51,3% Anweisungsüberdeckung und 25,6% Zweigüberdeckung erreicht wurden. Interessanterweise wurde festgestellt, dass 92,8% der generierten Tests eine Ähnlichkeit von weniger als 50% zu existierenden Tests aufwiesen, was darauf hindeutet, dass das Modell nicht einfach bekannte Tests reproduziert (Schäfer u. a. 2024). In der Studie wurde auch die Qualität der generierten Tests untersucht. Es wurde festgestellt, dass ein Median von 61,4% der generierten Tests nicht-triviale Assertions enthielten, die tatsächlich Funktionalität aus dem Zielpaket überprüften (Schäfer u. a. 2024). Zudem wurde der Einfluss verschiedener Komponenten in den Prompts untersucht. Es wurde herausgefunden, dass alle einbezogenen Informationen (Funktionssignatur, Implementierung, Dokumentationskommentare und Nutzungsbeispiele) zur Effektivität der generierten Tests beitrugen.

Schließlich wurde die Leistung von TestPilot mit verschiedenen LLMs (gpt3.5-turbo[35], code-cushman-002 (siehe Codex und StarCoder in 5.1.1.1)) verglichen. Dabei zeigte sich, dass die Effektivität des Ansatzes von der Größe und dem Trainingssatz des LLM beeinflusst wird, aber nicht grundsätzlich von einem bestimmten Modell abhängt.

Die Erkenntnisse von Schäfer u. a. (2024) werden aufgegriffen und es wird ein mehrstufiges System für die Generation von Tests implementiert.

[35] `https://platform.openai.com/docs/models/gpt-3-5-turbo`

**AthenaTest**

Der von Tufano, Drain u. a. (2020) entwickelte Ansatz demonstriert die Effektivität von Finetuning für die Generierung von Unit-Tests. AthenaTest verwendet einen zweistufigen Ansatz aus Vortraining und Finetuning eines Transformermodells.
Das Vortraining erfolgt auf einem großen Korpus englischer Texte sowie Programmcode, um sowohl sprachliche als auch programmiertechnische Fähigkeiten zu erlernen. Das anschließende Finetuning nutzt einen aus GitHub extrahierten parallelen Datensatz mit 780K Test-Methoden-Paaren zur Spezialisierung auf die Testgenerierung.

Durch die Verwendung von fokalem Kontext werden nicht nur die zu testende Methode, sondern auch relevante Informationen wie Klassenname, Konstruktoren, andere Methodensignaturen und Felder berücksichtigt, was zu präziseren Ergebnissen führt. Diese zusätzlichen Informationen verbessern die Validierungsverluste um 11,1%.

Die Evaluation auf den Defects4j-Projekten zeigt, dass AthenaTest in der Lage ist, kompilierbare und ausführbare Tests zu generieren, die eine ähnliche Überdeckung wie EvoSuite erreichen. In einer Entwicklerbefragung wurden die generierten Tests zudem als besser lesbar und verständlich eingeschätzt.

Mit diesem Ansatz aus Vortraining und kontextreichem Finetuning liefert AthenaTest einen guten Ansatz für die automatische Testgenerierung. Die Arbeit demonstriert das Potential von LLMs für die Erzeugung natürlicher und effektiver Tests.
Diese Arbeit greift verschiedene Aspekte von AthenaTest auf und entwickelt sie weiter, insbesondere die Nutzung von fokalem Kontext und die zweistufige Trainingsphase.

**TestGenLLM**

Die systematische Verbesserung bestehender Unit-Tests bei Meta zeigt das Potenzial LLM-basierter Testverbesserung. TestGen ist ein von (Alshahwan u. a. 2024) entwickeltes LLM-basiertes Tool zur Verbesserung von Unit-Tests bei Meta[36]. Ihr Ansatz nutzt LLMs , um bestehende, von Menschen geschriebene Tests zu erweitern und zu verbessern.
Ein wesentlicher Unterschied zu anderen Arbeiten besteht darin, dass TestGen-LLM vollständig ausgearbeitete Softwareverbesserungen empfiehlt, die mit überprüfbaren Garantien für Verbesserungen und Nicht-Regression des bestehenden Verhaltens untermauert sind.

TestGen-LLM verwendet einen mehrstufigen Filterprozess, um die Qualität der generierten Tests sicherzustellen:

1. Kompilierbarkeit: Nur Tests, die erfolgreich kompiliert und gebaut (build) werden können, werden weiter betrachtet.

2. Zuverlässigkeit: Tests müssen bei wiederholter Ausführung unter gleichen Bedingungen durchgängig dasselbe Ergebnis liefern. Dies bedeutet, dass der Test entweder immer besteht oder immer fehlschlägt, ohne von nicht-deterministischen Faktoren wie Timing oder Systemzustand beeinflusst zu werden.

3. Überdeckungsverbesserung: Nur Tests, die die bestehende Codeüberdeckung erhöhen, werden akzeptiert.

Dieser Prozess stellt sicher, dass die generierten Tests nicht nur syntaktisch korrekt sind, sondern auch einen messbaren Mehrwert für die Testsuite bieten.

[36] `https://www.meta.com/de/` – Meta ist die Muttergesellschaft von Facebook, Instagram, WhatsApp und forscht unter anderem im Bereich des maschinellen Lernens. Viele der Modelle von Meta werden frei veröffentlicht (siehe: `https://huggingface.co/meta-llama`).

In einer Evaluation auf Instagram-Produkten[37] für Reels und Stories stellten die Autoren fest, dass 75% der von TestGen-LLM generierten Testfälle korrekt gebaut wurden. 57% bestanden alle Testkriterien und erhöhten insgesamt die Überdeckung um 25%.
Bei bei Meta-internen Programmierveranstaltungen für Instagram und Facebook verbesserte TestGen-LLM 11,5% aller Klassen, auf die es angewendet wurde, wobei 73% seiner Empfehlungen von Meta-Softwareingenieuren für den Produktionseinsatz akzeptiert wurden (Alshahwan u. a. 2024).

Ein besonderer Aspekt von TestGen-LLM ist sein mehrstufiger, modularer Ansatz („ensemble“ genannt), der verschiedene LLMs und Prompting-Strategien kombiniert (Alshahwan u. a. 2024).
In der Studie wurde mit verschiedenen Prompts experimentiert und herausgefunden, dass unterschiedliche Strategien jeweils einzigartige Beiträge zur Testgenerierung lieferten. Dies unterstreicht die Vorteile eines vielseitigen Ansatzes in der LLM-basierten Testgenerierung.

Die Autoren untersuchten auch den Einfluss des Sampling-Parameters „Temperatur“ auf die Qualität der generierten Tests. Die Studie zeigte, dass eine Temperatur von 0 die besten Ergebnisse lieferte. Dies bedeutet, dass für die Testgenerierung vorhersagbare und reproduzierbare Ausgaben, die sich strikt an etablierte Testmuster halten, vorteilhafter sind als kreative Variationen.

[37] `https://www.instagram.com/` Instagram ist eine Social-Media-Plattform zum Teilen von Fotos und Videos.

Ein weiterer wichtiger Aspekt ist die Fähigkeit von TestGenLLM, den Stil existierender Tests nachzuahmen und sich in bestehende Entwicklungsprozesse einzubinden. Die Autoren betonen die Bedeutung lokaler Systeme ohne Cloud-Anbindung für den Datenschutz und die Vertraulichkeit des Quellcodes. Diese Eigenschaften führten zu einer hohen Akzeptanz bei den Entwicklern, da die generierten Tests gut in die bestehende Codebasis passten.

Diese Arbeit greift die Erkenntnisse bezüglich der Temperatur, die Bewertung der Qualität der generierten Tests, der besseren Erklärbarkeit und die Integration in bestehende Entwicklungsprozesse auf.

**ChatUniTest**

Die Integration von Validierungs- und Reparaturmechanismen in die LLM-basierte Testgenerierung ist eine wichtige Erweiterung der bisherigen LLMs-Strategien. ChatUniTest wurde von Y. Chen u. a. (2024) als Framework zur Generierung von Unit-Tests auf Basis von LLM entwickelt. Ähnlich wie TestPilot nutzt ChatUniTest einen mehrstufigen Ansatz, geht jedoch über die reine Testgenerierung hinaus und integriert umfangreiche Validierungs- und Reparaturmechanismen. Das System arbeitet in drei Hauptphasen: Vorverarbeitung, Generierung und Nachverarbeitung.

In der Vorverarbeitungsphase wird der Quellcode mittels eines abstrakten Syntaxbaums analysiert, um Informationen auf Klassen- und Methodenebene zu sammeln. Eine Besonderheit ist der adaptive zentrale Kontextgenerierungsmechanismus (eng. adaptive focal context generation), der dynamisch relevanten Kontext zum Prompt hinzufügt, solange die Kontextlänge des LLM nicht überschritten wird.
Dieser Ansatz unterscheidet sich von TestGenLLM , das sich auf bestehende Tests konzentriert, und ermöglicht eine effizientere Nutzung des Kontextfensters als beispielsweise TestPilot. Ähnlich wie bei AlphaCodium wird CoT Prompting eingesetzt, um die Qualität der generierten Tests zu verbessern.

Die Generierungsphase nutzt die erstellten Prompts zur Generierung von Tests. Das System generiert dabei vollständige Testklassen. Wenn das Modell zunächst nur eine einzelne Testmethode erzeugt, wird diese automatisch in eine vorgefertigte Testklassenstruktur integriert. Diese Struktur enthält alle notwendigen Importe, Klassendeklarationen und Setup-Methoden. Dieser Ansatz baut auf der Methode von TestGenLLM auf und erweitert sie durch die vollautomatische Einbettung der Testmethoden.

Die Nachverarbeitungsphase unterscheidet ChatUniTest von anderen LLM-basierten Ansätzen durch einen dreistufigen Validierungsprozess:

- syntaktische Korrektheit.
- Kompilierbarkeit.
- Ausführbarkeit.

Für die Reparatur fehlerhafter Tests implementiert das System zwei Strategien. Die regelbasierte Reparatur korrigiert häufige und bekannte Fehler anhand vordefinierter Muster. Bei unbekannten Problemen, bei denen keine statische Reparatur ausreicht kommt eine LLM-basierte Reparatur zum Einsatz. Diese Kombination aus regelbasierter und modellbasierter Korrektur verbessert die kompilierbarkeit und Zuverlässigkeit der Tests im Vergleich zur ausschließlich modellbasierten Korrektur von TestPilot.

Eine Evaluation auf vier Java-Projekten, darunter sowohl etablierte Bibliotheken als auch neuere Projekte außerhalb des Trainingsdatensatzes, ergab eine durchschnittliche Codeüberdeckung von 59,6%. Vergleichbare Werkzeuge wie EvoSuite erreichten 38,2%.

ChatUniTest zeigt durch die Verbindung von kontextsensitiver Promptgenerierung und hybrider Testreparatur neue Möglichkeiten für die automatisierte Testgenerierung auf. Die Ergebnisse bestätigen die Wirksamkeit mehrstufiger und iterativer Ansätze in der LLM-basierten Testentwicklung, analog zu den Methoden von AlphaCodium und TestPilot.

Diese Arbeit greift die Erkenntnisse bezüglich des mehrstufigen Verbesserungsprozesses auf.

**Initial Investigation of ChatGPT**

Die von Guilherme und Vincenzi (2023) durchgeführte Studie evaluierte systematisch die Fähigkeiten von ChatGPT zur Generierung von Java Unit-Tests. Der Ansatz nutzt die OpenAI API mit dem Modell gpt-3.5-turbo und untersucht den Einfluss unterschiedlicher Temperatur-Parameter auf die Qualität der generierten Tests.
In der empirischen Evaluation mit 33 Java-Programmen erreichten die generierten Tests eine durchschnittliche Anweisungsüberdeckung von 93,5%. Bei der Kombination aller erfolgreichen Testfälle in einer Testsuite wurde eine Überdeckung von 99,2% erreicht. Diese Ergebnisse sind vergleichbar mit traditionellen Testgeneratoren wie EvoSuite. Ein zentraler Aspekt der Studie ist der vollständig automatisierte Ansatz ohne manuelle Nachbearbeitung der Tests. Die Ergebnisse zeigen, dass LLMs bereits ohne Interaktion qualitativ hochwertige Unit-Tests generieren können, was für die vorliegende Arbeit von besonderer Relevanz ist. Die Methode der iterativen Verbesserung der Prompts und die systematische Evaluation verschiedener Modellparameter bieten wichtige Erkenntnisse für die Gestaltung von LLM-basierten Testgeneratoren.

#### 5.1.1.1 Beispiele angepasster LLMs

Im Folgenden werden mehrere LLMs vorgestellt, die speziell für die Generierung von Code angepasst wurden. Der Fokus liegt dabei auf den Anpassungsmethoden und deren Anwendung in der Testgenerierung.

**AlphaCode**

AlphaCode ist ein von DeepMind[38] entwickeltes LLM, das darauf ausgelegt ist, Code in verschiedenen Programmiersprachen zu generieren und mehrschrittige Programmieraufgaben mit verschachtelten Algorithmen zu lösen.

Es hat in Programmierwettbewerben eine Leistung erzielt, die es in die oberen 54% der Teilnehmer einordnet (Y. Li u. a. 2022) und liegt damit vor vielen menschlichen Teilnehmern. Im Rahmen von AlphaCode wurde auch der CodeContests Datensatz erstellt, der eine manuell ausgewählte und überprüfte Sammlung von Code-Herausforderungen enthält (Y. Li u. a. 2022). Es wurde speziell für Probleme aus Programmier-Wettbewerben trainiert.

Drei Komponenten wurden als entscheidend für diese Leistung identifiziert:

1. Ein umfangreicher und bereinigter Datensatz für Training und Evaluation
2. Große und effizient zu sampelnde Transformer-basierte Architekturen
3. Umfangreiche Modellvorhersagen zur Exploration des Lösungsraums, gefolgt von einer automatischen Validierung der Programmergebnisse zur Auswahl der besten Lösungen

Analysen zeigten, dass AlphaCode nicht einfach nur Teile früherer Lösungen kopiert oder systematische Schwächen in der Problemstruktur ausnutzt. Das Modell löst erfolgreich komplexe Aufgaben wie Graphtraversierung, dynamische Programmierung oder Optimierungsprobleme. Die gelösten Aufgaben erfordern die Kombination verschiedener Algorithmen und Datenstrukturen.

[38] DeepMind ist ein zu Google gehörendes Unternehmen im Bereich des maschinellem Lernen und neuronaler Netzwerke (`https://deepmind.google/`)

### Codex

OpenAI entwickelte eine Reihe von LLMs, die speziell für die Codegenerierung trainiert wurden (M. Chen u. a. 2021). Diese Modelle finden unter anderem in GitHub Copilot Anwendung. Einige Codex-Modelle gelten inzwischen als veraltet und überholt[39]. Codex wurde durch Feinabstimmung von GPT-3 auf einem Datensatz von 159 GB nicht redundanter Python-Dateien aus öffentlich verfügbarem Code von GitHub entwickelt. Die Nicht-Redundanz wurde durch Entfernung von Duplikaten und stark ähnlichen Codesequenzen sichergestellt.

Um die Leistung von Codex zu evaluieren, wurde der HumanEval-Datensatz erstellt, der 164 handgeschriebene Programmierprobleme umfasst. Jedes Problem enthält eine Funktionssignatur, eine Docstring, einen Funktionskörper und mehrere Unittests.
Die Leistung wurde mit der pass@k-Metrik gemessen, die den Prozentsatz der Probleme angibt, für die das Modell mindestens eine korrekte Lösung aus k generierten Kandidaten findet. In Evaluationen auf dem HumanEval-Datensatz erreichte das größte Codex-Modell (Codex-12B) eine Pass@1-Rate von 28,8% bei Problemlösungsaufgaben (M. Chen u. a. 2021).

Mit 100 Stichproben pro Problem (Pass@100) stieg die Erfolgsquote auf 72,31%. Beim Übersetzen von Python zu JavaScript erzielte es eine Genauigkeit von 74,8% bei der funktionalen Korrektheit. Eine wichtige Erkenntnis aus der Codex-Studie ist, dass wiederholtes Sampling aus dem Modell eine effektive Strategie zur Produktion funktionierender Lösungen für schwierige Aufgaben ist.

[39] Deprecation Notiz von Openai: `https://platform.openai.com/docs/deprecations/2023-03-20-codex-models`

Trotz dieser Leistungen hat Codex auch Einschränkungen, wie Schwierigkeiten mit Funktionsdokumentationen mit mehreren Parametern, Randbedingungen und Ausnahmebehandlungen und der Bindung von Operationen an Variablen.

**Gemma**

Die Open-Source-Modellfamilie von Google demonstriert die Bedeutung mehrstufiger Trainingsansätze. Die Modelle sind in verschiedenen Größen verfügbar, von 2 Milliarden bis 7 Milliarden Parametern.

Ein wesentlicher Aspekt der Gemma-Modelle ist ihr zweistufiger Feinabstimmungsprozess, der Supervised Fine-Tuning (SFT) (siehe Abschnitt 4.4.6.2) und Reinforcement Learning from Human Feedback (Bestärkendes Lernen aus menschlichem Feedback, RLHF) kombiniert. Beim SFT werden die Modelle auf einer Mischung aus synthetischen und von Menschen generierten Prompt-Response-Paaren trainiert. Ziel dieser Phase ist die Minimierung der Verlustfunktion zwischen den Modellvorhersagen und den menschlich erstellten Antworten.

Im Anschluss an das SFT wird RLHF angewendet. Hierbei wird ein Belohnungsmodell auf Basis von menschlichen Präferenzurteilen trainiert. Die Zielfunktion maximiert dabei die Übereinstimmung der Modellausgaben mit diesen Präferenzurteilen. Die Policy verwendet dieselben Prompts wie in der SFT-Phase, passt jedoch die Ausgaben hinsichtlich der gelernten Präferenzen an.

Nach dem Training werden die Modelle aus beiden Phasen durch Modellmittelung kombiniert. Dieses Vorgehen reduziert nachweislich die Rate von Faktengenerierung ohne Quellenbelege und verbessert die Genauigkeit der Modellausgaben in standardisierten Tests wie HumanEval und MBPP. Die resultierenden Gemma-Modelle zeigen verbesserte Leistungen in diesen Benchmarks, wobei sie insbesondere bei der Vermeidung von Fehlinformationen und der Einhaltung definierter Nutzungsrichtlinien gute Ergebnisse erzielen.

Ähnliche Ansätze zur Verbesserung bereits vortrainierter Modelle finden im vorgestellten System Anwendung.

**StarCoder**

Die von Hugging Face entwickelten Modelle repräsentieren einen bedeutenden Fortschritt in der Open-Source-Codegenerierung. StarCoder und StarCoderBase sind von Hugging Face entwickelte 15,5B-Parameter-Modelle für Codegenerierung und -verbesserung in verschiedenen Programmiersprachen (R. Li u. a. 2023). StarCoderBase wurde auf 1 Billion Tokens aus The Stack trainiert, einer Sammlung von GitHub-Repositories unter Open-Source-Lizenzen wie MIT, Apache und BSD (Kocetkov u. a. 2022). Diese Lizenzen erlauben die freie Verwendung, Modifikation und Weiterverbreitung des Codes, auch für kommerzielle Zwecke.

In Evaluationen auf dem HumanEval-Datensatz erreichte StarCoder eine Pass@1-Rate von 33,6%, was eine Verbesserung gegenüber früheren Modellen darstellt. Auf dem MBPP-Benchmark erzielte es eine Pass@1-Rate von 64,5%. In dieser Arbeit wurden ähnliche Methoden zur weiteren Anpassung kleinerer Modelle auf der Basis größerer Modelle verwendet.

Diese angepassten LLMs zeigen das Potenzial des Fine-Tunings für konkrete Anwendungsfälle in der Softwareentwicklung. Die Modelle beweisen ihre Fähigkeiten sowohl bei einfachen als auch schwierigeren Programmieraufgaben, die ein tieferes Verständnis von Algorithmen und natürlicher Sprache voraussetzen. Diese Fähigkeiten durch gezieltes Fine-Tuning für die Testerstellung weiterzuentwickeln, ist Gegenstand dieser Arbeit.

### 5.1.2 Black-Box-Tests mit LLMs

Black-Box-Tests mit LLMs zielen darauf ab, die Funktionalität und das Verhalten von mobilen Applikationen zu überprüfen, ohne Kenntnis der internen Struktur oder des Quellcodes.

Diese Methode bietet besondere Vorteile, da sie die Vielfalt der Geräte, Betriebssysteme und Nutzungsszenarien berücksichtigen kann. LLM-basierte Black-Box-Tests ermöglichen eine flexible und anpassungsfähige Testgestaltung, die schnell auf Änderungen in der Benutzeroberfläche oder neue Funktionen reagieren kann. In den folgenden Abschnitten werden zwei Ansätze vorgestellt, die das Potenzial dieser Technologie demonstrieren: ScreenAI und AppAgent.

**ScreenAI**

Das in Baechler u. a. (2024) entwickelte Modell, ist ein multimodales Vision-Language-Modell, das speziell für das Verständnis von GUIs und Infografiken konzipiert wurde.
Die Funktionsweise von ScreenAI lässt sich wie folgt beschreiben (Baechler u. a. 2024):

1. **Bildverarbeitung:** Das Modell verwendet einen Vision-Encoder (ViT), um das Eingabebild in eine Sequenz von Bildmerkmalen umzuwandeln.

2. **Textverarbeitung:** Ein T5-Encoder (Text-to-Text Transfer Transformer) (Raffel u. a. 2020) verarbeitet den Eingabetext und generiert Textmerkmale.

3. **Multimodale Verarbeitung:** Die Bild- und Textmerkmale werden in einem multimodalen Encoder kombiniert, der Aufmerksamkeitsmechanismen verwendet, um Beziehungen zwischen visuellen und textuellen Elementen zu erfassen.

4. **Ausgabegenerierung:** Ein T5-Decoder generiert die Ausgabe basierend auf den kombinierten multimodalen Merkmalen.

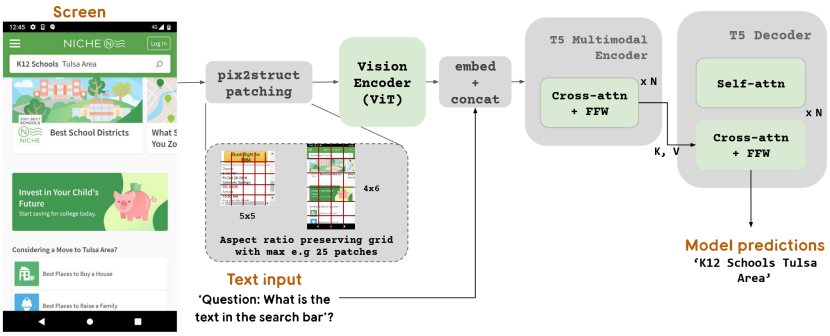



**Abbildung 5.1:** Architektur von ScreenAI. Auf einen Vision-Encoder folgt ein multimodaler Encoder (Baechler u. a. 2024).

Ein zentrales Element von ScreenAI ist die Verwendung des „Screen Schema“, einer textbasierten Repräsentation für GUIs. Diese enthält detaillierte Informationen über GUI-Elemente, deren Texte, Beschreibungen und räumliche Anordnung. Das Schema wird während des Trainings verwendet und ermöglicht es dem Modell vielschichtige GUI-Layouts zu verstehen und zu manipulieren. Die Architektur von ScreenAI ist in Abbildung 5.1 zu sehen.

Das Training von ScreenAI erfolgt in zwei Phasen:

1. **Pretraining:** Hier wird das Modell auf einer großen Mischung von Datensätzen trainiert, die GUI-Screenshots, Infografiken und andere visuelle Daten umfassen. Dies ermöglicht dem Modell, allgemeine Fähigkeiten zum Verstehen von visuellen Layouts zu entwickeln.

2. **Fine-Tuning:** In dieser Phase wird das Modell auf spezifischen Aufgaben wie GUI-Element-Annotation, Frage-Antwort-Aufgaben, GUI-Navigation und Zusammenfassung von Bildschirminhalten trainiert.

ScreenAI erzielt bei nur 5 Milliarden Parametern state-of-the-art Ergebnisse auf verschiedenen GUI- und infografikbasierten Benchmarks. Diese Leistung wird der effizienten Architektur, dem umfassenden Trainingsansatz und der innovativen Verwendung des Screen Schemas zugeschrieben. Multimodale Transformer-Modelle zur Oberflächenerkennung von mobilen Applikationen, die nach einem ähnlichen Prinzip wie ScreenAI aufgebaut sind, werden im entwickelten System eingesetzt.

**AppAgent**

Der Ansatz des multimodalen Testens wird durch das von C. Zhang u. a. (2023) vorgestellte Agentensystem weiterentwickelt. Das System ermöglicht im Gegensatz zu anderen Ansätzen (auch ScreenAI) direkte GUI-Interaktionen durch eine vereinfachte Aktionsraumdarstellung, ohne auf systeminterne Zugriffe angewiesen zu sein.

Die Funktionsweise von AppAgent lässt sich in zwei Hauptphasen unterteilen (C. Zhang u. a. 2023):

1. **Explorationsphase:**

    - Der Agent lernt die Funktionsweise der App entweder durch autonome Interaktion oder durch Beobachtung menschlicher Demonstrationen.
    - Bei der autonomen Interaktion führt der Agent verschiedene Aktionen aus und beobachtet deren Auswirkungen auf die GUI.
    - Bei der Beobachtung menschlicher Demonstrationen zeichnet der Agent die vom Menschen ausgeführten Aktionen und deren Ergebnisse auf.

- Während dieser Phase erstellt der Agent eine Wissensbasis über die Funktionen und das Layout der App.

2. **Einsatzphase:**

   - Der Agent nutzt das in der Explorationsphase erworbene Wissen, um mehrschrittige Aufgaben auszuführen.
   - Er interpretiert die von den Autoren vordefinierten, textuellen Aufgabenstellung, plant die erforderlichen Schritte und führt diese auf der GUI aus.
   - Bei jedem Schritt beobachtet der Agent den aktuellen Zustand der GUI, plant die nächste Aktion und führt diese aus.

AppAgent verwendet GPT-4 als Basismodell, was ihm ermöglicht, natürlichsprachliche Anweisungen zu verstehen und in konkrete Aktionen auf der Benutzeroberfläche umzusetzen. Das System arbeitet mit einem vereinfachten Aktionsraum, der grundlegende Interaktionen umfasst (C. Zhang u. a. 2023):

- Tippen auf GUI-Elemente.
- Wischen in verschiedene Richtungen.
- Texteingabe.
- Zurücknavigation.
- Beenden der Aufgabe.

Diese Aktionen werden auf nummerierte GUI-Elemente angewendet, was eine präzise Steuerung ohne die Notwendigkeit exakter Bildschirmkoordinaten ermöglicht.

AppAgent kann sich an neue Apps anpassen, ohne extra Training für jede einzelne App zu benötigen. Dies wird durch die Kombination des leistungsfähigen Sprachmodells mit der flexiblen Explorationsphase erreicht (C. Zhang u. a. 2023). Die Leistungsfähigkeit von AppAgent wurde in Experimenten mit 50 Aufgaben über 10 verschiedene Apps evaluiert. Das System zeigte dabei eine hohe Effektivität bei der Bewältigung dieser Aufgaben (C. Zhang u. a. 2023). AppAgent ist an der Schnittstelle zwischen Black-Box-Tests Tests mit LLMs und Black-Box-Tests mit RL, für das im nächsten Abschnitt noch weitere Beispiele gegeben werden, angesiedelt.

Die Ansätze zur Steuerung von mobilen Applikationen und das darauf aufbauende Verständnis für die Oberfläche werden im entwickelten System adaptiert.

## 5.2 Automatisierte Testgenerierung mit RL

RL hat in den letzten Jahren zunehmend an Bedeutung für die automatisierte Testgenerierung gewonnen, insbesondere im Bereich der mobilen Applikationen. Im Gegensatz zu herkömmlichen Testmethoden ermöglicht RL einen adaptiven Ansatz, bei dem ein Agent durch Interaktion mit der Anwendung lernt, effektive Appdurchläufe zu generieren. Dieser Ansatz ist besonders nützlich für Black-Box-Tests, bei denen die interne Struktur der Anwendung unbekannt ist und trotzdem die Funktionalität und das Verhalten von mobilen Applikationen überprüft werden kann. RL-Algorithmen werden genutzt, um Benutzerinteraktionen zu simulieren und die Benutzeroberfläche zu interpretieren.

Im Kontext mobiler Apps bietet diese Methode besondere Vorteile, da die Vielfalt der Geräte, Betriebssysteme und Nutzungsszenarien berücksichtigt werden kann (Romdhana u. a. 2022). Dieser Abschnitt präsentiert eine Übersicht über aktuelle Forschungsarbeiten, die RL-Techniken für das Testen von mobilen Applikationen einsetzen, mit besonderem Fokus auf Black-Box-Testmethoden.

**Q-Learning für Desktop-Software**

In Bauersfeld und Vos (2012) wurde ein Ansatz für das automatisierte Testen von Desktop-Software mittels Q-Learning entwickelt (siehe Abschnitt 4.5.1.1). Das Ziel der Arbeit war es, die Effizienz der GUI-Erkundung zu steigern, indem die Wahrscheinlichkeitsverteilung der Ereignissequenzen dynamisch angepasst wird. Die Besonderheit des Ansatzes liegt in der Belohnungsfunktion, durch die neue oder selten ausgeführte Aktionen bevorzugt werden. Dadurch wird eine umfassendere Erkundung der Anwendung erreicht, da der Agent dazu angeregt wird, bisher unerforschte Bereiche der GUI zu untersuchen.

In dem Algorithmus wird die Wahrscheinlichkeitsverteilung über den Ereignissequenzraum zur Laufzeit geändert, um die Erkundung der GUI zu begünstigen. Die Belohnungsfunktion $R : A \times S \times S \to \mathbb{R}$ wird definiert als (Bauersfeld und Vos 2012):

$$R(a, s, s') := \begin{cases} r_{init}, & \text{wenn } x_a = 0, \\ \frac{1}{x_a}, & \text{sonst} \end{cases} \tag{5.1}$$

Die Belohnungsfunktion erhält als Parameter die auszuführende Aktion $a$ aus dem Aktionsraum $A$, den aktuellen Zustand $s$ und den resultierenden Zustand $s'$ aus dem Zustandsraum $S$. Die Variable $x_a$ bezeichnet dabei die vom System global erfasste Anzahl der bisherigen Ausführungen der Aktion $a$. Der Wert $r_{init}$ ist ein hoher positiver Initialwert, der neue Aktionen stark belohnt.

Diese Arbeit legte wichtige Grundlagen für die Anwendung von RL in der GUI-Testautomatisierung. Die Erkenntnisse wurden in der im folgenden beschriebenen Arbeit auf mobile Plattformen übertragen, obwohl sie ursprünglich für Desktop-Software konzipiert waren (Adamo u. a. 2018).

**Q-Learning für Android-Apps**

In (Adamo u. a. 2018) wurde der Q-Learning-Ansatz von (Bauersfeld und Vos 2012) spezifisch für das GUI-Testen von Android-Apps adaptiert.

Die Methode zeichnet sich durch die Verwendung dynamischer Ereignisextraktion aus, wodurch kein vorbestehendes abstraktes Modell der zu testenden Anwendung erforderlich ist. Dadurch wird die Flexibilität und Anwendbarkeit des Ansatzes erheblich erhöht.

Der Kern der Arbeit liegt in einem Belohnungsmechanismus, durch den die Ausführung neuer oder selten ausgeführter Ereignisse gefördert wird. Die Belohnung für ein Ereignis wird dabei umso kleiner, je häufiger es bereits ausgeführt wurde. Dadurch wird der Agent motiviert, bisher wenig explorierte Bereiche der Anwendung zu untersuchen.

Dieser Ansatz führte zu einer signifikant höheren Codeüberdeckung im Vergleich zu zufälligem Testen. In der empirischen Evaluation wurde eine um 3,31% bis 18,83% höhere Codeüberdeckung erreicht(Adamo u. a. 2018).

Die Weiterentwicklung dieser Methoden führte zur Anwendung von DL, das die Leistungsfähigkeit tiefer neuronaler Netze nutzt, um Testszenarien mit verschachtelten Abhängigkeiten und parallelen Ausführungspfaden zu bewältigen.

### Neugierigkeitsgetriebenes Lernen (Curiosity driven Learning)

Neugierigkeitsgetriebenes RL erweitert klassische RL-Ansätze durch die Integration intrinsischer Motivation. Die Grundidee besteht in der Implementierung eines zusätzlichen Belohnungssignals, das die Exploration unbekannter Zustände fördert (Pathak u. a. 2017; Frank u. a. 2014). Im Kontext des Testens mobiler Apps ermöglicht dieser Ansatz eine systematische Erkundung des Zustandsraums der Benutzeroberfläche.

Die Architektur neugierigkeitsgetriebener RL-Systeme basiert auf vier Komponenten (Pathak u. a. 2017; Frank u. a. 2014):

1. **Intrinsische Belohnungsfunktion:** Parallel zur extrinsischen Belohnung für definierte Testziele implementiert das System eine intrinsische Belohnungsfunktion, die die Exploration neuer Zustände quantifiziert.

2. **Prädiktives Zustandsmodell:** Ein neuronales Netzwerk modelliert die Zustandsübergänge der App und ermöglicht Vorhersagen über resultierende Zustände nach Aktionsausführung.

3. **Neuigkeitsmetrik:** Die Abweichung zwischen prognostiziertem und tatsächlichem Zustand wird als Grundlage für die Berechnung der intrinsischen Belohnung verwendet. Die mathematische Formulierung variiert je nach Implementierung, basiert jedoch typischerweise auf Distanzmetriken im Zustandsraum.

4. **Explorations-Exploitations-Mechanismus:** Die Integration der intrinsischen und extrinsischen Belohnungen erfolgt durch eine gewichtete Summe oder ähnliche Aggregationsmechanismen.

### Hide-and-seek

Die in Baker u. a. (2020) beschriebene Simulation (zu sehen in Abbildung 5.2) ermöglicht es Agenten, durch einfache Regeln und kontinuierliches Training fortgeschrittene Strategien und Verhaltensweisen zu entwickeln. Diese Verhaltensweisen sind nicht explizit programmiert, sondern entwickeln sich aus der Interaktion der Agenten mit ihrer Umgebung und miteinander.

Die Agenten sind in zwei Gruppen eingeteilt: *Sucher* und *Verstecker*.
Die Verstecker haben das Ziel, sich zu verstecken und außerhalb des Sichtfelds der Sucher zu bleiben. In einer Vorbereitungsphase können sie die Umgebung manipulieren, indem sie Objekte wie Kisten und Rampen bewegen und fixieren, um Barrieren zu errichten.
Die Sucher hingegen müssen nach Ablauf dieser Phase die Verstecker finden und in Sichtweite behalten. Beide Gruppen können mit Objekten in der Umgebung interagieren. Die Agenten erhalten Belohnungen basierend auf ihren Erfolgen. Verstecker erhalten die Belohnung für das erfolgreiche Verstecken, Sucher für das Entdecken der Verstecker.

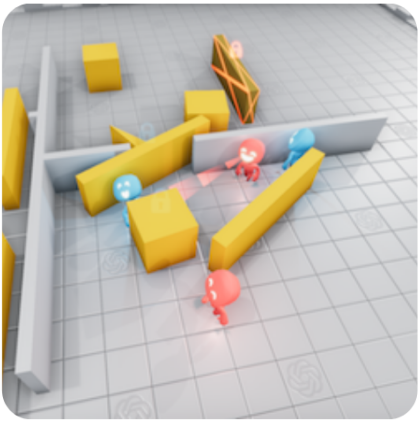


**Abbildung 5.2:** Darstellung der Hide-and-Seek Umgebung aus (Baker u. a. 2020).

Im Verlauf des Trainings entwickeln die Agenten zunehmend komplexere Strategien, wie das „Box Surfing“ der Sucher. Hierbei nutzen die suchenden Agenten Rampen, um auf Boxen zu gelangen und diese dann als Fortbewegungsmittel zu verwenden, um Hindernisse zu überwinden, die von den sich versteckenden Agenten (Verstecker) errichtet wurden.

Das „Box Surfing“ kann als eine Form der Nutzung nicht intendierter Mechaniken der Simulationsumgebung betrachtet werden. Die Agenten entwickelten eine Methode, die Physik der Umgebung auf eine Weise zu nutzen, die über das ursprüngliche Design hinausging. Dies weist Parallelen zu Fehlern oder Defekten (siehe Abschnitt 3.1.2) auf, die in Softwaresystemen, mit vielen Funktionen, auftreten können.
Im Kontext des Softwaretestens demonstriert das „selbstentwickelnde“ Verhalten das Potenzial von Curiosity-driven RL, unvorhergesehene Szenarien zu entdecken.

Ähnlich wie die Agenten im Hide-and-Seek-Spiel die Grenzen ihrer Umgebung erkundeten, könnte ein RL-Agent beim Testen einer mobilen Applikation neuartige Interaktionsmuster oder Eingabesequenzen identifizieren, die zu bisher unentdeckten Fehlern führen könnten.

**Neugierbasierte Testansätze mit Q-Learning**

In der Arbeit von M. Pan u. a. (2020) wird mit Q-testing ein Reinforcement Learning-basierter Ansatz für das Testen von Android-Apps vorgestellt, der auf dem Prinzip der Neugier basiert.
Das System nutzt einen Q-Learning-Algorithmus (siehe Abschnitt 4.5.1) kombiniert mit einer neugierbasierten Explorationsstrategie. Ein neuronales Netzwerk bewertet dabei die Ähnlichkeit von Zuständen auf funktionaler Ebene, um die Belohnung für neue, unerforschte Zustände zu berechnen.
In einer empirischen Studie mit 50 Open-Source-Android-Apps erreichte Q-testing eine durchschnittliche Anweisungsüberdeckung von 46,62% und identifizierte 197 voneinander unterscheidbare Abstürze. Demgegenüber fand Monkey (siehe Abschnitt 3.3.4) nur 109 Abstürze.

Der darauf aufbauende ARES-Ansatz von (Romdhana u. a. 2022) erweitert diese Idee durch fortgeschrittene DL-Algorithmen wie DDPG, SAC und TD3 (siehe Abschnitt 4.5.1). Die implementierte Belohnungsfunktion belohnt das Entdecken neuer Zustände sowie das Aufdecken von Abstürzen und bestraft gleichzeitig das Verlassen der zu testenden Applikation.
In empirischen Evaluierungen mit 68 Android-Apps erreichte ARES eine durchschnittliche Codeüberdeckung von 54,2% und identifizierte 171 voneinander unterscheidbare Abstürze. Diese Ergebnisse übertrafen die Leistung etablierter Werkzeuge wie TimeMachine (50,4% Überdeckung, 179 Abstürze), Sapienz[40] (48,8% Überdeckung, 103 Abstürze) und Monkey (siehe Abschnitt 3.3.4) (43,9% Überdeckung, 51 Abstürze) (Romdhana u. a. 2022).

## 5.3 Zusammenfassung

Der aktuelle Stand der Technik im Bereich des automatisierten Testens mobiler Applikationen zeigt eine konvergente Entwicklung verschiedener Ansätze. LLMs haben sich als funktional für die Aufdeckung von Defekten durch White-Box-Tests erwiesen, während RL-Methoden erfolgreich für Black-Box-Tests zur Identifikation konkreter Ausfälle eingesetzt werden.

Diese Entwicklung zeigt einen klaren Trend zur Integration verschiedener Technologien, um umfassendere und effektivere Teststrategien für mobile Apps zu entwickeln.

[40] TimeMachine und Sapienz sind Werkzeuge für die automatisierte Suche nach Fehlern (`https://www.ghs.com/products/timemachine.html` und `https://engineering.fb.com/2018/05/02/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/`)

# 6 Konzept und Architektur eines KI-basierten Testframeworks

In diesem Kapitel wird die Architektur eines Testframeworks für mobile Applikationen vorgestellt, welches LLMs (siehe Abschnitt 4.4 und 3.2.1.2) für White-Box-Tests und RL (siehe Abschnitt 4.5 und 3.2.1.2) für Black-Box-Tests kombiniert.

Das Framework wird durch eine zweiphasige Pipeline, dargestellt in Abbildung 6.1, für beide Testansätze realisiert:

1. Die *Trainingsphase* bereitet die jeweiligen Modelle auf ihre Aufgaben vor
2. Die *Operationsphase* führt die Tests durch

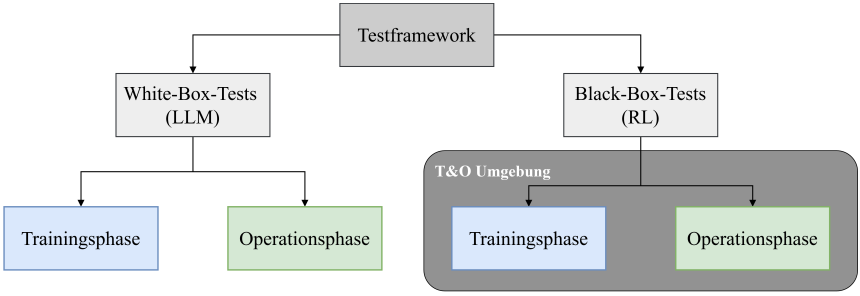



**Abbildung 6.1:** Gesamtübersicht Architektur des Testframeworks

Die folgenden Abschnitte erläutern den Gesamtaufbau des Frameworks, dessen Kernkomponenten und deren Interaktionen innerhalb der Testpipeline.

## 6.1 Aufbau des Testframeworks

Durch das Framework werden die in Abschnitt 3.1.2 klassifizierten Fehlerarten systematisch adressiert.

Die **White-Box-Tests**, basierend auf den in Abschnitt 3.2.1.2 beschriebenen Prinzipien, konzentrieren sich auf die Identifikation von *Fehlern* und *Defekten* im Quellcode durch die automatisierte Generierung von Unit-Tests.

Komplementär dazu ermöglichen die **Black-Box-Tests**, entsprechend den in Abschnitt 3.2.1.2 dargelegten Grundsätzen, die Aufdeckung von *Störungen* durch fehlerhafte Benutzerinteraktionen sowie die daraus resultierenden *Ausfälle*.

Diese komplementären Ansätze ermöglichen eine umfassende Fehlererkennung entlang der gesamten Fehlerkette: Vom ursprünglichen *Fehler* im Code über resultierende *Defekte* und *Störungen* bis hin zu sichtbaren *Ausfällen*.

Die Architektur des Frameworks integriert etablierte Standards und Normen in einem Gesamtkonzept. Grundlegend werden die Anforderungen der ISO/IEC 25040:2011 mit den Prozessen der ISO/IEC/IEEE 29119-1:2022 und ISO/IEC/IEEE 29119-2:2021 verbunden. Für die White-Box-Test- und Black-Box-Test-Komponente kommen technische Qualitätsmetriken (siehe Überdeckungsmetriken in Abschnitt 3.2.3) nach ISO/IEC 25023:2016 sowie Richtlinien zur Qualitätssicherung der LLM-Trainingsdaten gemäß ISO/IEC 25024:2015 zum Einsatz.

Der gesamte Testprozess, dargestellt in Abbildung 6.2, basiert auf den in Abschnitt 3 eingeführten Dimensionen des Softwaretestens und gliedert sich in drei aufeinander aufbauende Phasen:

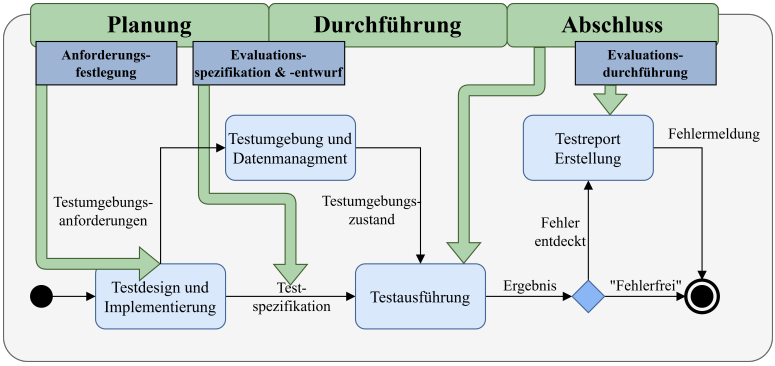



**Abbildung 6.2:** Testprozess basierend auf ISO/IEC 25040:2011 und ISO/IEC/IEEE 29119-2:2021

1. **Planung:** definiert die Anforderungen an die Testumgebung, die Testziele und das Design der Testimplementierung
2. **Durchführung:** führt die eigentlichen Tests aus
3. **Abschluss:** dokumentiert die Ergebnisse systematisch und wertet sie aus

### 6.1.1 Planungsphase

Die Planungsphase gliedert sich, basierend auf den in Abschnitt 3 beschriebenen Grundlagen zu Softwaretests, in drei Schritte: die *Anforderungsfestlegung*, die *Evaluationsspezifikation und den -entwurf* und das *Testdesign und die Implementierung*.

**Anforderungsfestlegung**

Die Anforderungsfestlegung definiert den Zweck und Umfang der Evaluation basierend auf der ISO/IEC 25040:2011 und der ISO/IEC/IEEE 29119-2:2021.

Die Dimensionen des Testobjekts wurden aufgrund technischer und methodischer Anforderungen festgelegt:

- **Software under Test (Zu testende Software, SUT):** Native Android-Applikationen wurden als Testobjekt gewählt, da Android als Open-Source-Plattform umfassende Systemzugriffe und Testmöglichkeiten bietet. Die ausführliche technische Dokumentation und die offene Architektur ermöglichen eine tiefgehende Integration von Testwerkzeugen. Zusätzlich stellen diese Applikationen gemäß Abschnitt 3.3 durch die Heterogenität der Geräte, variable Netzwerkbedingungen und touch-basierte Interaktionen besondere Anforderungen an den Testprozess.
- **Language under Test (Zu testende Programmiersprache, LUT):** Die Wahl fiel auf Java als Programmiersprache, da sie für die native Android-Entwicklung etabliert ist und über ausgereifte Testframeworks wie JUnit und Espresso verfügt (siehe Abschnitt 3.3.3). Dies ermöglicht, zusammen mit der umfangreichen Dokumentation und dem großen Ökosystem an Entwicklungswerkzeugen, eine systematische Integration von Unit- und GUI-Tests. Darüber hinaus ist Java mit 12,3% im verwendeten Trainingsdatensatz stark vertreten[41].

[41] github-code: Datensatz, der zum Training in Abschnitt 7.3.1.2 verwendet wurde (https://huggingface.co/datasets/codeparrot/github-code)

- **Level under Test (Zu testende Testebene, LvlUT):** Die Kombination von Code-Ebene für White-Box-Tests und GUI-Ebene für Black-Box-Tests Tests adressiert die in Abschnitt 3.2.1 beschriebene Notwendigkeit einer umfassenden Testüberdeckung. Diese duale Strategie ermöglicht die Erkennung sowohl von Code-basierten Fehlern als auch von Interaktionsproblemen auf Benutzerebene.
- **Component under Test (Zu testende Komponente, CUT):** Die Fokussierung auf Methoden und mehrschrittige GUI-Interaktionen basiert auf den in Abschnitt 3.3.1 identifizierten Anforderungen mobiler Applikationen. Diese Komponenten sind besonders relevant für die Qualitätssicherung, da sie sowohl die funktionale Korrektheit als auch die Benutzerinteraktion abdecken.

Diese Festlegungen ermöglichen eine systematische und umfassende Evaluation der SQ unter Berücksichtigung der Charakteristika mobiler Applikationen.

#### Evaluationsspezifikation und -entwurf

Auf Basis der Anforderungsfestlegung erfolgt die Evaluationsspezifikation durch die Auswahl unterschiedlicher Qualitätsmaße und die Definition von Messkriterien. Diese orientieren sich an:

- Technischen Qualitätsmetriken nach ISO/IEC 25023:2016 für White-Box-Tests.
- Datenqualitätsmetriken nach ISO/IEC 25024:2015 für die LLM-Trainingsdaten.

Der Evaluationsentwurf berücksichtigt dabei die in Abschnitt 3.1.2 klassifizierten Fehlerarten und die in Abschnitt 3.1.2.2 beschriebenen Fehlerausprägungen mobiler Applikationen.

**Testdesign und Implementierung**

Das Testdesign umfasst die Generierung konkreter Testfälle unter Berücksichtigung der in Abschnitt 3.2.2 definierten Kriterien.

Die Implementierung erfolgt durch:

1. Statische Prüftechniken (siehe Abschnitt 3.2.1.1) für die Testgenerierung
2. Dynamische Prüftechniken (siehe Abschnitt 3.2.1.1) für die Ausführung
3. Integration von GUI-Tests (siehe Abschnitt 3.3.1) für die Benutzeroberfläche

Parallel dazu wird die Entwicklungsumgebung durch Containerisierung (siehe Abschnitt 3.3.5.4) bereitgestellt.

**Entwicklungsumgebung**

Die containerisierte Entwicklungsumgebung adressiert die in Abschnitt 3.3 diskutierten Herausforderungen mobiler Applikationen, insbesondere die Gerätefragmentierung und die Vielfalt der Betriebssystemversionen.

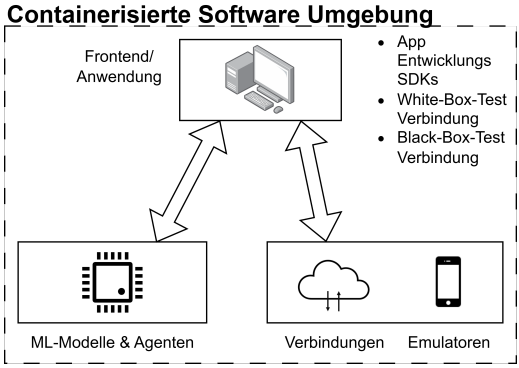



**Abbildung 6.3:** Containerisierte Entwicklungsumgebung mit integrierten Komponenten für White-Box- und Black-Box-Tests

Die in Abbildung 6.3 dargestellte Containerarchitektur gliedert sich in drei Hauptkomponenten.
Das *Frontend* bildet die Anwendungsschicht und stellt die Benutzeroberfläche bereit. Die *ML-Modelle und Agenten* realisieren die KI-gestützte Testgenerierung und -ausführung. Die *Verbindungsschicht* integriert Emulatoren für die Ausführung der mobilen Applikationen. Unterstützend stehen Entwicklungs-SDKs und spezifische Verbindungen für White-Box- und Black-Box-Tests zur Verfügung.

Diese Architektur erfüllt drei zentrale Anforderungen:

1. Gewährleistung einer reproduzierbaren und konsistenten Testumgebung über verschiedene Systeme hinweg gemäß ISO/IEC 25023:2016
2. Isolation der Test-Komponenten von der Host-Umgebung zur konfliktfreien Integration unterschiedlicher Testansätze
3. Vereinfachung der Initialisierung durch vordefinierte Konfigurationen

Die modulare Struktur der containerisierten Umgebung ermöglicht die Integration der in Abschnitt 3.3.5 beschriebenen Testwerkzeuge sowie die flexible Erweiterung um neue Testmethoden. Diese technische Basis unterstützt sowohl die LLM-basierten White-Box-Tests als auch die RL-gestützten Black-Box-Tests und schafft eine erweiterbare Infrastruktur für zukünftige Entwicklungen im Bereich des automatisierten Testens mobiler Applikationen.

### 6.1.2 Durchführungsphase

Die Durchführungsphase implementiert die in der Evaluationsspezifikation definierten Tests und baut direkt auf dem Testdesign auf. Entsprechend der in Abschnitt 3.1.2 definierten Fehlerarten integriert sie White-Box- und Black-Box-Test-Komponenten in einem systematischen Prozess.
In der Durchführungsphase werden die Operationsphasen der KI-Komponenten ausgeführt. Die Operationsphasen verwenden die in den Trainingsphasen trainierten KI-Komponenten zur Testerstellung und -durchführung.

**White-Box-Tests mit LLMs**

Diese Komponente basiert auf den in Abschnitt 3.2.1.2 eingeführten Prinzipien des White-Box-Testens und zielt darauf ab eine möglichst hohe Codeüberdeckung zu erreichen.

Die White-Box-Test-Komponente:

- Führt statische Codeanalysen gemäß Abschnitt 3.2.1.1 durch.
- Generiert Unit-Tests nach den in Abschnitt 3.2.1.4 definierten Prinzipien.
- Prüft die Testfälle auf Erfüllung der in Abschnitt 3.2.2.1 spezifizierten Kriterien:
  - Syntaktische Korrektheit

  - Kompilierbarkeit
  - Ausführbarkeit

- Bewertet die Testüberdeckung anhand der in Abschnitt 3.2.3 eingeführten Metriken:
  - Anweisungsüberdeckung
  - Zweigüberdeckung
  - Pfadüberdeckung

**Black-Box-Tests mit RL**

Diese Komponente implementiert die in Abschnitt 3.2.1.2 beschriebenen Black-Box-Testprinzipien.

Systematische GUI-Tests werden gemäß den in Abschnitt 3.3.1 definierten Anforderungen durch den RL-Agenten durchgeführt. Dieser Ansatz ermöglicht das Testen von mehrschrittigen Interaktionssequenzen und die Identifikation von Problemen auf Benutzeroberflächen-Ebene, die durch reine Code-Analyse möglicherweise unentdeckt blieben.

Die generierten Tests werden mit etablierten Werkzeugen (siehe Abschnitt 3.3.5) ausgeführt und evaluiert.

### 6.1.3 Abschlussphase

Die Abschlussphase des Testframeworks implementiert einen systematischen Evaluationsprozess nach ISO/IEC 25040:2011, der die Ergebnisse beider Testansätze zusammenführt, analysiert und dem Benutzer auf einem Frontend aufbereitet anzeigt. Im Zentrum steht die qualitative und quantitative Auswertung der Testergebnisse anhand der in der Planungsphase definierten Evaluationskriterien.

Für White-Box-Tests erfolgt die Bewertung primär anhand der in Abschnitt 3.2.3 eingeführten Überdeckungsmetriken. Die erreichte Anweisungs- und Zweigüberdeckung wird gemessen und dokumentiert.

Bei Black-Box-Tests fokussiert sich die Evaluation auf die durch den RL-Agenten erreichte Testüberdeckung der Benutzeroberfläche und die Effektivität bei der Aufdeckung von Störungen und Ausfällen.

Der Evaluationsbericht dokumentiert systematisch alle identifizierten Probleme unter Verwendung der in Abschnitt 3.1.2 eingeführten Fehlerbezeichnungen. Für jeden gefundenen Fehler werden die Reproduktionsschritte, der Fehlerkontext sowie mögliche Ursachen und außerdem KI-basierte Lösungsvorschläge erfasst.
Diese detaillierte Dokumentation ermöglicht eine effiziente Fehlerbehebung und unterstützt die kontinuierliche Verbesserung der generierten Tests.

## 6.2 White-Box-Tests

Die Entwicklung der Testgenerierungskomponente basiert auf der Evaluation zweier grundlegender Ansätze. Der erste Ansatz, bezeichnet als *Übersetzungsansatz*, transformiert Quellcode direkt in korrespondierenden Testcode. Der zweite Ansatz, der *eigenständige Ansatz*, befähigt das LLM zur Generierung von Tests durch systematische Analyse der Methodensignaturen, Abhängigkeiten und Programmlogik.

Nach eingehender Evaluation unterschiedlicher Ansätze in mehreren studentischen Abschlussarbeiten (Jacob Hoffmann 2023; Khalil Sakly 2022; Lars Bissinger 2024; Zhi Wang 2024; David Diemand 2024; Tamino Ludwig 2024) erwies sich der eigenständige Ansatz als vorteilhaft gegenüber dem Übersetzungsansatz durch folgende Eigenschaften:

- Der eigenständige Ansatz ermöglicht eine höhere Testqualität durch tieferes Verständnis der Programmlogik.
- Er kann flexibler auf unterschiedliche Codestrukturen reagieren.
- Er bietet bessere Möglichkeiten zur Anpassung der Teststrategien.
- Er erreicht eine höhere Codeüberdeckung.

Die Entscheidung, den Schwerpunkt auf Unit-Tests zu legen, basiert auf der Erkenntnis, dass eine hohe Codeüberdeckung ein wichtiger Indikator für die Qualität und Robustheit einer Software ist (Ivanković u. a. 2024).

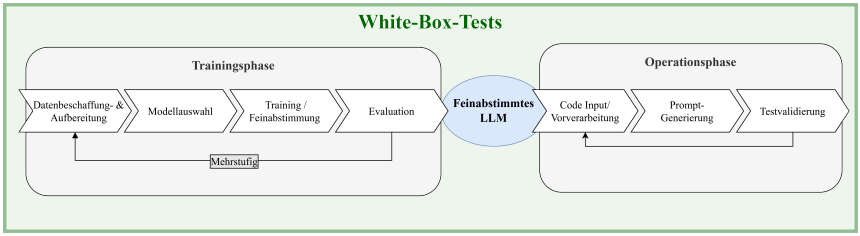



**Abbildung 6.4:** Ablauf der White-Box-Tests

### 6.2.1 Konzept der White-Box-Test Komponente

Die in Abbildung 6.4 dargestellte White-Box-Testkomponente implementiert einen zweiphasigen Ansatz, bestehend aus Trainings- und Operationsphase.
In der Trainingsphase wird das LLM auf die spezifische Aufgabe der Unit-Test-Generierung angepasst.
Die anschließende Operationsphase nutzt das feinabgestimmte Modell zur automatisierten Generierung von Tests für beliebige, im Training nicht verwendete Codebasen.

### 6.2.2 Trainingsphase des LLMs

Die Trainingsphase zielt auf die Entwicklung eines spezialisierten Modells zur Generierung qualitativ hochwertiger Unit-Tests ab. Im Gegensatz zu generalisierten Basismodellen, die aufgrund ihrer Größe erhebliche Rechenressourcen benötigen und auf heterogenen Datensätzen trainiert wurden (siehe Abschnitt 4.4.6.2), wird ein in der Parameteranzahl kleineres Open-Source Modell mit einem fokussierten, hochwertigen Datensatz trainiert. Dieser Ansatz reduziert den Ressourcenbedarf bei gleichzeitiger Verbesserung der Testgenerierungsfähigkeiten.

Die Trainingsphase wird schrittweise durchgeführt:

1. Systematische Datenbeschaffung und -aufbereitung für Java-basierte Softwaretests
2. Selektion eines geeigneten Basismodells unter Berücksichtigung der allgemeinen Fähigkeit Programmcode zu generieren und Ressourceneffizienz
3. Verbesserung des Trainingsprozesses durch Quantisierung und PEFT (siehe Abschnitt 4.4.6.4)
4. Evaluation der Modellleistung anhand definierter Qualitätskriterien
5. Gegebenenfalls Iteration der Schritte 2-4 zur Modellverbesserung

Das Training erfolgt primär mit Java-Code und den dazugehörigen Unit-Tests, wobei die Architektur eine spätere Erweiterung auf weitere Programmiersprachen wie Kotlin oder Python ermöglicht.
Der Prozess baut auf den Erkenntnissen von J. Hoffmann und Frister 2024 auf und implementiert fortgeschrittene Strategien zur effizienten Modellverbesserung.

#### 6.2.2.1 Datenbeschaffungsprozess

Der Datenbeschaffungsprozess orientiert sich an den Hauptphasen des CRISP-DM-Zyklus (siehe Abschnitt 4.4.1) und des Data Life Cycle (DLC) nach ISO/IEC 25024:2015, sowie dem erweiterten SQuaRE Datenqualitätsmodell für KI-Systeme nach Nakajima und Nakatani (2021).

**Geschäftsverständnis und Anforderungsanalyse** In dieser ersten Phase werden die Ziele der LLM-basierten Testgenerierung für mobile Applikationen definiert und die Qualitätsanforderungen gemäß ISO/IEC 25024:2015 festgelegt. Der Fokus liegt auf nativen Android-Anwendungen, wobei die Architektur die Flexibilität für zukünftige Erweiterungen berücksichtigt. Die Qualitätsanforderungen basieren auf den von Nakajima und Nakatani (2021) definierten KI-spezifischen Erweiterungen des SQuaRE-Frameworks.

**Datenverständnis und Datenbeschaffung** Als primäre Datenquelle dient GitHub[42], was den Zugriff auf eine Vielzahl von Codebeispielen und zugehörigen Tests ermöglicht. Die Qualität der Trainingsdaten wird durch drei zentrale Eignungskriterien bestimmt (Nakajima und Nakatani 2021):

- **Zielauswahleignung (Target Selection Adequacy):** Beschreibt die Angemessenheit der gewählten Merkmalsdimensionen für die Testgenerierung unter Vermeidung von ungewollten Informationslecks.
- **Datenauswahleignung (Data Selection Adequacy):** Umfasst die gezielte Selektion relevanter Codebeispiele und Tests unter Berücksichtigung von Ausreißern und Spezialfällen.
- **Stichprobeneignung (Sampling Adequacy):** Sichert die repräsentative Verteilung der Trainingsdaten zur Minimierung von Verteilungsverschiebungen (Distribution Shift) im praktischen Einsatz.

Diese Eignungskriterien bilden die Grundlage für die systematische Qualitätssicherung der Trainingsdaten im entwickelten Framework. **Datenvorbereitung und Qualitätssicherung** Diese Phase implementiert einen mehrstufigen Ansatz zur Sicherstellung einer hohen Datenqualität unter Berücksichtigung der spezifischen Anforderungen an KI-Trainingsdaten:

- **Vorfilterung:** Die Datensätze werden nach folgenden Qualitätsmerkmalen gefiltert (Nakajima und Nakatani 2021):
  - Accuracy (semantische Korrektheit der Daten)
  - Completeness (Vollständigkeit der Datensätze)
  - Consistency (Kohärenz der Kennzeichnungen)
  - Credibility (Authentizität der Daten)

[42] GitHub ist eine webbasierte Plattform für Versionskontrolle und kollaborative Softwareentwicklung (`https://github.com`)

  - Currentness (Aktualität der Daten)

- **Präzises Code-Test-Matching:** Entwicklung von Heuristiken für Pfad- und Namens-Matching, Analyse von Import-Statements und kontextuelle Analyse mittels LLM-Techniken unter Berücksichtigung der Target Selection Adequacy nach Nakajima und Nakatani (2021).
- **Mehrstufiger Filterprozess durch Ensemble-Ansatz:** Der Ensemble-Ansatz, wie in (Alshahwan u. a. 2024) beschrieben, kombiniert die Ausgaben mehrerer LLMs, um die Qualität der generierten Tests zu verbessern. Dieser Ansatz nutzt die Diversität der von verschiedenen Modellen generierten Tests und implementiert die Data Selection Adequacy nach Nakajima und Nakatani (2021).
- **Datensatzerweiterung:** Erweiterung des Datensatzes durch ein auf (Abdin u. a. 2024) basierendes Konzept zur Generierung synthetischer Daten unter Berücksichtigung der Sampling Adequacy zur erhöhten Repräsentation (Nakajima und Nakatani 2021).

Darüber hinaus werden auch die in Abschnitt 4.4.2.3 beschriebenen Methoden verwendet, um die Datenqualität weiter zu verbessern.

**Modellauswahl und -vorbereitung** In dieser Phase erfolgt die Auswahl und Vorbereitung eines geeigneten Basis-Modells. Dies beinhaltet:

- Vergleich verschiedener Modellarchitekturen unter Berücksichtigung von Größe, Vortraining, Leistung in relevanten Benchmarks (beispielsweise HumanEval (siehe Abschnitt 4.4.9.3)), Adaptierbarkeit und Ressourcenanforderungen. Evaluation der Modellarchitektur unter Berücksichtigung der in Abschnitt 4.6.3 definierten Anforderungen und Limitationen.
- Verwendung von Strategien zur Verbesserung des Trainingsprozesses, einschließlich Quantisierung und Parameter-Efficient Fine-Tuning.

- Dokumentation der Provenance aller Modellentscheidungen und Datenverarbeitungsschritte zur Gewährleistung der Nachvollziehbarkeit (Nakajima und Nakatani 2021).

#### 6.2.2.2 Feinabstimmung des LLMs

Nach der Datenbeschaffung und -evaluation erfolgt die Feinabstimmung des ausgewählten Basismodells zur Verbesserung der Unit-Test-Generierung. Der Prozess implementiert Parameter-effiziente Trainingsmethoden und quantisierte Modelle (siehe Abschnitt 4.4.5). Diese Techniken reduzieren die Ressourcenanforderungen, wie beispielsweise der Speicherbedarf bei gleichzeitiger Beibehaltung der Modellqualität.

Der Feinabstimmungsprozess gliedert sich in vier Schritte:

1. **Modellvorbereitung:** Die Vorbereitung des Modells erfolgt durch 4-bit Quantisierung und Gradient Checkpointing.

2. **Parameter-effizientes Training:** Die Feinabstimmung nutzt die LoRA-Technik, die nur einen kleinen Teil der Modellparameter anpasst. Durch die Definition spezifischer Hyperparameter wie *lora_alpha*, *lora_dropout* und Rang wird die Effizienz des Trainings gesteuert (siehe Abschnitt 4.4.5).

3. **Iteratives Training:** Das Training erfolgt in definierten Schritten mit regelmäßiger Evaluation. Wichtige Parameter wie Lernrate, Gewichtung des Gradientenabstiegs und Aufwärmphase werden systematisch angepasst. Die Trainingsfortschritte werden überwacht und protokolliert.

4. **Modellevaluation:** Die Evaluation des Modells erfolgt anhand der Metrik PPL auf Trainings- und Validierungsdaten (siehe Abschnitt 4.4.9.1). Diese Metrik quantifiziert die Vorhersagequalität des Modells bei der Generierung von Unit-Tests.

### 6.2.3 Operationsphase: Code-Analyse und Testgenerierung

In der Operationsphase wird das feinabgestimmte LLM eingesetzt, um Unit-Tests für neuen, ungesehenen Code zu generieren. Dieser Prozess nutzt die im LLM gespeicherten Informationen über Codestrukturen, Testmuster und die Programmiersprache, um effektive Tests zu erstellen.

#### 6.2.3.1 Prozessablauf

Der Ablauf der Operationsphase gliedert sich in vier Schritte:

**Vorverarbeitung: Code-Analyse und Informationsextraktion** In der Vorverarbeitung wird der Quellcode der zu testenden Anwendung analysiert. Je nach gewählter Strategie können ganze Klassen, mehrere Klassen oder einzelne Methoden als Eingabe dienen. Die Extraktion relevanter Informationen umfasst:

- **Methodensignaturen:** Name, Parametertypen und Rückgabetyp.
- **Dokumentation:** Kommentare zur Funktionsweise der Methode.
- **Abhängigkeiten:** Verknüpfungen zu anderen Klassen oder Methoden.

Die Methoden zur Extraktion dieser Informationen basieren auf den in Abschnitt 3.2.1.2 beschriebenen White-Box-Testprinzipien.

**Prompt-Generierung** Basierend auf den extrahierten Informationen wird ein Prompt für das LLM generiert. Dieser Prompt nutzt ein Template und strukturiert die Informationen in einem für das Modell verständlichen Format, wobei Methodensignaturen, Abhängigkeiten und Dokumentation in einer definierten Reihenfolge präsentiert werden.
Die Prompt-Generierung berücksichtigt die in Abschnitt 4.4.8 diskutierten Techniken, wie Chain-of-Thought und Step-Back-Prompting, um dem Modell eine systematische Analyse und Testgenerierung zu ermöglichen.

**Test-Generierung durch das LLM** Das LLM verarbeitet den Prompt und erzeugt einen Unit-Test als Text, wobei es das während des Trainings erworbene Wissen über Testmuster und Best Practices anwendet. Der generierte Test wird in einem standardisierten Format ausgegeben, das die notwendigen Importanweisungen, Testklassendeklarationen, Setup-Methoden und eigentlichen Testfälle enthält.

**Test-Integration und -Ausführung** Der generierte Unit-Test wird in die bestehende Test-Suite integriert und ausgeführt.

Die Ergebnisse der Testausführung werden gesammelt und für die weitere Analyse aufbereitet. Dieser Schritt umfasst auch mehrere automatische Validierungsschritte der generierten Tests, um sicherzustellen, dass sie syntaktisch korrekt, kompilierbar und ausführbar sind.

#### 6.2.3.2 Validierungs- und Verbesserungsstrategien

Um die Effizienz und Genauigkeit der Testgenerierung weiter zu verbessern, werden folgende Strategien eingesetzt:

**Validierung der Testergebnisse** In einem ersten Schritt werden die generierten Tests auf syntaktische Korrektheit mithilfe klassischer Parsing-Techniken überprüft. Dabei wird sichergestellt, dass die Tests syntaktisch korrekt sind und keine offensichtlichen Fehler enthalten.
Anschließend werden die Tests kompiliert und ausgeführt, um sicherzustellen, dass sie kompilierbar und ausführbar sind.

**Iterative Verbesserung**

Fehlerhafte (nicht kompilierbare) Testergebnisse werden als Feedback an das LLM zurückgeführt, um die Tests erneut zu generieren. Dies geschieht durch Übergabe der Testausführungsergebnisse und der Compilerlogs und Fehlermeldungen.

**Fortgeschrittene Prompt-Techniken** Die Implementierung fortgeschrittener Prompting-Techniken erfordert eine initiale Konfiguration zur Definition geeigneter Prompt-Templates und Validierungsregeln. Nach dieser einmaligen Konfiguration erfolgt die Testgenerierung automatisiert.
Die eingesetzten Techniken, beschrieben in Abschnitt 4.4.8, umfassen:

- Chain-of-Thought-Prompting für mehrschrittige Testlogik, das dem Modell ermöglicht, Zwischenschritte bei der Testgenerierung zu berücksichtigen.
- Take-a-step-back-Prompting für die systematische Analyse der Testanforderungen.
- Iteratives Prompting zur schrittweisen Verfeinerung der Tests mit dokumentierter Fehleranalyse und Verbesserungshistorie.

Die Nachvollziehbarkeit der Testergebnisse wird durch automatisierte Protokollierung aller Generierungsschritte und Entscheidungen sichergestellt. Die generierten Unit-Tests werden anhand objektiver Metriken zur Testüberdeckung evaluiert, wie in Abschnitt 3.2.3 beschrieben. Diese systematische Dokumentation ermöglicht die Analyse und Validierung der Testergebnisse durch Entwickler.

## 6.3 Black-Box-Tests

Die Black-Box-Test-Komponente des Frameworks implementiert einen Reinforcement Learning (Bestärkendes Lernen, RL)-basierten Ansatz zum automatisierten Testen von mobilen Applikationen, mit Fokus auf Android-Anwendungen.

Die Architektur orientiert sich an den Qualitätsmerkmalen der ISO/IEC 25010:2023 und adressiert die in Abschnitt 3.3 diskutierten spezifischen Anforderungen des mobilen App-Testens. Die Integration der in Abschnitt 3.3.1 beschriebenen GUI-Testprinzipien mit RL-Methoden ermöglicht es dem RL-Agenten, effektive Interaktionssequenzen zu generieren.

Die modulare Struktur unterstützt die systematische Exploration der Anwendungsoberfläche sowie die Identifikation von Fehlerzuständen durch den Vergleich aufeinanderfolgender Bildschirmzustände. Diese Implementierung basiert unter anderem auf Erkenntnissen aus der studentischen Abschlussarbeit von Can Pigur (Pigur und Meyjohann 2024).

### 6.3.1 Konzept der Black-Box-Test-Komponente

Die Black-Box-Test-Komponente basiert auf einem RL-Agenten, der durch systematische Interaktion mit der Benutzeroberfläche einer Android-Applikation Testfälle generiert und ausführt. Die Implementierung erfolgt in einer containerisierten Testumgebung. Der Agent agiert ausschließlich über die Benutzeroberfläche mit der zu testenden Applikation in einem Android-Emulator (DUT), wodurch die in Abschnitt 3.2.1.2 definierten Prinzipien des Black-Box-Testens eingehalten werden.

Die Testumgebung implementiert die MDP-Prinzipien (siehe Abschnitt 4.5) durch eine systematische Erfassung von Zuständen, Aktionen und Belohnungen. Die Zustände werden durch Bildschirmaufnahmen und Gerätedaten repräsentiert, während die Aktionen durch standardisierte GUI-Interaktionen definiert sind. Das Belohnungssystem quantifiziert den Erfolg der Aktionen basierend auf Zustandsänderungen und Testzielen.

Die Containerisierung nach ISO/IEC/IEEE 29119-2:2021 gewährleistet reproduzierbare Testbedingungen durch:

- Standardisierte Android-Emulator-Konfigurationen mit definierten Gerätespezifikationen.
- Versionskontrolle der Testumgebung und Abhängigkeiten.
- Automatisierte Umgebungsinitialisierung mit deterministischen Ausgangszuständen.

Durch diese standardisierte Umgebung wird eine konsistente Testausführung über verschiedene Systeme hinweg gewährleistet und eine spätere Integration in CI/CD-Pipelines ermöglicht.
Die Architektur gliedert sich in vier zentrale Komponenten:

- **Feature-Extraktion:** Ein spezialisiertes Beobachtungssystem erfasst und analysiert den aktuellen Systemzustand anhand des Bildschirminhalts (RGB-Bild), zeitlicher Abstände zwischen Interaktionen und des Gerätezustands. Die Implementierung nutzt DL-Architekturen zur Erkennung und semantischen Analyse von GUI-Elementen. Durch Transfer-Learning-Techniken und vortrainierte Modelle (wie beispielsweise YOLO) wird das sogenannte „Cold-Start-Problem“ des RL-Agenten adressiert (LeCun, Bengio und G. Hinton 2015). Das Cold-Start-Problem beschreibt die Herausforderung, dass ein RL Agent, der ohne jegliche Vorkenntnisse startet, eine längere Trainingszeit benötigt, um effektive Aktionen zu erlernen.

- **Aktionsmodellierung:** Die Interaktion mit der Benutzeroberfläche erfolgt durch ein hybrides Aktionsmodell. Dieses kombiniert diskrete Aktionstypen (*LIFT*, *TOUCH*, *REPEAT*) mit kontinuierlichen Koordinaten im normalisierten Bildschirmraum, wie es von der AndroidEnv ermöglicht wird (Toyama u. a. 2021). Die Normalisierung ermöglicht die Übertragbarkeit der Aktionen zwischen verschiedenen Bildschirmgrößen und Auflösungen.
- **Belohnungssystem:** Ein mehrschichtiges Bewertungssystem quantifiziert die Qualität der ausgeführten Aktionen. Die Bewertung basiert auf der Exploration neuer Bildschirmzustände und erfolgreicher GUI-Interaktionen. Bildbasierte Ähnlichkeitsmetriken und Distanzmaße ermöglichen die objektive Bewertung von Zustandsübergängen.
- **Systemintegration:** Standardisierte Wrapper-Module ermöglichen die flexible Integration verschiedener RL-Algorithmen und gewährleisten die Kompatibilität zwischen den Komponenten. Diese modulare Struktur unterstützt die Erweiterbarkeit des Systems und die Integration neuer Teststrategien.

Die vorgestellte Architektur ermöglicht eine systematische Exploration der Anwendungsoberfläche und die automatische Identifikation von Fehlerzuständen durch den kontinuierlichen Vergleich aufeinanderfolgender Bildschirmzustände.

### 6.3.2 Trainingsphase

Die Trainingsphase entwickelt einen spezialisierten RL-Agenten für die Generierung und Ausführung effektiver GUI-Interaktionen. Die Implementation orientiert sich an den in Abschnitt 4.5 beschriebenen RL-Prinzipien und nutzt AndroidEnv (siehe Abschnitt 4.6.1) als standardisierte Trainingsumgebung.

Die Trainingsphase ermöglicht die Nutzung verschiedener RL-Algorithmen wie A2C, PPO oder SAC (siehe Abschnitt 4.5.1), deren Auswahl auf der Leistungsfähigkeit bei der Generierung effektiver Testsequenzen und der Anpassungsfähigkeit an verschiedene App-Strukturen basiert.
Ein hybrides Belohnungssystem fördert die systematische Exploration der Benutzeroberfläche durch die Kombination extrinsischer und intrinsischer Belohnungen. Extrinsische Belohnungen quantifizieren den Erfolg spezifischer GUI-Interaktionen, während intrinsische Belohnungen durch Curiosity-driven Learning (siehe Abschnitt 5.2) die Exploration des Zustandsraums fördern.

Transfer-Learning-Techniken mit vortrainierten Modellen adressieren das Cold-Start-Problem bei der GUI-Elementeerkennung. Die kontinuierliche Anpassung der Trainingsparameter erfolgt durch Hyperparameter-Anpassung (siehe Abschnitt 4.5.2).

**Ähnlichkeitsmessungen zwischen Oberflächen**

Die Ähnlichkeitsmessung zwischen GUI-Zuständen bildet ein zentrales Element der Trainingsphase (siehe Abschnitt 4.6.2). Das System implementiert bildbasierte Metriken zur Quantifizierung der Zustandsübergänge, die dem Agenten eine effektive Bewertung seiner Aktionen ermöglichen und seine Neugier fördern.

### 6.3.3 Operationsphase

Die Operationsphase implementiert den in ISO/IEC 25040:2011 definierten Evaluationsprozess für die automatisierte GUI-Testausführung durch den trainierten RL-Agenten.

Der systematische Prozess umfasst:

1. **Initialisierung:** Neuinstallation der zu testenden Applikation in der AndroidEnv-Umgebung zur Gewährleistung eines definierten Ausgangszustands.
2. **Zustandserfassung:** Kontinuierliche Analyse des Bildschirminhalts und der GUI-Elemente mittels der trainierten Feature-Extraktionskomponente.
3. **Aktionsauswahl:** Auswahl der nächsten GUI-Interaktion basierend auf der gelernten Strategie.
4. **Aktionsausführung:** Ausführung der ausgewählten Aktion über die ADB-Schnittstelle (siehe Abschnitt 2.1.1) mit anschließender Validierung.
5. **Ergebnisanalyse:** Bewertung der Aktionsauswirkungen durch Analyse der Zustandsübergänge und Erkennung potentieller Fehlerzustände mittels Log-Analyse.
6. **Berichterstattung:** Systematische Dokumentation identifizierter Fehler, kritischer Zustände und den vom Agenten ausgewählten Interaktionssequenzen.

Der gesamte Prozess wird durch ein zentrales Monitoring-System überwacht. Die Ergebnisse werden über die in Abschnitt 7.2 beschriebene Frontend-Schnittstelle visualisiert.

# 7 Implementierung des Testframeworks

Das Kapitel beschreibt die Implementierung des in Kapitel 6 vorgestellten automatisierten Testframeworks für mobile Applikationen.

Entsprechend der Architektur gliedert sich die Implementierung in die Entwicklungs- und Testumgebung sowie die Trainings- und Operationsphasen der White-Box-Test- und Black-Box-Test-Komponenten und dem Frontend, in dem die Ergebnisse angezeigt werden.

Der gesamte Code des Testframeworks ist im KIT-GitLab verfügbar[43].

[43] GitLab-Repository des Testframeworks: `https://gitlab.kit.edu/kit/aifb/BIS/kit-bis/mobileappkitestplattform`

## 7.1 Entwicklungsumgebung und Containerisierung

Die Implementierung des Testframeworks basiert auf einer containerisierten Entwicklungsumgebung entsprechend der in Abschnitt 3.3.5.4 und 6.1.1 beschriebenen Architektur. Als Basis dient ein TensorFlow GPU-Image[44], das um die erforderlichen Komponenten für White-Box- und Black-Box-Tests erweitert wurde.

Die Containerkonfiguration wurde durch ein mehrstufiges Dockerfile realisiert und wird durch eine devcontainer.json-Konfiguration für die Visual Studio Code Integration ergänzt. Diese Struktur ermöglicht eine reproduzierbare und isolierte Testumgebung.

### 7.1.1 Komponenten der Testumgebung

Die **White-Box-Test-Komponente** implementiert die für Unit-Tests erforderliche Java-Entwicklungsumgebung. Diese umfasst das Microsoft OpenJDK 21[45] für die Programmausführung sowie Gradle 7.5.1[46] und Maven 3.9.9[47] für Build-Management und Dependency-Verwaltung.
JUnit wird für die Testausführung entsprechend der in Abschnitt 3.2.2.2 definierten Anforderungen eingesetzt.

[44] TensorFlow GPU-Image: Ein Docker-Container mit vorinstalliertem TensorFlow und GPU-Unterstützung (`https://hub.docker.com/r/tensorflow/tensorflow`)
[45] OpenJDK: Eine Open-Source-Implementierung der Java-Entwicklungsplattform (`https://www.microsoft.com/openjdk`)
[46] Gradle: Build-Automatisierungstool (`https://gradle.org`)
[47] Apache Maven: Software-Projektmanagement-Tool (`https://maven.apache.org`)

Die **Black-Box-Test-Komponente** integriert die für maschinelles Lernen benötigten Frameworks. TensorFlow 2.18.0 mit GPU-Unterstützung bildet die Grundlage für die Modellimplementierung. Stable-Baselines3 2.1.0 stellt gemäß Abschnitt 4.5.1.4 die RL-Algorithmen bereit. AndroidEnv implementiert die in Abschnitt 4.6.1 beschriebene Verknüpfung mit der Android-Emulation. OpenCV[48] und FAISS[49] ermöglichen Bildverarbeitung und Ähnlichkeitsanalysen. Das Tracking der Erfolgsmetriken der ML-Algorithmen erfolgt über Weights & Biases[50].

Die **Android-Entwicklungsumgebung** stellt das Android SDK[51] mit API Level 34 bereit. Die Integration von Platform Tools und Command Line Tools ermöglicht die Kommunikation mit Android-Geräten über ADB[52]. System Images für x86_64-Architekturen mit Google APIs unterstützen die Emulation von Android-Geräten.

### 7.1.2 Konfiguration und Ausführung

Die Python-3.11-Umgebung wird über eine requirements.txt konfiguriert. Diese definiert neben den Machine-Learning-Frameworks auch Flask[53] für das Frontend sowie Bibliotheken für Code-Analyse und Testauswertung.

[48] OpenCV: Open Source Computer Vision Library (`https://opencv.org`)
[49] FAISS: Facebook AI Similarity Search (`https://github.com/facebookresearch/faiss`)
[50] Weights & Biases: MLOps-Platform für ML-Tracking (`https://wandb.ai`)
[51] Android SDK: Software Development Kit für Android-Entwicklung (`https://developer.android.com/studio`)
[52] Android Debug Bridge: Kommandozeilen-Tool zur Kommunikation mit Android-Geräten (`https://developer.android.com/tools/adb`)
[53] Flask: Python Web Framework (`https://flask.palletsprojects.com`)

Die Entwicklungsumgebung wird durch vordefinierte Umgebungsvariablen konfiguriert und nutzt ein nicht-privilegiertes Benutzerkonto. Die Integration erfolgt über Visual Studio Code mit vordefinierten Erweiterungen für Python, Java und Android-Entwicklung. Der Container startet mit GPU-Unterstützung und öffnet Port 5037 für ADB-Verbindungen.

Die finale Versionsauswahl der Komponenten erfolgte durch iteratives Testen verschiedener Versionskombinationen. Der Fokus lag dabei auf der grundlegenden Kompatibilität zwischen den Machine-Learning-Frameworks, Android-Tools und der Entwicklungsumgebung. Getestet wurden insbesondere die GPU-Unterstützung, die ADB-Kommunikation sowie die korrekte Interaktion der ML-Frameworks. Aus den funktionsfähigen Kombinationen wurde eine stabile Version für die Implementierung ausgewählt.

## 7.2 Frontend-Komponente

Die Frontend-Implementierung basiert auf einer Flask-Webanwendung, die als zentrale Schnittstelle zwischen Benutzern und den KI-Komponenten dient. Teile des Frontends wurden im Rahmen eines Praktikums mit dem Titel „Entwicklung einer Testplattform für mobile Applikationen“ mit Studierenden umgesetzt (Bissinger u. a. 2024).
Die Architektur folgt dem Model-View-Controller (MVC) Muster (Reenskaug 1979), wodurch eine klare Trennung der Zuständigkeiten und eine wartbare Codestruktur gewährleistet wird.

### 7.2.1 Benutzerverwaltung und Datenpersistenz

Beim ersten Aufruf der Anwendung landen Benutzer auf einer Übersichtsseite, die das Testframework beschreibt.

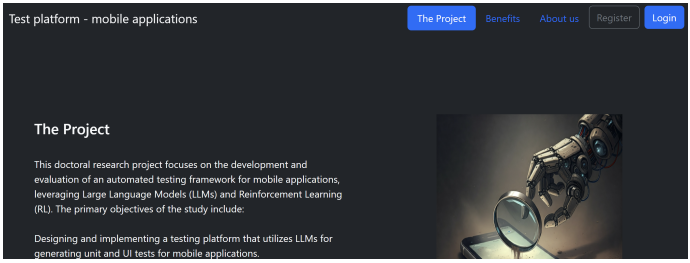



**Abbildung 7.1:** Hauptseite des Testframeworks

Die Authentifizierung wird über eine Authentifizierungsschnittstelle realisiert, die eine Benutzerregistrierung, Anmeldung und Passwortverwaltung implementiert. Benutzer können sich mit Benutzername und Passwort registrieren, wobei die Passwörter gehasht gespeichert werden. Die Authentifizierung verwendet Flask-Login für die Sitzungsverwaltung.

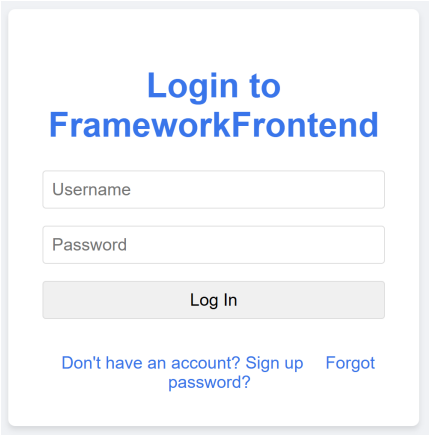



**Abbildung 7.2:** Registrierungsmaske des Testframeworks

Die Persistenz der Daten wird durch eine SQLite3-Datenbank[54] realisiert, die über SQLAlchemy ORM[55] angesprochen wird. Diese Architektur ermöglicht eine objektorientierte Verwaltung der Anmeldedaten, Benutzerinformationen, Repositories und Testergebnisse.

[54] SQLite3: Eine leichtgewichtige, dateibasierte relationale Datenbank, die ohne separaten Server auskommt (`https://www.sqlite.org`)

[55] SQLAlchemy ORM: Eine objekt-relationales Mapping-Bibliothek (ORM) für Python, die eine flexible und performante Abstraktionsebene für den Zugriff auf relationale Datenbanken bietet (`https://www.sqlalchemy.org`)

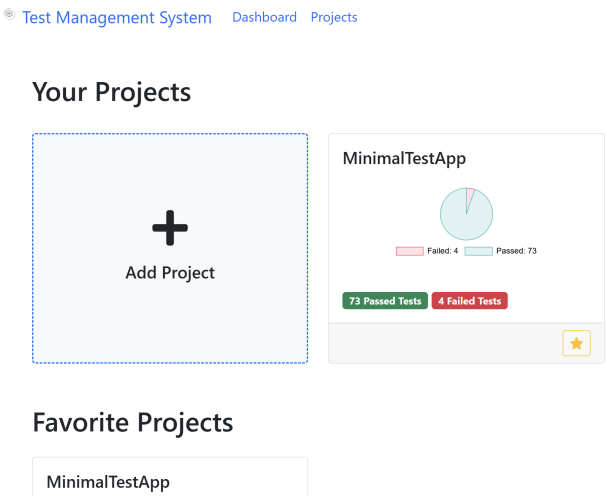



**Abbildung 7.3:** Dashboard des Testframeworks

Nach der Anmeldung ermöglicht ein Dashboard den Zugriff auf die getesteten Repositories und deren Ergebnisse. Benutzer können GitHub- oder GitLab-Repositories über deren URL und optional einen Zugangstoken einbinden. Das System klont diese Repositories in benutzerspezifische Verzeichnisse und macht sie für Tests verfügbar. Ein Favorisierungssystem ermöglicht das Markieren häufig getesteter Projekte für schnelleren Zugriff.

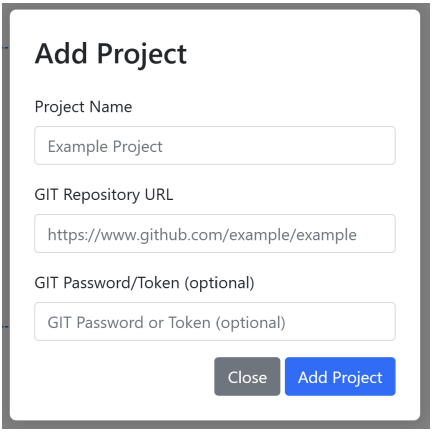



**Abbildung 7.4:** Repository-Download im Testframework

### 7.2.2 Echtzeit-Testausführung

Die Template-Hierarchie unterstützt modulare UI-Komponenten mit einem Basis-Template für gemeinsame Elemente. Das Dashboard-Template präsentiert eine Übersicht aller verfügbaren Repositories und deren Teststatus, während spezialisierte Ergebnis-Templates detaillierte Testresultate visualisieren.

Die Darstellung der Testergebnisse erfolgt in Echtzeit über Server-Sent Events, die über eine dedizierte Event-Stream-Verbindung den aktuellen Fortschritt der Testausführung kontinuierlich an das Frontend übermitteln.

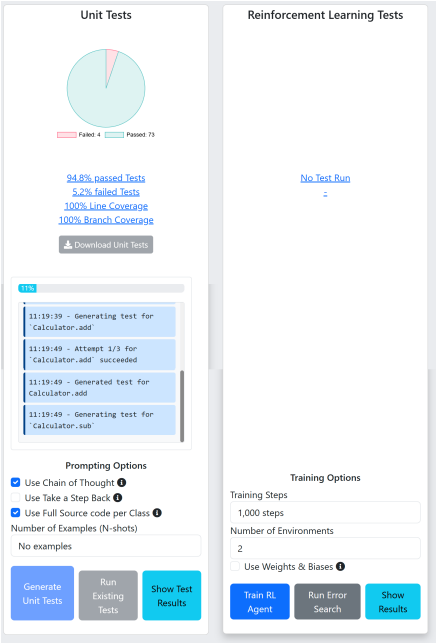



**Abbildung 7.5:** Repository-Übersicht im Testframework

Dazu wird das EventSource-Interface[56] des Browsers genutzt, das eine unidirektionale Verbindung zum Server aufbaut. Der Server sendet dabei formatierte Ereignisnachrichten im JSON-Format, die Informationen über den Generierungsfortschritt, die Anzahl erstellter Tests und eventuelle Fehlermeldungen enthalten.

Ein Fortschrittsindikator visualisiert diese Updates und zeigt den aktuellen Status der Testgenerierung und -ausführung an. Eventuelle Fehlermeldungen oder Warnungen werden unmittelbar farblich codiert und in der Benutzeroberfläche angezeigt.

[56] EventSource-Interface: Eine API, die es ermöglicht, Server-Sent Events (SSE) zu empfangen, indem eine unidirektionale Verbindung zum Server hergestellt wird (`https://html.spec.whatwg.org/multipage/server-sent-events.html`)

### 7.2.3 Test-Komponenten

Im Kern der Implementierung steht die `repository`-Klasse, die die primäre Schnittstelle für die Verarbeitung von HTTP-Anfragen zu den ML-Komponenten, der Oberfläche und dem zu testenden Repository bildet. Dies wird in Flask durch die Verwendung von Routen realisiert, die HTTP-Anfragen an spezifische Endpunkte weiterleiten. Es definiert zwei zentrale Routen für White-Box-Tests mittels LLM und Black-Box-Tests mittels RL. Diese Routen sind über dedizierte Callback-Handler mit den jeweiligen Test-Komponenten verbunden.

Die **White-Box-Test-Komponente** ermöglicht die Konfiguration, Erstellung und Ausführung von Unit-Tests über eine intuitive Benutzeroberfläche. Über Checkboxen können verschiedene Testparameter angepasst werden, darunter die Verwendung von Chain-of-Thought Prompting, Take-a-Step-Back Prompting sowie die Auswahl zwischen der Analyse kompletter Klassen oder nur der einzelnen Methoden.

Die Testergebnisse werden durch ein Kreisdiagramm visualisiert, das mittels Chart.js[57] implementiert ist. Das Diagramm stellt die Verteilung erfolgreicher und fehlgeschlagener Tests dar und ermöglicht eine interaktive Exploration der Daten durch Hover-Effekte und Tooltips. Ergänzend werden quantitative Metriken wie die Zeilenüberdeckung und Zweigüberdeckung in einem weiteren Popup angezeigt (siehe Abschnitt 3.2.3).

[57] Chart.js: Eine JavaScript-Bibliothek zur Erstellung responsiver und interaktiver Datenvisualisierungen (`https://www.chartjs.org`)

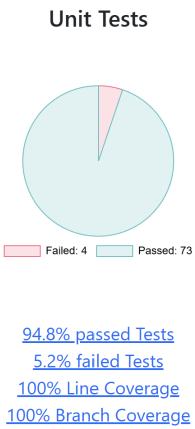



**Abbildung 7.6:** Kreisdiagramm der Testergebnisse im Testframework

Die generierten Unit-Tests können als ZIP-Archiv heruntergeladen und bei Bedarf auch in externen Entwicklungsumgebungen ausgeführt werden. Für fehlgeschlagene Tests werden detaillierte Stacktraces, Fehlerbeschreibungen und KI-generierte Lösungsvorschläge bereitgestellt.

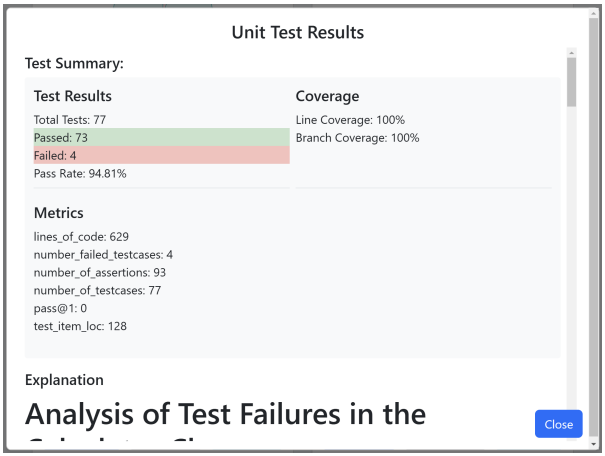



**Abbildung 7.7:** Popup mit Testmetriken im Testframework

Die **Black-Box-Test-Komponente** bietet eine visuelle Schnittstelle zur Konfiguration und Überwachung des RL-basierten GUI-Testings. Der Testfortschritt wird in Echtzeit visualisiert, wobei sowohl der aktuelle Trainingsstand des RL-Agenten als auch die Ergebnisse der Testausführung dargestellt werden. Die Visualisierung umfasst Metriken wie die Anzahl der durchgeführten Trainingsepochen und der aktuellen Belohnung.

Bei der Testausführung werden die durchgeführten Aktionen und die daraus resultierenden GUI-Interaktionen in Screenshots gespeichert. Dies ermöglicht eine Einschätzung des aktuellen Trainingszustands des Agenten und eine nachträgliche Analyse der Testdurchführung.

Die Integration der Test-Komponenten erfolgt über eine standardisierte Schnittstelle, die asynchrone Testausführung unterstützt und Echtzeit-Updates ermöglicht. Ein Fehlermanagement gewährleistet die Isolation und angemessene Behandlung von Fehlern der Oberfläche oder der Test-Komponenten. Diese Implementierung stellt eine erweiterbare Frontend-Lösung dar, die eine Brücke zwischen Benutzerinteraktion und den LLM- sowie RL-Testfunktionalitäten schlägt. Die konsequente Trennung von Zuständigkeiten und die Fehlerbehandlung gewährleisten eine zuverlässige Testausführung und Visualisierung der Ergebnisse.

## 7.3 White-Box-Test-Komponente

Die White-Box-Test-Komponente folgt der in Abschnitt 6.2 beschriebenen zweiphasigen Architektur. Die erste Phase umfasst die Datenbeschaffung und das Training des LLM für die Testgenerierung, während die zweite Phase die operative Nutzung des trainierten Modells zur Generierung und Ausführung von Unit-Tests realisiert. Abbildung 7.8 zeigt die Implementierung der White-Box-Test-Komponente, die in Trainings- (oben) und Operationsphase (unten) unterteilt ist.

Die Operationsphase basiert auf der statischen Analyse des Quellcodes und der anschließenden automatisierten Generierung von Unit-Tests durch das LLM.

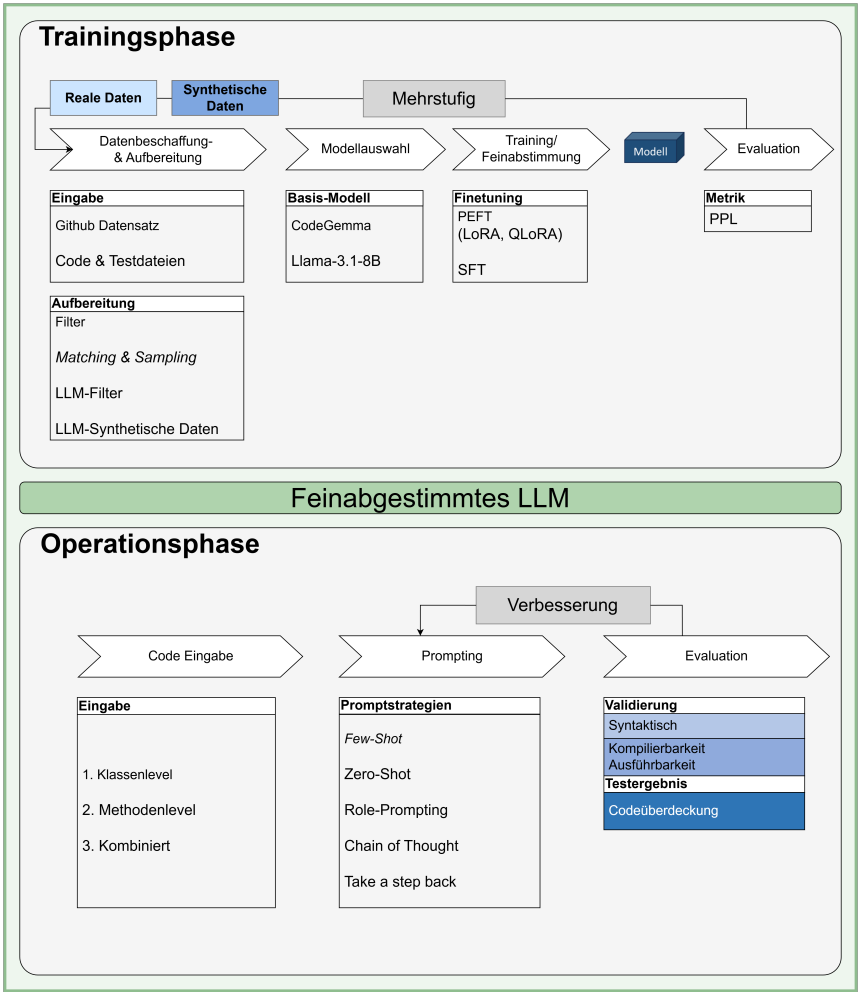



**Abbildung 7.8:** Implementierung der White-Box-Test-Komponente aufgeteilt in Traings- und Operationsphase

Die folgenden Abschnitte beschreiben die Implementierung der Trainings- und Operationsphase, einschließlich der Datenbeschaffung, des Modelltrainings sowie der Prompt-Engineering-Strategien für die Testgenerierung.

### 7.3.1 Trainingsphase

Die LLM-Trainingsphase basiert auf einem mehrstufigen Prozess, der die Beschaffung, Aufbereitung und Qualitätssicherung der Trainingsdaten sowie das eigentliche Training des Modells umfasst. Die initiale Version der Datenbeschaffung wurde im Rahmen einer Bachelorarbeit entwickelt (Jacob Hoffmann 2023) und in (J. Hoffmann und Frister 2024) weiterentwickelt.

Die anschließende Zusammenarbeit mit dem Bachelorarbeiter als wissenschaftliche Hilfskraft führte zur Implementierung einer spezialisierten Feinabstimmungspipeline für die LLM-basierte Testgenerierung. Diese umfasst die systematische Datenbeschaffung durch Filterung und Aufbereitung von Code-Test-Paaren, die Integration eines mehrstufigen Qualitätssicherungsprozesses sowie die Entwicklung einer Evaluationsumgebung für die generierten Tests.

#### 7.3.1.1 Datenbeschaffung und -aufbereitung

Der Datenbeschaffungsprozess folgt dem in Abschnitt 6.2.2.1 beschriebenen CRISP-DM-Prozess und verwendet den Github-code-clean Datensatz[58] als Ausgangsbasis.
Der Prozess erfolgt in Python unter Verwendung der Hugging Face Datasets[59] für effizientes Datenmanagement und Dask[60] für die verteilte Datenverarbeitung.

**Initiale Datenbeschaffung** Der Ausgangsdatensatz „github-code-clean“ stellt eine gefilterte Version des Github-code Datensatzes[61] dar, bei dem bereits grundlegende Qualitätskriterien angewendet wurden:

- Durchschnittliche Zeilenlänge $< 100$ Zeichen.
- Alphanumerischer Zeichenanteil $> 0{,}25$.
- Entfernung automatisch generierter Dateien durch Schlüsselwortsuche.

Der initiale Github-code Datensatz umfasst insgesamt 115 Millionen Code-Dateien in verschiedenen Programmiersprachen, was 25.086.677.429 Token entspricht. Nach Anwendung der grundlegenden Qualitätsfilter werden etwa $3,39$ Millionen Dateien (2,94% des Datensatzes) entfernt. Von den verbleibenden Dateien im Github-code-clean Datensatz sind $19,39$ Millionen Java-Dateien, die sich in $17,19$ Millionen Code-Dateien und $2,20$ Millionen Test-Dateien aufteilen. Das entspricht insgesamt 25.086.677.429 Tokens aufgeteilt in 3.398.983.566 Test-Tokens und 21.687.693.863 Code-Tokens.

[58] github-code-clean: Vorgefilterter Code Datensatz (`https://huggingface.co/datasets/codeparrot/github-code-clean`)
[59] Hugging Face Datasets: Eine Bibliothek für effizientes Datenladen und -verarbeitung (`https://huggingface.co/docs/datasets`)
[60] Dask: Eine flexible Bibliothek für parallele Berechnungen in Python (`https://www.dask.org`)
[61] github-code: ungefilterter Code Datensatz (`https://huggingface.co/datasets/codeparrot/github-code`)

Die mehrstufige Qualitätssicherung orientiert sich an den von Nakajima und Nakatani (2021) definierten Qualitätskriterien für KI-Trainingsdaten:

- **Accuracy:** Sicherstellung der semantischen Korrektheit durch die LLM-basierte Ensemble-Filterung.
- **Completeness:** Gewährleistung vollständiger Code-Test-Paare durch den Matching-Prozess.
- **Consistency:** Standardisierung durch Pfadtransformation und einheitliche Namenskonventionen.
- **Credibility:** Validierung der Authentizität durch mehrfache Modellüberprüfung.
- **Currentness:** Aktualitätsprüfung durch die synthetische Datenanreicherung.

Die implementierte Pipeline adressiert dabei die von Nakajima und Nakatani (2021) definierten Adequacy-Dimensionen systematisch:

- **Target Selection Adequacy:** Realisiert durch die regelbasierte Pfadtransformation und das präzise Code-Test-Matching.
- **Data Selection Adequacy:** Implementiert durch den mehrstufigen Ensemble-Filterprozess.
- **Sampling Adequacy:** Umgesetzt durch die kontrollierte synthetische Datenanreicherung.

Die Provenance (Herkunft) der Daten wird durch durchgängiges Logging und regelmäßige Checkpoints dokumentiert, wodurch die Nachvollziehbarkeit aller Datenverarbeitungsschritte gewährleistet ist.

Ein Python-Skript lädt diesen Datensatz über die Hugging Face Datasets-API und führt eine unmittelbare Filterung nach Java-Dateien durch. Dabei entfernt es nicht benötigte Metadaten wie Lizenzinformationen, um den Speicherbedarf zu verringern.

**Extraktion von Test- und Implementierungsdateien** Die Trennung von Test- und Implementierungsdateien erfolgt durch eine regelbasierte Analyse der Dateipfade. Reguläre Ausdrücke identifizieren Test-Dateien anhand etablierter Namenskonventionen. Als Test-Dateien werden dabei Dateien klassifiziert, deren Namen entweder mit „Test“ beginnen oder auf „Test“ vor der Dateiendung enden, wobei die Groß- und Kleinschreibung berücksichtigt wird.

**Matching-Prozess** Der Matching-Prozess implementiert einen mehrstufigen Ansatz zur Zuordnung von Test- und Implementierungsklassen unter Verwendung der MinHash-LSH[62] Algorithmen für die Duplikaterkennung:

- **Pfadtransformation:** Die Implementierung standardisiert die Pfade durch systematische Abbildung von Test-Verzeichnissen auf entsprechende Implementierungsverzeichnisse. Dabei werden die in Java-Projekten üblichen Verzeichnisstrukturen wie „src/test/java“ und „src/main/java“ berücksichtigt.
- **Duplikaterkennung:** Zur effizienten Erkennung von Code-Duplikaten wird ein MinHash-LSH-Algorithmus implementiert. Dieser erlaubt die effiziente Identifikation ähnlicher Code-Fragmente ohne paarweisen Vergleich aller Dokumente.
- **Datenzusammenführung:** Die eigentliche Zusammenführung der Test- und Implementierungsdateien erfolgt über Dask, wobei Repository-Name, Verzeichnispfad und Basisdateiname als Schlüssel dienen.

[62] MinHash-LSH: Eine Bibliothek zur effizienten Ähnlichkeitssuche mittels Hashing (`https://github.com/ekzhu/datasketch`)

Der implementierte Matching-Prozess reduziert den initialen Datensatz von 17, 2 Millionen Java-Dateien und 2, 2 Millionen Test-Dateien auf 577.376 zusammengehörige Paare von Test- und Implementierungsklassen, was einem Gesamtumfang von 948.724.441 Code-Tokens und 821.391.840 Test-Tokens entspricht. Diese deutliche Reduktion gewährleistet, dass nur qualitativ hochwertige und eindeutig zuordenbare Code-Test-Paare für das weitere Training verwendet werden.

**LLM-basierte Qualitätsfilterung** Nach dem Matching-Prozess implementiert das System eine dreistufige LLM-basierte Ensemble Filterung (siehe Abschnitt 5.1.1) zur Qualitätssicherung der Code-Test-Paare. Die Implementierung nutzt drei verschiedene Sprachmodelle in einer Filterpipeline, wobei jedes nachfolgende Modell die vom vorherigen Modell als qualitativ hochwertig eingestuften Paare weiter analysiert.

Die Filterung erfolgt durch parallele Verarbeitung mittels ThreadPoolExecutor[63] mit mehreren gleichzeitigen Worker-Threads. Jedes Modell bewertet die Code-Test-Paare hinsichtlich ihrer Konsistenz und Qualität. Die Modelle waren zum Zeitpunkt der Filterung in ihrer Größe jeweils unter den besten Modellen ihrer Klasse speziell in ihrer Fähigkeit Code zu erstellen.

Die Pipeline umfasst:

1. **Erste Filterung (Phi-3-mini)[64]:** Eine initiale Qualitätsbewertung reduziert den Datensatz auf 44.554 Paare. Das Modell prüft grundlegende Qualitätsmerkmale und Konsistenz zwischen Code und Tests.

2. **Zweite Filterung (Code-Gemma-small)[65]:** Eine detailliertere Analyse der verbliebenen Paare durch ein auf Programmcode spezialisiertes Modell verringert den Datensatz auf 1.742 Paare.

[63] ThreadPoolExecutor: Eine Python-Bibliothek zur parallelen Ausführung von Tasks (`https://docs.python.org/3/library/concurrent.futures.html`)

[64] Phi-3-mini: Ein kompaktes Sprachmodell von Microsoft (`https://huggingface.co/microsoft/phi-3-mini`)

[65] Code-Gemma-small: Ein auf Programmcode spezialisiertes Sprachmodell von Google (`https://huggingface.co/google/code-gemma-2b`)

3. **Dritte Filterung (Llama-3-tiny)[66]:** Eine finale Qualitätsprüfung resultiert in 1.144 hochqualitativen Code-Test-Paaren mit insgesamt 988.777 Tokens.

Die Implementierung beinhaltet Mechanismen zur Fehlerbehandlung und Wiederaufnahme, einschließlich regelmäßiger Checkpoints nach jeweils 100 verarbeiteten Einträgen.
Der resultierende Datensatz enthält ausschließlich Code-Test-Paare, die von allen drei Modellen als qualitativ hochwertig eingestuft wurden, wodurch eine hohe Datenqualität für das nachfolgende Training sichergestellt wird. Durch die große Verringerung des Datensatzes wird die Trainingszeit und der Ressourchenverbrauch verbessert.

**Synthetische Datenanreicherung** Die Implementierung der synthetischen Datenanreicherung erfolgt nach der LLM-basierten Filterung und zielt darauf ab, die Qualität und Diversität des Datensatzes durch detaillierte Dokumentation und erweiterte Tests zu erhöhen. Die Implementierung nutzt GPT-4[67] und GPT-4o[68] zur Verbesserung der Code-Test-Paare und wird über die OpenAI API[69] angesteuert.

Der Anreicherungsprozess umfasst zwei Hauptaspekte:

- **Code-Dokumentation:** Für jede Implementierungsklasse werden detaillierte Docstrings generiert, die das beabsichtigte Verhalten der zu testenden Einheiten beschreiben. Diese Dokumentation folgt den Java-Dokumentationsstandards und erleichtert das Verständnis der Testanforderungen.

[66] Llama-3-tiny: Ein effizientes Sprachmodell von Meta AI (`https://huggingface.co/meta-llama/Llama-2-7b`)
[67] GPT-4: Multimodales KI-Sprachmodell von OpenAI, veröffentlicht März 2023 (`https://openai.com/index/gpt-4/`)
[68] GPT-4o: Angepasste Version von GPT-4 mit erweiterten Fähigkeiten, veröffentlicht Mai 2024 (`https://openai.com/index/hello-gpt-4o/`)
[69] OpenAI API: Eine Programmierschnittstelle für den Zugriff auf OpenAI's Sprachmodelle (`https://platform.openai.com/docs/api-reference`)

- **Test-Erweiterung:** Die existierenden Testfälle werden durch zusätzliche Tests ergänzt und mit Begründungen versehen. Jeder neue oder modifizierte Test wird mit einer „Chain-of-Thought“-Erklärung versehen, die die Testabsicht und -gestaltung erläutert.

Die Implementierung verwendet eine ThreadPoolExecutor-basierte parallele Verarbeitung mit 16 Worker-Threads und implementiert ein Checkpoint-System, das nach jeweils 100 verarbeiteten Einträgen den Fortschritt sichert.

Die synthetische Datenanreicherung resultiert in einem erweiterten Datensatz mit:

- 1.666.098 Tokens durch GPT-4 generierte Dokumentation und Tests.
- 1.122.106 Tokens durch GPT-4o generierte Dokumentation und Tests.

Dieser verbesserte Datensatz bildet die Grundlage für das nachfolgende Training des LLM zur Testgenerierung.

#### 7.3.1.2 Training des Modells

Das Training des LLM basiert auf den in Abschnitt 4.4.6 beschriebenen Trainingsmethoden und implementiert den PEFT-Ansatz mit LoRA zur Anpassung der Modellparameter.
Die Implementierung nutzt die Hugging Face Transformers-Bibliothek[70] für das Training und die Modellverwaltung.

Die Trainingsimplementierung gliedert sich in vier Hauptphasen:

- Modellinitialisierung
- Konfiguration

[70] Hugging Face Transformers: Eine Bibliothek für State-of-the-Art Natural Language Processing (`https://github.com/huggingface/transformers`)

- Trainingsausführung
- Modellspeicherung

In der Initialisierungsphase erfolgt die Quantisierung des Basismodells mittels des in Abschnitt 4.4.5 beschriebenen NF4-Formats. Diese Quantisierung komprimiert die ursprünglichen 32-Bit-Floating-Point-Werte der Modellparameter auf 4-Bit-Darstellungen, wodurch der Speicherverbrauch um den Faktor 8 sinkt. Die NF4-Kodierung ist dabei speziell für die in neuronalen Netzen typischerweise normalverteilten Gewichte optimiert, sodass trotz der Kompression die wesentlichen Modelleigenschaften erhalten bleiben.

Das System implementiert zusätzlich Gradient Checkpointing als Methode zur Verbesserung des GPU-Speicherverbrauchs während des Trainingsprozesses. Bei der Backpropagation werden die Gradienten nicht permanent im Speicher gehalten, sondern bei Bedarf neu berechnet. Diese Strategie reduziert den maximalen Speicherbedarf während des Trainings deutlich, da nur die Aktivierungen ausgewählter Schichten zwischengespeichert werden müssen. Die Neuberechnung der Gradienten führt zwar zu einer längeren Verarbeitungszeit, ermöglicht aber das Training größerer Modelle auf GPUs mit begrenztem Speicher.

Die Konfigurationsphase umfasst die Einstellung der LoRA-Parameter entsprechend der in Abschnitt 4.4.6.4 beschriebenen Methode. Ein LoRA-Rang von 64 bietet einen Kompromiss zwischen Modellkapazität und Effizienz. Die Skalierung der adaptierten Gewichte erfolgt durch einen LoRA-Alpha-Parameter von 16. Ein LoRA-Dropout von $0,1$ verhindert Überanpassung durch zufällige Deaktivierung von 10% der Verbindungen während des Trainings.

Die Trainingsausführung implementiert mehrere Optimierungstechniken zur effizienten Modellverbesserung. Ein zentrales Element stellt der Lernraten-Scheduler dar, der die Anpassung der Lernrate während des Trainings steuert. In der initialen Warmup-Phase, die sich über die ersten 10% der gesamten Trainingsschritte erstreckt, wird die Lernrate linear von null auf den konfigurierten Zielwert erhöht. Diese graduelle Steigerung stabilisiert den frühen Trainingsprozess, indem sie zu große Gewichtsanpassungen zu Beginn des Trainings verhindert, wenn das Modell noch weit vom optimalen Zustand entfernt sein kann.

Die Datenverarbeitung erfolgt durch eine Kombination aus kleinen physischen Batches und Gradient-Akkumulation. Das System verarbeitet zunächst Batches von jeweils 4 Sequenzen, was einen begrenzten GPU-Speicher von 24GB ermöglicht.

Um dennoch von den Vorteilen größerer Batch-Größen zu profitieren, werden die Gradienten über 8 aufeinanderfolgende Batch-Verarbeitungen akkumuliert, bevor eine Gewichtsaktualisierung stattfindet. Diese Technik der Gradient-Akkumulation ermöglicht eine effektive Batch-Größe von 32 Sequenzen, ohne den verfügbaren GPU-Speicher zu überschreiten. Größere Batch-Größen führen häufig zu stabileren Gradientenschätzungen und damit zu einer robusteren Modellkonvergenz, was die Qualität des resultierenden Modells verbessern kann.

Ein kontinuierliches Monitoring erfasst alle 100 Trainingsschritte den Trainings- und Evaluierungsverlust, sowie die PPL (siehe Abschnitt 4.4.9.1) als zentrale Qualitätsmetrik für die Vorhersagegenauigkeit des Modells. Die Evaluierung auf einem separaten Validierungsdatensatz erfolgt alle 500 Schritte, um die Generalisierungsfähigkeit des Modells zu überprüfen. Die Integration des Weights & Biases Systems protokolliert zusätzlich technische Leistungsmetriken wie GPU-Auslastung, Speicherverbrauch und Trainingsgeschwindigkeit.

Die systematische Anpassung der Modellparameter erfolgt durch eine Grid-Search-Strategie. Diese durchsucht den Parameterraum nach einer effizienten Kombination von Lernrate und Trainingsdauer. Die Lernrate, die die Größe der Gewichtsanpassungen während des Trainings steuert, wird in sechs Schritten zwischen $1e-5$ und $3e-4$ variiert. Die Trainingsdauer wird durch die Anzahl der Epochen definiert, wobei eine Epoche einen vollständigen Durchlauf durch den Trainingsdatensatz bezeichnet. Es werden Epochenzahlen von 1 bis 3 getestet. Jede Parameterkombination wird dreimal mit unterschiedlichen Zufallsinitialisierungen trainiert, um die Stabilität der Ergebnisse statistisch abzusichern. Die Auswahl des besten Modells basiert auf der PPL des Validierungsdatensatzes, wobei die Standardabweichung über die drei Initialisierungen als Stabilitätsmaß dient.

Nach Abschluss des Trainings werden die ursprünglichen Modellgewichte mit den durch LoRA trainierten Adaptern zu einem finalen Modell zusammengeführt. Dieses finale Modell vereint das Basiswissen des freinabgestimmten Modells mit den spezifisch für die Testgenerierung gelernten Anpassungen. Die Archivierung umfasst sowohl eine lokale als auch eine Online-Speicherung des Modells[71] sowie aller Trainingsartefakte einschließlich Konfigurationen, Metriken und Checkpoints.

### 7.3.2 Operationsphase

Die Operationsphase der White-Box-Test-Komponente implementiert den praktischen Einsatz des trainierten LLM zur Generierung von Unit-Tests. Die zuvor beschriebene Architektur wird durch drei aufeinander aufbauende Schritte realisiert:

[71] Modelgallery: `https://huggingface.co/collections/bis-aifb-kit/unit-test-generation-models-675986fed3ac91d6239c0e7e`

1. **Vorverarbeitung des Eingabe-Quellcodes:**
   Extraktion und strukturierte Aufbereitung der zu testenden Java-Klassen und deren Methoden. Diese Phase umfasst die Analyse der Codestruktur, die Identifikation relevanter Methoden sowie die Sammlung von Kontextinformationen.

2. **LLM-basierte Testgenerierung:**
   Generierung der Unit-Tests durch spezialisierte Prompting-Strategien. Das trainierte Modell nutzt die aufbereiteten Informationen, um kontextbezogene und effektive Tests zu erstellen.

3. **Nachverarbeitung und Validierung:**
   Verarbeitung der generierten Tests inklusive Syntaxprüfung, Kompilierung und Ausführung. Diese Phase stellt die Qualität und Funktionsfähigkeit der erstellten Tests sicher.

Die folgenden Abschnitte beschreiben die konkrete Implementierung dieser drei Phasen im Detail.

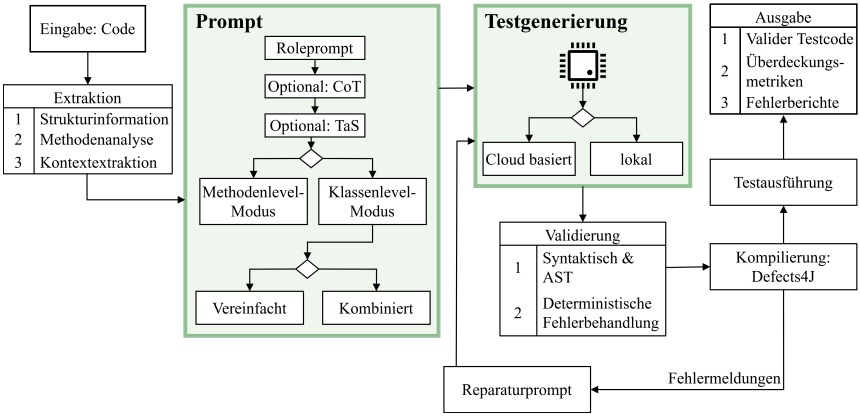



**Abbildung 7.9:** Ablaufdiagramm der Operationsphase

#### 7.3.2.1 Vorverarbeitung

Die Implementierung der Vorverarbeitung baut auf den Erkenntnissen mehrerer studentischer Abschlussarbeiten auf (Lars Bissinger 2024; Tamino Ludwig 2024; David Diemand 2024; Khalil Sakly 2022; Michael Mayer 2022) und unterstützt zwei unterschiedliche Verarbeitungsmodi für die LLM-basierte Testgenerierung.

Der **Klassenlevel-Modus** führt eine minimale Vorverarbeitung des Quellcodes durch und testet damit klassenbassiert. Diese beschränkt sich auf:

- Entfernung von Lizenzbedingungen und Kommentaren vor der Package--Deklaration
- Extraktion der Package-Deklaration mittels regulärer Ausdrücke zur korrekten Positionierung

Der **Methodenlevel-Modus** implementiert eine detaillierte Analyse des Quellcodes durch den `javalang`-Parser zur Erstellung eines Abstract Syntax Tree (AST)[72] und testet damit methodenbasiert.

Die systematische Extraktion nutzt den `javalang`-Parser[73] und umfasst:

- **Strukturinformationen:**
  - Package-Deklaration für die Namensraumzuordnung
  - Import-Statements zur Abhängigkeitsanalyse
  - Klassendeklaration
- **Methodenanalyse:** Kategorisierung in:

[72] AST: Eine baumartige Repräsentation der syntaktischen Struktur des Quellcodes, die von Compilern und Entwicklungswerkzeugen zur Codeanalyse verwendet wird (`https://en.wikipedia.org/wiki/Abstract_syntax_tree`)

[73] javalang: Eine Python-Bibliothek zum Parsen von Java-Quellcode (`https://github.com/c2nes/javalang`)

  - Öffentliche Methoden (Testgenerierungsziele)
  - Private Methoden (Abhängigkeitsanalyse)
  - Konstruktoren (Objektinstanziierung)

- **Kontextextraktion:** Für jede Methode wird ein `MethodData`-Objekt erstellt mit:
  - Methodensignatur (Parameter, Rückgabetyp)
  - Dokumentation (JavaDoc, Inline-Kommentare)
  - Anweisungen (Methodenkörper)
  - Klassenkontext

Ein durchgängiges Logging-System protokolliert sämtliche Verarbeitungsschritte und auftretende Fehler. Die aufbereiteten Informationen werden anschließend in den entsprechenden Prompt-Templates für die LLM-basierte Testgenerierung verwendet.

Die resultierenden `MethodData`-Objekte bilden eine strukturierte Repräsentation des Quellcodes, die speziell auf die Anforderungen der nachfolgenden LLM-basierten Testgenerierung ausgerichtet ist. Durch die umfangreiche Kontextinformation wird eine zuverlässige Grundlage für die Generierung korrekter Unit-Tests geschaffen.

#### 7.3.2.2 Prompting-Implementierung

Die Implementierung des Prompting-Systems baut auf den in Abschnitt 4.4.8 beschriebenen Prompting-Techniken auf und wurde auf Basis der Erkenntnisse aus folgenden studentischen Abschlussarbeiten kontinuierlich verbessert (Lars Bissinger 2024; Tamino Ludwig 2024; Michael Mayer 2022). Das System implementiert einen modularen, templatebasierten Ansatz zur Generierung von Unit-Tests, der die verschiedenen Aspekte des Prompting-Prozesses systematisch adressiert. Die Auswahl der verwendeten Prompttemplates kann über die in Abschnitt 7.2.3 beschriebenen Parameter erfolgen.

Den Kern des Systems bildet das in Prompt 7.1 dargestellte, rollenbasierte Basistemplate, der das LLM in die Rolle eines Software-Testers mit Spezialisierung auf Java Unit-Tests versetzt. Diese Rollendefinition orientiert sich an den Prinzipien des Role Prompting und wird durch eine niedrige Sampling-Temperatur von $0,3$ ergänzt, die aber über eine Umgebungsvariable bei Bedarf angepasst werden kann. Diese konservative Temperatureinstellung reduziert die Zufälligkeit in den generierten Ausgaben und gewährleistet, dass die Ausgaben des Modells sich nahe am gegebenen Input orientieren.

```
You are an experienced software tester specializing in Java
↪  Unit Tests.

Your task is to create a meaningful JUnit4 test class for
↪  the following Java class and method:
```

**Prompt 7.1:** Rolleninformation und Aufgabendefinition

Die Template-Struktur folgt einem hierarchischen Aufbau, der die verschiedenen Informationsebenen systematisch integriert. Zunächst werden die grundlegenden Kontextinformationen wie Package-Deklarationen und Import-Statements eingebettet, gefolgt von den Methodendetails einschließlich Signatur, Dokumentation und Implementierung. Diese strukturierte Informationsbereitstellung ermöglicht es dem LLM, den Kontext der Testaufgabe vollständig zu erfassen.

Für unterschiedliche Testszenarien implementiert das System drei verschiedene Prompting-Strategien, die je nach Anforderung aktiviert werden können. Der **Methodenlevel-Modus** verwendet das beschriebene Role Prompting für reguläre Testfälle und kann bei Bedarf die weiteren Prompting-Techniken hinzufügen kann. Der **Klassenlevel-Modus** verwendet jeweils eine vereinfachte und kombinierte Version der Prompting-Techniken.

Bei erhöhter Komplexität, mit mehreren voneinander abhängigen Methoden, kann zusätzlich Chain-of-Thought (CoT) hinzugefügt werden, das entsprechend der in Abschnitt 4.4.8 beschriebenen Methode eine schrittweise Zerlegung des Testproblems vornimmt. Dieser Prompt kommt vor allem beim Methodenbasierten Testen zum Einsatz.

```
Let's proceed step-by-step:
1. First, understand the class and method behavior
2. Identify the method's dependencies and required setup
3. Plan test cases for different scenarios
4. Implement the test class with proper assertions
```

**Prompt 7.2:** Chain-of-Thought Prompt Part

Zusätzlich steht der TaS-Ansatz zur Verfügung, der durch eine initiale Abstraktionsebene das grundlegende Verständnis der Testanforderungen verbessert.

```
Before we start generating unit tests, let's take a step
↪  back and consider:
1. What are the general principles of unit testing?
2. What are the specific challenges of testing this type of
↪  method?
3. What are the most important edge cases to consider?
```

**Prompt 7.3:** Take a Step Back Prompt Part

Ein wesentliches Merkmal der Implementierung ist die flexible Kontextualisierung des Quellcodes. Das System ermöglicht sowohl die isolierte Betrachtung einzelner Methoden als auch die Integration der gesamten Klassenstruktur. Der Methodenlevel-Modus eignet sich dabei besonders für Unit-Tests einzelner Funktionalitäten, während der Klassenlevel-Modus eine umfassendere Testüberdeckung auch mehrschrittiger Funktionen ermöglicht. Der Methodenlevel-Modus bietet sowohl die Möglichkeit, die Methoden einzeln zu betrachten und auszuführen, als auch die Option, die generierten Tests vor der Ausführung mithilfe des in Prompt 7.4 dargestellten Prompts zu einer Testklasse zusammenzufassen.

```
You are an experienced software tester specializing in Java
↪  Unit Tests.
Your task is to combine the following JUnit4 test classes
↪  into a single test class.
Make sure to remove any duplicate test methods and ensure
↪  that the combined test class maintains the original
↪  test functionality.
Classes to combine:
{test_methods}

Important:
1. Combine the test methods into a single test class.
2. Ensure that the combined test class maintains the
↪  original test functionality.
3. Ensure the generated test code is compatible with the
↪  specified target Java source version
↪  ({source_version}).
Name the combined test class {class_name}Test and make it
↪  public.
```

**Prompt 7.4:** Methodentest Prompt um mehrere Methoden einer Testklasse zu kombinieren

Beim Methodenlevel-Modus wird der Prompt in mehrere logische Einheiten unterteilt, die aufeinander aufbauen. Als grundlegende Kontextinformation wird zunächst die Paketstruktur bereitgestellt:

```
Package Information:

package org.calculator;
...
//further imports
```

**Prompt 7.5:** Package Information

Darauf aufbauend wird dem LLM der vollständige Klassenkontext zur Verfügung gestellt, der ein umfassendes Verständnis der zu testenden Funktionalität ermöglicht. Zusätze wie beispielsweise Lizenzbedingungen werden vor der Deklaration von *package* entfernt.

Für die Testgenerierung wird anschließend die zu testende Methode mit ihrer Signatur, den zugehörigen Kommentaren und der konkreten Implementierung bereitgestellt:

```
//method_signature
float mod(float a, float b)
//method_comments
 * Calculates the modulus of the first number by the second
 ↪  number.
 *
 * @param a The first number.
 * @param b The second number.
 * @return The result of a % b.
 */
//method_body
    return a % b;
```

**Prompt 7.6:** Zu testende Methode

Um sicherzustellen, dass alle notwendigen Aspekte in den generierten Tests berücksichtigt werden, werden dem LLM abschließend detaillierte Anforderungen für die Testgenerierung als Gesamtprompt übergeben. Die Anforderungen wurden systematisch aus den *JUnit Best Practices*[74] abgeleitet. Grundlegende Testanforderungen wie korrekte Annotationen und Imports basierend auf etablierten JUnit-Konventionen.
Die resultierende Liste von Anforderungen wurde iterativ verfeinert und nach ihrer Wirksamkeit der generierten Tests validiert. Die präzise Formulierung der Anforderungen ist dabei essentiell, da das LLM diese direkt in konkrete Teststrukturen übersetzt:

```
* Requirements:
1. Use JUnit4 annotations (@Test, @Before, etc.)
2. Add the appropriate imports (e.g.):
    - org.junit.*;
    - Any required classes from the same package:
    ↪ {package_info}
3. Create a minimal set of test cases for:
    - Normal/expected inputs
    - Edge cases
    - Error conditions
5. Don't write repetitive or redundant test cases. Focus on
↪  the most important edge cases and error conditions.
6. Add a test setup if necessary
7. Add corresponding assertions
8. Handle all required exception tests
9. Include the same package declaration as the class being
↪  tested
```

**Prompt 7.7:** Anforderungen an die Testgenerierung

[74] JUnit Best Practices: Empfohlene Vorgehensweisen für die Verwendung des JUnit-Frameworks (`https://junit.org/junit5/docs/current/user-guide/`)

Diese systematische Strukturierung des Prompts unterstützt das LLM bei der Generierung von Unit-Tests.

Die technische Integration des Prompting-Systems erfolgt über eine standardisierte API-Schnittstelle, die sowohl cloud-basierte LLMs als auch die in Abschnitt 7.3.1.2 trainierten lokalen Modelle unterstützt. Damit es nicht zu Endlosschleifen kommt, wird die maximale Anzahl an Tokens begrenzt, nachdem das Modell die Beantwortung des Prompts abschließt. Die API-Abstraktion erlaubt zudem einen einfachen Austausch der verwendeten Modelle, was die kontinuierliche Integration verbesserter LLMs vereinfacht.

#### 7.3.2.3 Testgenerierung und -ausführung

Die Implementierung der Testgenerierung erfolgt ebenfalls durch eine mehrstufige Pipeline, die auf der in Abschnitt 6.2 beschriebenen Architektur aufbaut. Der Prozess umfasst die Validierung der generierten Tests, deren Ausführung mittels Maven sowie die Extraktion von Testmetriken durch JaCoCo. Zentral ist dabei die Unterscheidung zwischen syntaktischer Korrektheit, Kompilierbarkeit und Ausführbarkeit der Tests, die in aufeinanderfolgenden Phasen sichergestellt werden.

**Systeminitialisierung**

Die Ausführungslogik ist in der `TestExecutor`-Klasse implementiert, die den gesamten Prozess der Testgenerierung, -ausführung und -analyse koordiniert.

Der Ausführungsprozess beginnt mit der Initialisierung der Maven-basierten Testumgebung. Die Initialisierung umfasst die Erstellung der erforderlichen Verzeichnisstrukturen sowie die Integration der generierten Tests in das standardisierte Maven-Projektlayout mit `src/main` und `src/test` Verzeichnissen. Vor der Maven-basierten Ausführung durchläuft der generierte Testcode eine *syntaktische Validierung und Fehlerbehandlung.*

**Syntaktische Validierung und Fehlerbehandlung**

Die syntaktische Validierung und Fehlerbehandlung des generierten Testcodes wird durch ein mehrstufiges System realisiert, das auf den Klassen `TestCodeProcessor`, `CodeValidator` und `CodeAnalyzer` aufbaut.

Der `TestCodeProcessor` extrahiert zunächst den relevanten Code aus der LLM-Antwort und führt grundlegende Bereinigungen durch. Dabei werden Markup-Artefakte entfernt, Einrückungen standardisiert und redundante Code-Block-Begrenzungen eliminiert. Ein spezieller Algorithmus identifiziert und entfernt zudem irrelevanten Code nach der letzten schließenden Klassenklammer.

Der `CodeValidator` analysiert anschließend die fundamentale Teststruktur. Die Validierung umfasst die Überprüfung der Package-Struktur, Import-Statements sowie der korrekten Deklaration von Testklassen und -methoden. Besonderes Augenmerk liegt auf der Präsenz essentieller Test-Elemente wie @Test-Annotationen und Assert-Statements.

Eine tiefergehende Analyse erfolgt durch den `CodeAnalyzer`, der einen vollständigen AST des Testcodes erstellt. Diese AST-basierte Analyse ermöglicht eine präzise Identifikation struktureller Probleme und deren systematische Korrektur. Der Analyseprozess erkennt fehlende Methodenmodifikatoren, inkorrekte Testannotationen und unvollständige Assert-Statements. Basierend auf der Analyse der Methodensignaturen und Rückgabewerte werden fehlende Assertions automatisch generiert und eingefügt.

Die `CodeRepairer`-Klasse implementiert die eigentliche Fehlerkorrektur. Bei identifizierten Syntaxfehlern wird ein mehrstufiger Reparaturprozess initiiert, der zunächst die problematischen Codebereiche lokalisiert und isoliert. Anschließend erfolgt eine schrittweise Korrektur, beginnend mit häufigen Fehlermustern wie fehlenden Imports oder inkorrekten Annotationen. Bei persistierenden Problemen reduziert der Repairer den Code systematisch auf seine funktionalen Kernbestandteile, wodurch zumindest eine Teilfunktionalität erhalten bleibt.

Die Import-Analyse stellt einen weiteren zentralen Aspekt der Validierung dar. Durch AST-Traversierung werden fehlende Abhängigkeiten identifiziert und automatisch als Vorschlag ergänzt. Dies umfasst sowohl projektspezifische Imports als auch Standard-Test-Dependencies wie JUnit und Mockito. Eine anschließende Deduplizierung und Sortierung der Import-Statements gewährleistet eine übersichtliche und wartbare Codestruktur.

Die gesamte Validierungs- und Korrekturpipeline implementiert das Prinzip der *graceful degradation*. Selbst bei schwerwiegenden Syntaxfehlern wird versucht, einen maximal funktionsfähigen Testcode zu erhalten. Alle Validierungsschritte und Korrekturen werden detailliert protokolliert, wodurch eine systematische Verbesserung der LLM-basierten Testgenerierung ermöglicht wird. Nicht behebbare Fehler fließen in die iterative Verbesserungsphase ein, in der das LLM gezielt zur Korrektur spezifischer Probleme aufgefordert wird.

Diese erste Validierungsebene stellt die *syntaktische Korrektheit* sicher und filtert ungültige Tests vor der Kompilierungsphase aus, falls sie nicht automatisch korrigiert werden können.

#### Test-Ausführung

Die eigentliche Testausführung erfolgt in zwei aufeinander aufbauenden Phasen: In der **ersten Phase** werden die syntaktisch validierten Unit-Tests kompiliert und ausgeführt. Die Kompilierung stellt als zweite Validierungsebene die *Kompilierbarkeit* der generierten Tests sicher. Die Ausführung erfolgt durch das Maven Surefire Plugin[75], das die standardisierte Testausführungsumgebung für Java-Projekte bereitstellt und damit die *Ausführbarkeit* der Tests verifiziert.

[75] Maven Surefire Plugin: Eine Testausführungsumgebung für Java-Projekte (`https://maven.apache.org/surefire/maven-surefire-plugin/`)

Die **zweite Phase** fokussiert sich auf die Erfassung und Analyse der Testüberdeckung durch JaCoCo (siehe Abschnitt 3.3.5). Die Integration von JaCoCo erfolgt über ein Maven-Plugin, das während der Testausführung detaillierte Überdeckungsdaten sammelt.

Das System erfasst dabei:

- **Überdeckungsmetriken:** Zeilen- und Zweigüberdeckung gemäß Abschnitt 3.2.3.
- **Ausführungsmetriken:** Erfolgs- und Fehlerquoten der Tests.
- **Fehlerberichte:** Detaillierte Stacktraces und Fehlermeldungen.

Ein `ProgressTracker` aggregiert diese Metriken und stellt sie der Frontend-Schnittstelle in Echtzeit zur Verfügung.

#### Iterative Testverbesserung

Bei fehlgeschlagenen Tests initiiert das System einen dreistufigen Verbesserungsprozess. Dies erfolgt nur, wenn überhaupt syntaktisch korrekter Code erstellt wurde. Ansonsten wird die Methode übersprungen. Die Fehlerinformationen aus Maven und JaCoCo werden in einen weiteren strukturierten Prompt (siehe Verbesserungs Prompt) transformiert, der das LLM zur gezielten Fehlerkorrektur anleitet.

```
You are an expert software tester specializing in java
↪  apps.
Fix the following JUnit4 test with the given error.
Ensure the generated test code is compatible with the
↪  specified target Java source version
↪  ({source_version}).
If necessary, remove the problematic test code.
Return only the fully corrected code file without
↪  explanations and remove text between the code.
Test Code:
{validated_code}
*Error:
`{error}`
```

**Prompt 7.8:** Verbesserungs Prompt, der das Large Language Model anweist, den Code mithilfe der Kompilierfehler zu verbessern

Die Fehlerkorrektur erfolgt iterativ in drei Schritten:

1. **Fehleranalyse:** Systematische Auswertung der Compiler- oder Laufzeitfehler
2. **Prompt-Generierung:** Integration der Fehlerinformationen in ein spezialisiertes Korrektur-Template entsprechend Abschnitt 7.3.2.2
3. **Test-Regenerierung:** Erzeugung einer verbesserten Testversion durch das LLM

Dieser Zyklus wird bis maximal dreimal durchlaufen, bevor eine Methode als nicht automatisiert testbar markiert wird.

Die Fehlerbehandlung implementiert dabei eine *graceful degradation*, bei der auch bei suboptimalen Eingaben noch brauchbare Tests generiert werden können. Das System erkennt und behandelt verschiedene Arten von Eingabefehlern systematisch, von syntaktischen Problemen bis hin zu unvollständigen Methodensignaturen.

Die robuste Fehlerbehandlung in Kombination mit den verschiedenen Prompting-Strategien ermöglicht eine zuverlässige Testgenerierung auch bei herausforderenden Codebasen. Zu diesen Herausforderungen zählen unzureichend dokumentierter Quellcode, die Verwendung projektspezifischer oder ungewöhnlicher Bibliotheken sowie komplexe Abhängigkeitsstrukturen zwischen Klassen und Methoden. Die Fehlerbehandlung versucht auch unter diesen erschwerten Bedingungen valide Tests zu generieren.

**LLM-generierte Fehlererklärungen**

Bei den fehlgeschlagenen Tests wird das LLM erneut aufgerufen, diesmal mit dem Fokus auf die Analyse und Erklärung der Fehlerursache. Das Modell generiert dabei Erklärungen und Lösungsvorschläge der Fehlerursachen, die Entwickler bei der manuellen Korrektur oder dem Verständnis der Testproblematik unterstützen. Diese Erklärungen basieren auf der Analyse der Compiler-Ausgaben, Laufzeitfehler und der Testüberdeckungsdaten.

**Ergebnisspeicherung und -kommunikation**

Das System speichert sämtliche Testergebnisse in einer strukturierten JSON-Datei. Diese enthält neben den quantitativen Metriken auch die qualitativen LLM-Analysen und dient als Grundlage für weitere Analysen und kontinuierliche Verbesserungen. Die Kommunikation mit der Frontend-Komponente erfolgt über die in Abschnitt 7.2 beschriebene Schnittstelle und ermöglicht eine Verfolgung des Testfortschritts. Die asynchrone Aktualisierung der Benutzeroberfläche erlaubt Entwicklern, den Fortschritt der Testausführung kontinuierlich zu überwachen und mögliche Probleme frühzeitig zu identifizieren.

#### 7.3.2.4 Implementierung der Evaluationspipeline

Zur Bewertung der LLM-basierten Testgenerierung implementiert die Evaluationspipeline einen systematischen Ansatz unter Verwendung des Defects4J-Frameworks (siehe Abschnitt 3.3.5.5). Basierend auf den in Abschnitt 7.3.2 beschriebenen Phasen und Prozessen der White-Box-Test-Komponente wurde die Implementierung um spezifische Anforderungen der systematischen Evaluation erweitert. Im Fokus steht dabei die automatisierte Evaluation der generierten Unit-Tests hinsichtlich ihrer syntaktischen Korrektheit, Ausführbarkeit und erreichten Codeüberdeckung.

Die Architektur der Pipeline basiert auf einem modularen Aufbau mit spezialisierten Komponenten. Zentral koordiniert ein `PipelineOrchestrator` den gesamten Evaluationsprozess und steuert die Interaktion zwischen den einzelnen Komponenten. Mit Hilfe des `Defects4JHandler` erfolgt die Verarbeitung der Eingabe-Codebasis aus dem Repository sowie die Extraktion relevanter Java-Klassen unter Beibehaltung der Projektstruktur und Abhängigkeiten. Über einen `ModelHandler` wird die Interaktion mit dem LLM realisiert, wobei sowohl lokal trainierte als auch Cloud-basierte Modelle unterstützt werden. Ein spezialisierter `TestEvaluator` übernimmt die Kompilierung und Ausführung der generierten Tests sowie die Erfassung der Evaluationsmetriken. Die Berechnung der Codeüberdeckung wird durch einen `CoverageCalculator` realisiert.

Der Initialisierungsprozess der Pipeline umfasst mehrere aufeinander aufbauende Schritte. Nach der Konfiguration der Logging-Mechanismen erfolgt das Laden der Modell- und Evaluationsparameter aus der Datei `config.py`. Die anschließende Initialisierung der Handler-Komponenten etabliert die notwendigen Verbindungen zum Defects4J-Repository und dem ausgewählten LLM. Bei einer Wiederaufnahme der Evaluation prüft das System auf existierende Evaluationsergebnisse in `result_output/`, um eine effiziente Fortsetzung zu ermöglichen und redundante Berechnungen zu vermeiden.

Die eigentliche Testgenerierung und -evaluation folgt einem systematischen Prozess. Für jede zu testende Java-Klasse extrahiert die Pipeline zunächst die relevante Java-Version aus der `pom.xml` des Projekts durch die Methode `extract_java_version_from_pom`. Diese Information fließt in die Generierung der Unit-Tests ein, die entweder durch eine klassenbasierte oder methodenbasierte Strategie erfolgt. Die klassenbasierte Strategie generiert Tests für die gesamte Klasse in einem Durchgang, während die methodenbasierte Strategie einzelne Methoden separat betrachtet und die resultierenden Tests durch den `TestCombiner` mit Prompt 7.4. zusammenführt oder jeweils einzeln ausführt. Die generierten Tests durchlaufen den oben beschriebenen mehrstufigen Validierungs- und Reparaturprozess in der `validate_and_repair` Methode, bevor sie im Kontext des Defects4J-Projekts kompiliert und ausgeführt werden.

Die Implementierung unterstützt eine flexible Konfiguration der Evaluationsparameter über die `config`-Strukturen. Für das `gpt-4o-mini` Modell werden sämtliche Prompt-Konfigurationen systematisch evaluiert, einschließlich verschiedener Kombinationen von Einzelmethoden-Generierung (`use_single_methods`), CoT Reasoning (`use_cot`) und dem TaS Prinzip (`use_tas`). Lokale Modelle arbeiten mit einer angepassten Standard-Prompt-Konfiguration. Die Ausführungsparameter ermöglichen die Steuerung der Reparaturversuche (`REPAIR_ATTEMPTS`) und die parallele Durchführung mehrerer Evaluationen durch eindeutige `INSTANCE`-IDs.

Die Pipeline implementiert ein umfassendes Logging-System zur detaillierten Dokumentation des Evaluationsprozesses. Sämtliche Zwischenergebnisse und finalen Evaluationsergebnisse werden in strukturierten JSON-Dateien gespeichert. Diese enthalten sowohl detaillierte Metriken zur Testqualität als auch aggregierte Evaluationsergebnisse, die eine systematische Analyse der Testgenerierungsqualität ermöglichen. Die standardisierte Struktur der Ergebnisdateien erleichtert den Vergleich verschiedener Konfigurationen und bildet die Grundlage für die kontinuierliche Verbesserung des Ansatzes.

Die implementierte Evaluationspipeline ermöglicht eine systematische und reproduzierbare Bewertung der LLM-basierten Testgenerierung. Durch die modulare Architektur und flexible Konfigurationsmöglichkeiten kann die Pipeline an verschiedene Evaluationsszenarien angepasst werden. Das integrierte Logging-System gewährleistet die Nachvollziehbarkeit der Ergebnisse und unterstützt die wissenschaftliche Analyse der Testgenerierungsqualität.

Die automatisierte Konfigurationserstellung und parallele Ausführung der Evaluationspipeline erfolgt durch zwei spezialisierte Python-Module.

Die Klasse `ConfigGenerator` implementiert einen systematischen Ansatz zur Erstellung von Testkonfigurationen basierend auf Pairwise-Testing, während der `MultiInstanceRunner` die parallele Ausführung der Evaluationen koordiniert. Die Konfigurationserstellung nutzt die `AllPairs`-Bibliothek zur systematischen Kombination der Testparameter. Die relevanten Parameter umfassen die Anzahl der Reparaturversuche (`repair_attempts`), die Methodengenerierungsstrategie (`use_single_methods`) sowie die Prompt-Konfigurationen wie `n_shots`, `use_cot`, `use_tas` und `use_source_code`. Für jede generierte Parameterkombination wird eine separate Konfigurationsdatei erstellt, die eine eindeutige `INSTANCE`-ID erhält.

Der `MultiInstanceRunner` implementiert eine ressourceneffiziente parallele Ausführung der Evaluationen. Das System unterscheidet dabei zwischen lokalen Modellen und Cloud-basierten Diensten. Für lokale Modelle wird die Anzahl parallel laufender Instanzen auf maximal zwei begrenzt, um eine Überlastung der GPU-Ressourcen zu vermeiden. Cloud-basierte Modelle unterliegen keiner derartigen Beschränkung. Die Ausführung erfolgt durch separate Prozesse, die über einen `ProcessManager` koordiniert werden. Ein integriertes Logging-System protokolliert den Ausführungsstatus jeder Instanz. Diese automatisierte Konfigurationserstellung und parallele Ausführung ermöglicht eine effiziente und systematische Evaluation verschiedener Modell- und Prompt-Kombinationen. Die implementierte Ressourcensteuerung gewährleistet dabei eine ausgeglichene Nutzung der verfügbaren Hardware-Ressourcen.

**Auswertung der Evaluation**

Die Auswertung der Evaluationsergebnisse erfolgt durch den `ResultAnalyzer`, der eine systematische Analyse der generierten JSON-Dateien implementiert. Die Implementierung nutzt das Pandas-Framework zur effizienten Datenverarbeitung und -aggregation sowie das Defects4J-Framework zur Validierung der Testergebnisse.

Der `ResultAnalyzer` extrahiert für jede Evaluationskonfiguration die relevanten Metriken zur Bewertung der Testqualität. Die primären Metriken umfassen die Anzahl der fokalen Methoden (`Focal_Methods`), die syntaktische Korrektheit der Tests, die Erfolgsrate der Testfälle (`Test_Cases_Passing`) und auftretende Build-Fehler (`Test_Cases_Build_Error`). Zusätzlich wird die erreichte Zeilen- und Bedingungsüberdeckung ermittelt, die einen objektiven Vergleich der Testqualität ermöglicht. Ein integrierter `CoverageCalculator` berechnet die tatsächliche Codeüberdeckung der generierten Tests durch automatisierte Ausführung im Defects4J-Framework.

Die Analyseergebnisse werden in einer strukturierten tabellarischen Form aufbereitet. Neben den quantitativen Metriken zur Testqualität erfasst die Analyse auch qualitative Aspekte wie die Laufzeit der Evaluationen und die verwendeten Prompt-Konfigurationen. Der `ResultAnalyzer` berechnet zudem aggregierte Metriken wie die Gesamterfolgsrate der Testgenerierung und die durchschnittliche Testüberdeckung für verschiedene Modellkonfigurationen. Die implementierte Auswertungslogik unterstützt sowohl die detaillierte Analyse einzelner Evaluationsläufe als auch den systematischen Vergleich verschiedener Modell- und Prompt-Konfigurationen. Durch die standardisierte Aufbereitung und Speicherung der Ergebnisse wird eine wissenschaftlich fundierte Analyse der LLM-basierten Testgenerierung ermöglicht. Die gewonnenen Erkenntnisse bilden die Grundlage für die kontinuierliche Verbesserung des Ansatzes und die Anpassung der Testgenerierungsstrategien.

## 7.4 Black-Box-Test-Komponente

Die Implementierung der Black-Box-Test-Komponente realisiert die in Abschnitt 6.3 konzipierte Architektur.

Die technische Umsetzung basiert auf dem AndroidEnv-Framework (siehe Abschnitt 4.6.1). Als Trainingsapp dient die integrierte Vokram-App[76], die eine Testumgebung für die RL-Agenten bietet.

Alle in AndroidEnv integrierten Test-Apps können als austauschbare Testumgebung für die RL-Agenten dienen. Sie liefern jeweils einen intern berechneten Score als Belohnungssignal zur Bewertung des Trainings.

Die Implementierung gliedert sich in die Trainingsphase zur Entwicklung eines RL-Agent und die Operationsphase für die eigentliche Testdurchführung. Die Trainings- und Operationsphase sind in einer containerisierten Umgebung implementiert. Diese wird in Abschnitt 7.1 beschrieben und ermöglicht reproduzierbare Testbedingungen.
Die zentralen technischen Herausforderungen umfassen dabei die echtzeitfähige Verarbeitung von Bildschirminhalten zur Gewährleistung flüssiger Benutzerinteraktionen sowie die zuverlässige Interaktion mit der Benutzeroberfläche. Die Echtzeitfähigkeit erfordert eine schnelle Verarbeitungszeit einzelner Bildschirmaufnahmen, um eine verzögerungsfreie Interaktion des Agenten mit der Benutzeroberfläche zu ermöglichen.

[76] Vokram: App zur Erprobung von Markov-Entscheidungsprozessen durch farbige Bildschirmschaltflächen `https://github.com/google-deepmind/android_env/blob/main/docs/example_tasks.md#vokram`

### 7.4.1 Trainingsphase

Die Trainingsphase implementiert einen auf dem Proximal Policy Optimization (PPO)-Algorithmus basierenden RL-Agenten (Schulman u. a. 2017). Die Wahl von PPO (siehe Abschnitt 4.5.1.2) begründet sich durch dessen effiziente Verarbeitung der stetigen Touch-Koordinaten bei gleichzeitig geringerem VRAM-Bedarf im Vergleich zu alternativen Algorithmen wie SAC.
Obwohl SAC eine potenziell bessere Explorationsfähigkeit bietet, erwies sich der reduzierte Ressourcenbedarf von PPO als vorteilhafter für die praktische Implementierung.
Aufgrund der Verwendung von Stable Baselines (siehe Abschnitt 4.5.1.4) ist die Implementierung flexibel und ermöglicht die Verwendung verschiedener RL-Algorithmen.

Der in Abbildung 7.10 dargestellte Agent operiert mit einem vereinfachten Aktionsmodell, das alle Interaktionen auf einfache Touch-Events (insbesondere „Taps“) reduziert. Diese Vereinfachung erfolgt durch einen spezialisierten Action Wrapper, der die komplexeren, von AndroidEnv definierten, GUI-Events (*LIFT*, *TOUCH*, *REPEAT*) in einzelne Touch-Events übersetzt. Dabei entspricht ein Tap einem kurzen Touch-Event, bestehend aus einem *TOUCH* gefolgt von einem *LIFT*. Diese Reduktion vereinfacht den Aktionsraum erheblich, während gleichzeitig die grundlegende Interaktionsfähigkeit mit der Benutzeroberfläche erhalten bleibt.
Sollen in Zukunft weitere Interaktionen unterstützt werden, kann der Aktionsraum um komplexere Touch-Gesten wie Swipes oder Multi-Touch-Events erweitert werden.

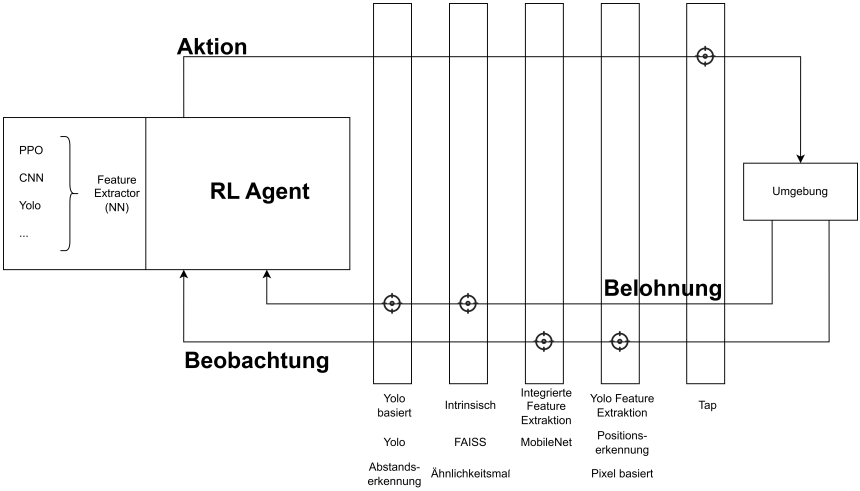



**Abbildung 7.10:** Aufbau RL Agent

Die X- und Y-Koordinaten der Touch-Events werden im Bereich [0,1] normalisiert, was eine grundlegende Geräteunabhängigkeit innerhalb vergleichbarer Geräteklassen ermöglicht. Diese Normalisierung funktioniert zuverlässig für Smartphones und Tablets mit ähnlichen Aspektverhältnissen. Bei extremen Bildschirmgrößenunterschieden, etwa zwischen einem Tablet und einer Smartwatch, oder bei stark abweichenden Aspektverhältnissen können jedoch Anpassungen der Touch-Koordinaten erforderlich sein, da sich die effektive Präzision der Interaktion und die relative Größe der GUI-Elemente erheblich unterscheiden. Die aktuelle Implementierung fokussiert sich auf Smartphones als primäre Zielplattform.

Die technische Integration mit AndroidEnv erfolgt über eine spezialisierte Wrapper-Struktur, die die standardisierten Gymnasium-Schnittstellen[77] implementiert und die Kommunikation mit dem Android-Emulator (DUT) koordiniert.

Die Trainingsphase verwendet die in Abschnitt 6.3.1 beschriebenen Komponenten Feature-Extraktion, Aktionsmodellierung, Belohnungssystem und Systemintegration.

#### 7.4.1.1 Feature-Extraktion

Die Feature-Extraktion implementiert zwei unterschiedliche Ansätze zur Verarbeitung der Bildschirminhalte der Android-Applikation. Diese sind in Abbildung 7.10 in der Beobachtung verortet. Die erste Variante nutzt eine vortrainierte YOLOv5-Architektur zur Erkennung von GUI-Elementen, während die zweite Variante direkt auf den Bildschirmpixeln operiert. Die Gewichte der YOLOv5-Architektur wurden im Rahmen einer studentischen Abschlussarbeit auf einem eigenen Datensatz von GUI-Elementen trainiert, um eine effiziente Erkennung der relevanten Bildschirmelemente zu gewährleisten (Vito Pepe Jaromir Völker 2022). Dies ermöglicht einen Vergleich der Effektivität beider Ansätze für die GUI-Testgenerierung.

[77] Gymnasium: Standardisierte Schnittstelle für die Integration von RL-Agenten in AndroidEnv `https://gymnasium.farama.org/index.html`

**YOLO-basierte Feature-Extraktion**

Die YOLO-basierte Implementierung nutzt eine vortrainierte YOLO-Architektur zur Extraktion relevanter Features aus den Bildschirminhalten. Diese Implementierung adressiert das Cold-Start-Problem des RL-Agenten durch die Nutzung bereits trainierter Modelle für die GUI-Element-Erkennung.

Die Architektur implementiert eine dreistufige Featureextraktion und Featurereduktion:

- **Featureextraktion:** Drei sequentielle Module (M1, M2, M3) verarbeiten das Eingabebild auf unterschiedlichen Abstraktionsebenen. Diese extrahieren Features mit steigender Komplexität:
  - M1 verarbeitet niedrigstufige Features mit 128 Kerneln auf einer 80x80 Feature-Map.
  - M2 extrahiert mittlere Abstraktionsebenen mit 256 Kerneln auf einer 40x40 Feature-Map.
  - M3 erfasst hochstufige Features mit 512 Kerneln auf einer 20x20 Feature-Map.
- **Dimensionsreduktion:** Parameter-Reducer in Form von 1x1 Konvolutionen reduzieren die Dimensionalität der Feature-Maps jeder Stufe. Diese Reduktion verringert die Komplexität der nachfolgenden Verarbeitung deutlich, während relevante Informationen weitgehend erhalten bleiben. Die 1x1 Konvolutionen komprimieren dabei die Feature-Maps auf eine niedrigere Dimensionalität, die einen praktikablen Kompromiss zwischen Verarbeitungseffizienz und Informationserhalt darstellt.

- **Feature-Integration:** Eine finale lineare Schicht kombiniert die reduzierten Features aller drei Stufen zu einem 256-dimensionalen Featurevektor. Diese Integration ermöglicht die Berücksichtigung von Features unterschiedlicher Abstraktionsebenen in der resultierenden Repräsentation.

Die Vorverarbeitung der Eingabebilder erfolgt durch Normalisierung und Padding (siehe Abschnitt 4.4.2.4) auf die erforderliche Eingabegröße. Die Normalisierung der Pixelwerte auf den Bereich [0,1] reduziert numerische Instabilitäten während des Trainings und der Inferenz, da große Werteunterschiede in den Eingabedaten zu unerwünschten Gradientenexplosionen führen können. Das implementierte Padding auf 160x160 Pixel gewährleistet die Kompatibilität mit der YOLO-Architektur.

Diese mehrstufige Extraktion ermöglicht die Erfassung verschiedener GUI-Eigenschaften auf unterschiedlichen Abstraktionsebenen, wobei die einzelnen Stufen charakteristische Merkmale der GUI-Elemente extrahieren. Die exakte Funktion einzelner Neuronen oder Schichten kann dabei nicht deterministisch bestimmt werden, da neuronale Netze als verteilte Systeme arbeiten und Merkmale durch das Zusammenspiel vieler Neuronen entstehen.

Eine beispielhafte Funktionsweise der unterschiedlichen Featureebenen könnte folgendermaßen aussehen:
Niedrigstufige Features erfassen grundlegende visuelle Elemente wie Kanten und Texturen. Mittlere Abstraktionsebenen identifizieren komplexere Strukturen und GUI-Komponenten. Hochstufige Features repräsentieren semantische Beziehungen zwischen GUI-Elementen. Also beispielsweise die Position eines Buttons relativ zu einem Textfeld und die gemeinsame Zugehörigkeit zu einem Formular.

Die extrahierten Feature bilden die Grundlage für die nachfolgende Aktionsauswahl des RL-Agenten und die Berechnung der Belohnungssignale. Die Feature-Repräsentation ermöglicht die präzise Unterscheidung verschiedener GUI-Zustände und unterstützt die systematische Exploration der Anwendungsoberfläche.

**Integrierter Feature Extractor**

Im Gegensatz zum YOLO-basierten Ansatz implementiert der integrierte Feature Extractor eine direkte Extraktion aus den Pixeldaten. Die Architektur nutzt die ersten Schichten des Multi-Layer-Perceptrons des PPO-Agenten für die Feature-Extraktion. Diese Integration in das RL-Modell ermöglicht ein End-to-End-Training der Featureextraktion zusammen mit der Aktionsauswahl.

Die Eingabepixel werden direkt durch diese Schichten verarbeitet, wodurch die Verbesserung der Featureextraktion während des RL-Trainings erfolgt. Diese direkte Integration bietet mehrere Vorteile:

- Adaptivität der Feature-Extraktion durch kontinuierliche Anpassung während des RL-Trainings.
- Reduktion der Modellkomplexität durch Vermeidung separater Vorverarbeitungsschritte.
- Direkte Anpassung der Featureextraktion für die spezifische Aufgabe der GUI-Exploration.

Die extrahierten Features werden direkt für die Aktionsauswahl des Agenten verwendet. Im Gegensatz zum YOLO-basierten Ansatz lernt das System selbstständig relevante Features für die GUI-Interaktion zu identifizieren, ohne auf vortrainierte Modelle angewiesen zu sein. Dies ermöglicht eine flexiblere Anpassung an verschiedene GUI-Strukturen, erfordert jedoch eine längere initiale Trainingsphase bzw. mehr Steps zur Konvergenz.

**Verkleinerter Observationspace**

Eine weitere Verbesserung der Feature-Extraktion erfolgt durch die Reduktion des Observationsraums durch systematische Vorverarbeitung der Bildschirminhalte.

Die ursprünglichen RGB-Bilder werden auf eine einheitliche Größe von 84×84 Pixeln skaliert und in Graustufenbilder konvertiert da dies die Effizienz des Trainingsprozesses signifikant verbessern kann (V. B und M. David 2015; Ma und Yuan 2019). Dadurch sinkt auch der Speicherbedarf der Feature-Extraktion. Diese Dimensionsreduktion erfolgt durch bilineare Interpolation und RGB-zu-Graustufen-Konvertierung mittels OpenCV[78]. Die Pixelwerte werden im uint8-Format normalisiert, um eine stabile Verarbeitung durch die Vermeidung von numerischen Instabilitäten zu gewährleisten.

Das Training kann sowohl mit den ursprünglichen RGB-Bildern als auch mit reduzierten Graustufenbildern erfolgen.

#### 7.4.1.2 Aktionsmodellierung

Die Aktionsmodellierung implementiert eine vereinfachte Schnittstelle für GUI-Interaktionen in einem normalisierten Koordinatenraum. Die Implementierung reduziert die komplexen Touch-Events der AndroidEnv-Umgebung auf zweidimensionale Tap-Events, die durch den RL-Agenten gesteuert werden.

Der Aktionsraum wird als kontinuierlicher 2D-Raum mit normalisierten Koordinaten im Bereich [0,1] × [0,1] definiert. Die Implementierung transformiert die normalisierten Koordinaten dann in gerätespezifische Events mittels eines spezialisierten Action Wrappers.

[78] OpenCV: Open Source Computer Vision Library `https://opencv.org/`

Der dafür geschriebene `GymnasiumInterfaceWrapper` übersetzt die Aktionen des Agenten in das von AndroidEnv erwartete Format. Dieser Wrapper implementiert:

- Konvertierung der 2D-Koordinaten in Touch-Events.
- Normalisierung der Bildschirmkoordinaten.
- Integration von Debugging-Funktionen zur Validierung der Aktionsausführung.

Die Implementierung ermöglicht eine spätere Erweiterung des Aktionsraums um komplexere Touch-Gesten durch Anpassung der Wrapper-Komponente, ohne Änderungen am RL-Agenten.

#### 7.4.1.3 Belohnungssystem

Die Implementierung des Belohnungssystems kombiniert intrinsische und extrinsische Belohnungssignale zur Steuerung des Explorationsverhaltens des RL-Agenten. Das System implementiert drei verschiedene Belohnungsvarianten, die jeweils unterschiedliche Aspekte der GUI-Exploration bewerten. Die Belohnungsvarianten sind in 7.10 auf dem Belohnungspfeil verortet.

Die Gesamtbelohnungsfunktion $R(s_t, a_t, s_{t+1})$ für einen Zustandsübergang von Zustand $s_t$ zu $s_{t+1}$ durch Aktion $a_t$ wird durch eine der folgenden Kombinationen berechnet:

$$R(s_t, a_t, s_{t+1}) = \begin{cases} R_{base}(s_t, a_t, s_{t+1}) + R_{yolo}(s_t, a_t, s_{t+1}) \\ \text{für Basisvariante} \\ \\ R_{baseXscore}(s_t, a_t, s_{t+1}) + R_{yolo}(s_t, a_t, s_{t+1}) \\ \text{für Score-skalierte Variante} \end{cases} \tag{7.1}$$

**Intrinsische Basisbelohnung ($R_{base}$)**

Die intrinsische Basisbelohnung quantifiziert die Neuartigkeit eines Zustandsübergangs $(s_t, a_t, s_{t+1})$ durch Vergleich mit bisher gesehenen Zuständen. Die Implementierung nutzt eine MobileNetV2-basierte Feature-Extraktion (M. Sandler u. a. 2019) und FAISS (siehe Abschnitt 4.6.2) zur kalkulatorisch performanten Berechnung von Zustandsdistanzen (Douze u. a. 2024).

Ursprünglich wurde auch ein Belohnungssystem basierend auf der Distanz zwischen Bildschirmzuständen auf Basis der Histogrammunterschiede implementiert, welches jedoch aufgrund schlechter Ergebnisse direkt verworfen wurde.

Die Belohnung berechnet sich als:

$$R_{base}(s_t, a_t, s_{t+1}) = \frac{d_{FAISS}(s_{t+1})}{20} \tag{7.2}$$

wobei $d_{FAISS}(s_{t+1})$ die L2-Distanz zwischen dem Folgezustand $s_{t+1}$ und dem ähnlichsten bekannten Zustand im Feature-Raum repräsentiert. Die Division durch 20 dient der Skalierung des Belohnungssignals.

**Extrinsische Score-skalierte Belohnung ($R_{baseXscore}$)**

Diese Variante erweitert die Basisbelohnung durch Integration des kumulativen Episodenscores. Für einen Zustandsübergang $(s_t, a_t, s_{t+1})$ berechnet sich die Belohnung als:

$$R_{baseXscore}(s_t, a_t, s_{t+1}) = s(t) \cdot R_{base}(s_t, a_t, s_{t+1}) \tag{7.3}$$

wobei $s(t)$ den zum Zeitpunkt $t$ erreichten kumulativen Score der aktuellen Episode darstellt. Diese Skalierung verstärkt die Belohnung für erfolgreiche Interaktionssequenzen durch Berücksichtigung der bisherigen Leistung des Agenten.

**YOLO-basierte Belohnung ($R_{yolo}$)**

Die YOLO-basierte Implementierung erweitert die Basisbelohnung um eine distanzbasierte Komponente. Für einen Zustandsübergang $(s_t, a_t, s_{t+1})$ berechnet sich die YOLO-basierte Belohnung als:

$$R_{yolo}(s_t, a_t, s_{t+1}) = R_{base}(s_t, a_t, s_{t+1}) - (p(t) \cdot \lambda \cdot d_t) \tag{7.4}$$

Die euklidische Distanz $d_t$ wird zwischen dem Touch-Punkt der Aktion $a_t = (x_{a_t}, y_{a_t})$ und dem nächstgelegenen GUI-Element im Zustand $s_t$ berechnet. Die GUI-Elemente werden durch YOLO mit ihren Bounding-Boxen erkannt, aus denen dann die Mittelpunktskoordinaten $(x_e, y_e)$ berechnet werden.

Die Distanz ergibt sich als:

$$d_t = \min_{e \in E} \sqrt{(x_{a_t} - x_e)^2 + (y_{a_t} - y_e)^2} \tag{7.5}$$

wobei $(x_{a_t}, y_{a_t})$ die Koordinaten des Touch-Punkts und $E$ die Menge aller durch YOLO erkannten GUI-Elemente mit ihren Mittelpunktskoordinaten $(x_e, y_e)$ im Zustand $s_t$ darstellt.
Der Gewichtungsfaktor $\lambda$ skaliert die Distanzkomponente in der Belohnungsfunktion. Er wurde nach empirischen Tests auf 1 gesetzt, da dieser Wert einen ausgewogenen Kompromiss zwischen der Bestrafung von ungenauen Interaktionen und der Beibehaltung einer ausreichenden Exploration ermöglicht. Größere Werte führten zu einer zu restriktiven Exploration nahe erkannter GUI-Elemente, während kleinere Werte die Bedeutung der Distanzkomponente zu stark reduzierten.

Der zeitabhängige Abschwächungsfaktor $p(t)$ reguliert die Gewichtung der Distanzkomponente über die Trainingszeit $T$ und wird als Hyperparameter vor dem Training festgelegt.

Er kann in drei Modi operieren:

$$p(t) = \begin{cases} 1 & \textit{Konstant}\text{: keine zeitliche Abschwächung,} \\ 1 - \frac{t}{T} & \textit{Linear}\text{: gleichmäßige Abschwächung,} \\ e^{-5t/T} & \textit{Exponentiell}\text{: initial starke, dann abnehmende} \\ & \text{Abschwächung.} \end{cases} \tag{7.6}$$

Die Wahl des Abschwächungsmodus erfolgt basierend auf der GUI-Komplexität und den Testzielen. Der konstante Modus wird für die meisten Anwendungsfälle benutzt, da er eine gleichmäßige Transition von präziser zu explorativer Interaktion ermöglicht. Der lineare Modus kann bei komplexen GUIs eingesetzt werden, um eine gleichmäßige Reduktion der Distanzkomponente über die Trainingszeit zu gewährleisten. Der exponentielle Modus kann bei GUIs eingesetzt werden, wo initial eine starke Führung zu bekannten Elementen benötigt wird, die später zugunsten freier Exploration reduziert werden soll.

### 7.4.2 Operationsphase

Die Operationsphase implementiert die praktische Testausführung durch den trainierten RL-Agenten. Die technische Umsetzung erfolgt durch eine ereignisgesteuerte Ausführungsschleife, die eine kontinuierliche Überwachung der Zustandsübergänge $(s_t, a_t, s_{t+1})$ gewährleistet.

#### Systeminitialisierung

Der Initialisierungsprozess umfasst die Installation der zu testenden APK über die ADB sowie die Konfiguration der Emulator-Umgebung gemäß der definierten Testparameter. Parallel erfolgt die Initialisierung der Feature-Extraktion und des Belohnungssystems. Ein definierter Ausgangszustand $s_0$ wird durch Neuinstallation der Anwendung sichergestellt.

**Testausführungsschleife**

Die zentrale Ausführungsschleife implementiert den MDP-basierten Testprozess (siehe Abschnitt 4.5). Das System extrahiert aus der AndroidEnv-Umgebung den aktuellen Zustand $s_t$ durch Erfassung des Bildschirminhalts. Daraus leitet der trainierte RL-Agent die nächste Aktion $a_t$ ab. Nach Ausführung der Aktion über die ADB-Schnittstelle wird der Folgezustand $s_{t+1}$ erfasst und die entsprechende Belohnung $R(s_t, a_t, s_{t+1})$ gemäß dem implementierten Belohnungssystem berechnet. Die Speicherung der Zustandshistorie erfolgt über Weights & Biases, um eine effiziente Nachverfolgung des Testverlaufs zu ermöglichen.

**Fehlererkennung und -dokumentation**

Das Überwachungssystem analysiert regelmäßig den Systemzustand auf Fehlerzustände. Die Implementierung kombiniert Logcat-Monitoring zur Crash-Erkennung mit einer Speicherung der dazu führenden Zustandsübergänge. Limitationen davon sind, dass nur explizite Fehlerzustände erkannt werden können und keine impliziten Fehlerzustände. Es findet im aktuellen Zustand kein Abgleich mit erwarteten Zuständen statt. Die Dokumentation der Fehler erfordert zusätzliche Maßnahmen zur detaillierten Beschreibung des Fehlers, inklusive Screenshots des fehlerhaften Zustands und der dazugehörigen Logcat-Einträge. Eine automatisierte Fehlerklassifizierung könnte die Analyse vereinfachen. Zusätzlich sollte die Reproduzierbarkeit des Fehlers dokumentiert und mit einem Capture & Replay Verfahren kombiniert werden, um die Fehlerbehebung zu erleichtern.

**Ergebnisspeicherung**

Die Dokumentation der Testresultate erfasst die Zustandsübergänge und zugehörigen Belohnungen. Das System protokolliert erkannte GUI-Elemente und Interaktionsmuster sowie detaillierte Informationen zu identifizierten Fehlerzuständen. Die hierarchische Speicherstruktur gewährleistet eine spätere Analyse der Testergebnisse.

**Parallelisierung**

Die Implementierung ermöglicht eine Parallelisierung der Testausführung durch simultanen Betrieb mehrerer Emulator-Instanzen. Jede Instanz kann in einer isolierten Umgebung operieren.

**Monitoring und Logging**

Ein zentrales Monitoring-System überwacht die Testausführung und visualisiert die Ergebnisse über die in Abschnitt 7.2 beschriebene Frontend-Schnittstelle. Diese Echtzeit-Visualisierung ermöglicht eine Bewertung der Testqualität und Explorationsstrategie des Agenten während der Operationsphase.

# 8 Evaluation des Testframeworks

Entsprechend den in Kapitel 3 (Grundlagen des Softwaretestens) eingeführten Qualitätskriterien erfolgt eine systematische Evaluation des entwickelten Testframeworks nach dem in Abschnitt 3.1.1 beschriebenen SQuaRE-Rahmenwerk. Ausgehend von der Architektur wird die Effektivität der White-Box- und Black-Box-Komponenten bei der Fehlererkennung anhand standardisierter Metriken bewertet (siehe Abschnitt 3.1.2.1).

Im Zentrum der Evaluation steht die funktionale Eignung nach ISO/IEC 25010:2023 der generierten Testfälle, da diese direkt die Fähigkeit des Systems zur Fehlererkennung und Testüberdeckung widerspiegelt. Leistungseffizienz nach ISO/IEC 25010:2023 wurde als zweite Charakteristik gewählt, weil der Ressourcenverbrauch bei KI-basierten Systemen eine kritische Rolle spielt. Wartbarkeit und Zuverlässigkeit des Frameworks wurden aufgrund ihrer Bedeutung für den praktischen Einsatz in Entwicklungsumgebungen als weitere Evaluationskriterien definiert.

Eine Bewertung der Qualität des Trainingsdatensatzes für das LLM erfolgt nach den Datenqualitätsaspekten Korrektheit, Konsistenz, Aktualität und Glaubwürdigkeit gemäß ISO/IEC 25024:2015. Zusätzlich wird analysiert, wie effektiv die implementierten Komponenten die in Abschnitt 3.1.2 definierten Fehlerarten adressieren.

Nach dem in Abschnitt 3.1.1 eingeführten SQuaRE-Rahmenwerk basiert das Evaluationskonzept auf den Qualitätsmodellen der ISO/IEC 25023:2016 und ISO/IEC 25024:2015. Anhand der in Kapitel 6 beschriebenen Architektur und ihrer in Kapitel 7 dargestellten Implementierung erfolgt eine systematische Evaluation.
Gemäß ISO/IEC 25040:2011 gliedert sich der Evaluationsprozess in fünf Phasen:
Festlegung der Evaluationsanforderungen, Spezifikation der Evaluation, Design der Evaluationsaktivitäten, Durchführung der Evaluation sowie Abschluss der Evaluation.

## 8.1 Evaluationsanforderungen

Basierend auf den in Abschnitt 1.2 definierten Zielen wurden gemäß ISO/IEC 25010:2023 *funktionale* und *nicht-funktionale* Evaluationsanforderungen bestimmt.

Zur Bewertung der funktionalen Eignung wird die Fähigkeit zur Identifikation von Fehlern nach der in Abschnitt 3.1.2 eingeführten Klassifikation sowie das Erreichen der in Abschnitt 3.2.3 definierten Testüberdeckungsmetriken evaluiert. Durch die Integration von White-Box-Tests und Black-Box-Tests nach Abschnitt 3.2.1.2 soll eine umfassende Testüberdeckung erreicht werden.

Zur Bewertung der Leistungseffizienz werden das Zeitverhalten und die Nutzung der Hardwareressourcen während der Testausführung analysiert. Dabei stehen insbesondere die Ausführungszeiten der unterschiedlichen Testgenerierungsstrategien sowie der Ressourcenverbrauch der KI-Modelle im Fokus.

Hinsichtlich der nicht-funktionalen Anforderungen wird die Wartbarkeit durch die Reproduzierbarkeit der containerisierten Testumgebung sowie die Modifizierbarkeit der generierten Testfälle sichergestellt. Zur Bewertung der Zuverlässigkeit des Systems werden die Fehlertoleranz der Testausführung und die Wiederherstellbarkeit der Testumgebung evaluiert.

Eine systematische Erfassung und Auswertung der Evaluationsergebnisse entsprechend der Qualitätsmodelle der SQuaRE-Normen ermöglicht die Bewertung der Zielerreichung sowie der erfolgreichen Integration der White-Box- und Black-Box-Test-Komponenten zu einem kohärenten Testsystem.

## 8.2 Evaluation der Entwicklungsumgebung

Eine systematische Evaluation der in Abschnitt 7.1 beschriebenen Entwicklungsumgebung erfolgt anhand der in ISO/IEC 25023:2016 definierten technischen Qualitätskriterien.

Zur Bewertung der containerisierten Entwicklungsumgebung werden die Qualitätsmerkmale nach ISO/IEC 25023:2016 herangezogen und durch quantitative Metriken untermauert.

**Portabilität (Portability):**

Bezüglich der **Adaptierbarkeit** („*Hardware environmental adaptability*“: PAd-1-G) zeichnet sich die implementierte Umgebung durch Unterstützung verschiedener Systeme mit Docker aus. Konfigurierbare Docker-Images ermöglichen einen flexiblen Wechsel zwischen GPU- und CPU-Varianten.

Für die **Installierbarkeit** („*Installation time efficiency*“: PIn-1-G, „*Ease of installation*“: PIn-2-G) sorgen vordefinierte Docker-Konfigurationen. Die genaue Installationsdauer ist abhängig von der Netzwerkgeschwindigkeit und der Größe der Base-Images bzw. der Leistung des Systems. Die Installation erfordert grundlegende Kenntnisse der Kommandozeile sowie eine funktionsfähige Docker-Installation. Der Installationsprozess umfasst drei Hauptschritte: das Klonen des Repositories, das Ausführen des Build-Skripts zur Erstellung des Docker-Images sowie das Starten des Containers. Diese Schritte sind in der Dokumentation beschrieben. Bei der GPU-Variante sind zusätzlich installierte CUDA-Treiber erforderlich.

**Wartbarkeit (Maintainability):**

Im Bereich der **Modularität** („*Coupling of components*“: MMo-1-G) existieren signifikante Abhängigkeiten zwischen den Python-ML-Frameworks und der Java-Entwicklungsumgebung. Diese manifestieren sich besonders in den gemeinsam genutzten CUDA-Bibliotheken für die GPU-Unterstützung sowie den Android SDK-Komponenten. Dies erschwert die Wartung und Erweiterung der Umgebung in ihrem aktuellen Zustand. Eine Aufteilung in separate Container würde die Modularität verbessern, erhöht jedoch den Verwaltungsaufwand.

Zur **Wiederverwendbarkeit** („*Reusability of assets*“: MRe-1-G) tragen standardisierte Base-Images und wiederverwendbare Konfigurationen bei.

Eine umfassende **Analysierbarkeit** („*System log completeness*“: MAn-1-G) wird durch ein mehrstufiges Logging-System erreicht. Die wichtigsten Systemaktionen können über das implementierte Monitoring-Interface eingesehen werden.

**Kompatibilität (Compatibility):**

Hinsichtlich der **Ko-Existenz** („*Co-existence with other products*“: CCo-1-G) ermöglicht die Containerisierung eine Ko-Existenz des Testframeworks und anderer Anwendungen auf dem gleichen System. Je nach verwendeter Hardware können jedoch Konflikte bei der gleichzeitigen Nutzung von GPU-Ressourcen auftreten.

Für die **Interoperabilität** („*Data formats exchangeability*“: CIn-1-G, „*Data exchange protocol sufficiency*“: CIn-2-G) sorgen standardisierte Datenformate (`.json`, exportierbare Unit-Tests) und Kommunikationsprotokolle (Server-Sent Events).

Im Bereich der **White-Box-Komponenten** wurde die Java-Entwicklungsumgebung erfolgreich mit Maven integriert, wobei die JUnit-Test-Ausführung innerhalb des Containers funktional umgesetzt und eine effektive Anbindung der LLM-Komponente an die Java-Entwicklungsumgebung realisiert wurde.

Die Integration der **Black-Box-Komponenten** umfasst die erfolgreiche Einbindung des Android SDK und der Emulatoren mit ADB-Verbindung über Port 5037 sowie die Integration der RL-Komponenten mit TensorFlow und der steuerbaren Umgebung mit AndroidEnv.

Die Leistungseffizienz (Performance Efficiency) der Entwicklungsumgebung wird in den jeweiligen Kapiteln 8.3 und 8.4 evaluiert.

Die Evaluation der Entwicklungsumgebung zeigt, dass die containerisierte Umgebung die in Abschnitt 6.1.1 definierten Anforderungen mit einigen Einschränkungen erfüllt und die Qualitätsmerkmale nach ISO/IEC 25023:2016 adressiert. Nicht vollständig erfüllt wurden die folgenden Qualitätsmerkmale: Die Modularität (MMo-1-G) wird durch die enge Kopplung der Komponenten eingeschränkt. Diese hohe Kopplung resultiert aus der Entscheidung, alle Komponenten in einem einzelnen Container zu integrieren. Eine Aufteilung in mehrere spezialisierte Container, orchestriert durch docker-compose[79], könnte die Wartbarkeit erhöhen und Ressourcen effizienter nutzen, wurde aber nicht umgesetzt.

Die Ko-Existenz (CCo-1-G) wird durch die gemeinsame Nutzung der GPU-Ressourcen limitiert, wenn sowohl ein LLM als auch ein RL-Modell gleichzeitig ausgeführt werden.

[79] docker-compose: ist ein Orchestrierungswerkzeug zur Definition mehrerer voneinander abhängiger Container (`https://docs.docker.com/compose/`)

Diese Einschränkungen resultieren aus der bewussten Priorisierung der Installationseffizienz und der Reproduzierbarkeit des Systems gegenüber einer strikten Modularisierung. Dies führt zwar zu einer reduzierten Wartbarkeit, gewährleistet jedoch durch den Einsatz standardisierter Container-Technologie die Portabilität zwischen verschiedenen Umgebungen.

## 8.3 Evaluation der White-Box-Test Komponente

Entsprechend der in Abschnitt 6.2 beschriebenen zweiphasigen Architektur erfolgt eine systematische Evaluation der White-Box-Test Komponente. Eine sorgfältige Bewertung der Komponente stellt eine zentrale Voraussetzung für die Beurteilung der Testgenerierungsqualität dar.

### 8.3.1 Evaluation der Trainingsphase

Die Evaluation der Trainingsphase konzentriert sich auf zwei Hauptaspekte: Die Qualität des erstellten Datensatzes nach ISO/IEC 25024:2015 sowie die Effektivität des Modelltrainings gemäß etablierter Metriken wie PPL. Eine systematische Bewertung dieser Aspekte bildet eine fundamentale Voraussetzung für die Zuverlässigkeit der Testgenerierung. Der Fokus liegt zunächst auf der grundsätzlichen Eignung des Datensatzes für das LLM-Training.

#### 8.3.1.1 Evaluation des generierten Datensatzes

Die Evaluation des in Abschnitt 6.2.2.1 beschriebenen Datensatzes erfolgt anhand der Datenqualitätskriterien nach ISO/IEC 25024:2015. Als Ausgangsbasis dient der GitHub-Code-Basisdatensatz[80], welcher durch mehrere, in Abschnitt 6.2.2.1 beschriebene und in Abschnitt 7.3.1.1 umgesetzte, Filterschritte und synthetische Datenanreicherung verbessert wurde.

[80] Github-Code-Dataset (`https://huggingface.co/datasets/codeparrot/github-code/blob/main/README.md`)

**Korrektheit („*Syntactic data accuracy*“: Acc-I-1, „*Semantic data accuracy*“: Acc-I-2)**

Zur Sicherstellung der Korrektheit des Datensatzes tragen mehrere qualitätssichernde Faktoren bei. Der ursprüngliche GitHub-Code-Basisdatensatz enthält ausschließlich vollständige Quelldateien aus aktiven Repositories. Eine implementierte dreistufige LLM-basierte Filterung gewährleistet die Qualität der Code-Test-Paare, während die synthetische Datenanreicherung durch GPT-4 und GPT-4o den Datensatz um dokumentierte Beispiele erweitert.

**Konsistenz („*Data format consistency*“: Con-I-2, „*Semantic consistency*“: Con-I-6)**

Zur Gewährleistung der Konsistenz wurden mehrere Maßnahmen implementiert. Grundlegend enthält der Basisdatensatz bereits vorgefilterte Dateien. Durch nachfolgende Filterschritte beschränkt sich der Datensatz ausschließlich auf Java-Dateien. Ein implementierter Matching-Prozess stellt die Zusammengehörigkeit von Code und Tests sicher, während die MinHash-LSH-basierte Duplikaterkennung redundante Einträge verhindert.

**Aktualität („*Update frequency*“: Cur-I-1, „*Timeliness of update*“: Cur-I-2)**

Eine hohe Aktualität des Datensatzes wird durch drei zentrale Aspekte sichergestellt. Der GitHub-Code-Datensatz basiert auf einem aktuellen Snapshot vom März 2022. Zusätzlich ergänzt die synthetische Datenanreicherung moderne Test-Patterns und aktuelle Best Practices. Durch den Einsatz aktueller LLMs (GPT-4, GPT-4o) für die Anreicherung werden aktuelle Standardkonventionen und -praktiken abgebildet.

**Glaubwürdigkeit („*Values credibility*“: Cre-I-1, „*Source credibility*“: Cre-I-2)**

Zur Sicherstellung der Glaubwürdigkeit des Datensatzes tragen drei wesentliche Eigenschaften bei. Sämtliche Quelldateien stammen aus öffentlichen GitHub-Repositories mit definierten Lizenzen. Ein implementierter Ensemble-Ansatz mit drei verschiedenen LLMs validiert die Qualität der Code-Test-Paare. Zusätzlich bleibt die Herkunft jeder Datei durch Repository-Name und Pfad nachvollziehbar dokumentiert.

Entsprechend der von Nakajima und Nakatani (2021) definierten Erweiterung des SQuaRE-Qualitätsmodells deckt sich die Evaluation des Datensatzes zusätzlich mit den Adequacy-Dimensionen. Die Feature-Extraktion durch den Ensemble-Ansatz implementiert die Data Selection Adequacy durch systematische Qualitätsfilterung. Die synthetische Datenanreicherung adressiert die Sampling Adequacy durch gezielte Ergänzung unterrepräsentierter Test-Patterns. Die Labeling Adequacy wird durch die konsistente Annotation der synthetischen Tests durch GPT-4 und GPT-4o sichergestellt. Die Target Selection Adequacy manifestiert sich in der gezielten Extraktion testbarer Code-Strukturen während des Matching-Prozesses.

Eine Gesamtbetrachtung der Evaluation des Datensatzes zeigt, dass der generierte Datensatz die grundlegenden Qualitätsanforderungen nach ISO/IEC 25024:2015 erfüllt.
Besondere Stärken manifestieren sich in der Aktualität und Nachvollziehbarkeit der Daten. Zusätzlich stellt der implementierte Qualitätssicherungsprozess durch den Ensemble-Ansatz die Eignung der Code-Test-Paare für das Training sicher. Diese Qualitätsmerkmale bilden eine solide Grundlage für das nachfolgende Training des LLM.

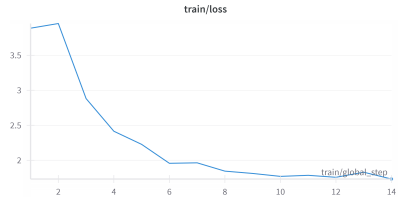
#### 8.3.1.2 Evaluation des Finetunings der feinabgestimmten Modelle

Eine Evaluation der Modelltrainings erfolgt durch systematische Analyse der aufgezeichneten Trainings- und Evaluierungsmetriken. Zentrale Aspekte der Bewertung bilden der Trainings- und Evaluierungsverlust sowie die Entwicklung der Lernrate während des Trainings und die daraus abgeleitete PPL (siehe Abschnitt 4.4.9.1).

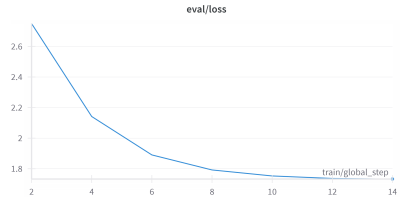
Der Trainingsprozess wurde wie in Abschnitt 7.3.1.2 beschrieben durchgeführt. Sämtliche Metriken wurden durch das Weights & Biases Monitoring-System aufgezeichnet.

**TestGen v2.1 mit CodeGemma-7B: TestGen-CodeGemma**

Der Trainingsverlauf des CodeGemma-Modells ist in den Abbildungen 8.1a, 8.1b und 8.2 dargestellt.




**(a)** Trainingsverlust TestGen-CodeGemma




**(b)** Evaluierungsverlust TestGen-CodeGemma

**Abbildung 8.1:** Trainings- und Evaluierungsverlust des CodeGemma-Modells

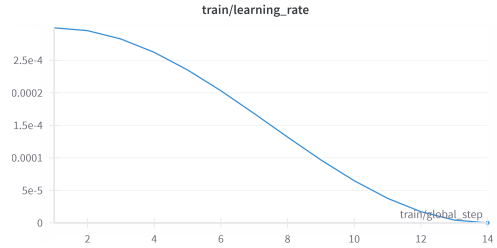


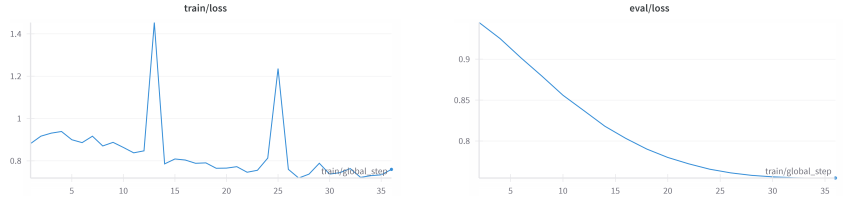

**Abbildung 8.2:** Lernratenverlauf TestGen-CodeGemma

Beim Evaluierungsverlust zeigt sich eine stetige Verbesserung von initial 2,7 auf final 1,7, was einer relativen Verbesserung von 37% entspricht. Aus diesem Evaluierungsverlust resultiert eine finale PPL von 5,47. Die erreichte Perplexität stellt ein quantitatives Maß für die Vorhersagegenauigkeit des Modells dar. Ein niedrigerer Perplexitätswert indiziert dabei eine höhere Konfidenz des Modells bei der Vorhersage der nächsten Token in der Sequenz, was für die Generierung syntaktisch korrekten Testcodes essentiell ist. Im Kontext der Testgenerierung ermöglicht eine niedrige Perplexität eine präzisere Abbildung der erlernten Testmuster auf neue Codekontexte.

In der initialen Warmup-Phase, die 3% der Trainingsschritte umfasst (`warmup_ratio=0.03`), steigt die Lernrate linear an. Nach etwa 10 Trainingsschritten konvergiert der Trainingsverlust bei einem Wert von 1,8 und zeigt keine Anzeichen von Überanpassung.

### TestGen v2.2 mit Llama-3.1-8B: TestGen-Llama

Die Trainingsmetriken des Llama-3.1-8B-Modells sind in den Abbildungen 8.3a, 8.3b und 8.4 dargestellt.




**(a)** Trainingsverlust TestGen-Llama **(b)** Evaluierungsverlust TestGen-Llama

**Abbildung 8.3:** Trainings- und Evaluierungsverlust TestGen-Llama

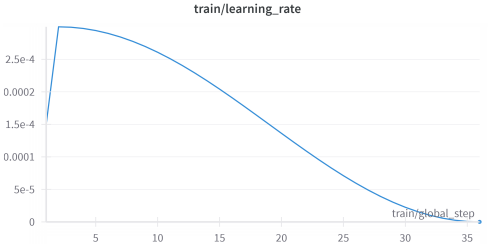



**Abbildung 8.4:** Lernratenverlauf des Llama-3.1-Modells

Beim Training des Llama-3.1-Modells zeigt sich eine graduelle Verbesserung des Evaluierungsverlusts von 0,92 auf 0,76, was in einer finalen Perplexität von 2,14 resultiert. Diese niedrigere Perplexität indiziert eine höhere Vorhersagegenauigkeit im Vergleich zum CodeGemma-Modell. Der Trainingsverlust weist zwei charakteristische Spitzen bei den Trainingsschritten 15 und 25 auf, stabilisiert sich jedoch anschließend bei etwa 0,75. Eine implementierte Lernratenanpassung verhindert eine zu schnelle Konvergenz in lokale Optima.

**Vergleichende Analyse**

Beide Modelle zeigen unterschiedliche Charakteristika im Trainingsverlauf. Das CodeGemma-Modell erreicht eine schnellere initiale Konvergenz, benötigt jedoch mehr Trainingsschritte. Im Gegensatz dazu zeigt das Llama-3.1-Modell eine stabilere Lernkurve mit weniger Volatilität im Evaluierungsverlust. Aufgrund der unterschiedlichen Modellarchitekturen, Vokabulargrößen und Tokenisierungsstrategien sind die absoluten Verlustwerte nicht direkt vergleichbar.

Die unterschiedlichen Perplexitätswerte der Modelle (CodeGemma: 5,47, Llama: 2,14) korrelieren mit der beobachteten Qualität der generierten Tests. Das Llama-Modell zeigt durch seine niedrigere Perplexität eine stabilere und zuverlässigere Testgenerierung, was sich in der höheren syntaktischen Korrektheit der erzeugten Tests widerspiegelt aber nicht in der Kompilierbarkeit. Diese Korrelation zwischen Perplexität und Testqualität unterstreicht die Bedeutung der Modellmetriken für die praktische Anwendbarkeit in der automatisierten Testgenerierung.

Durch implementierte Optimierungsstrategien wie Gradient Checkpointing und die maximale Gradientennorm von 0,3 wird die Stabilität beider Trainingsläufe gewährleistet. Eine kontinuierliche Aufzeichnung der Metriken durch Weights & Biases ermöglicht eine detaillierte Analyse der Trainingsverläufe.

### 8.3.2 Evaluation der Operationsphase

Die Evaluation der Operationsphase fokussiert auf die Qualität der generierten Tests und die Effektivität der Testgenerierung. Die Bewertung erfolgt anhand der in Abschnitt 3.2.3 definierten Metriken zur Codeüberdeckung und der Qualität der generierten Testfälle.

#### 8.3.2.1 Evaluation der Prompting-Strategien

Die Evaluation der Prompting-Strategien für die White-Box-Test-Komponente basiert auf den in Abschnitt 7.3.2.2 beschriebenen Implementierungen. Anhand der generierten Testqualität, gemessen durch die in Abschnitt 3.2.3 eingeführten Metriken zur Codeüberdeckung, erfolgt eine systematische Bewertung.

Eine systematische Evaluation erfolgte durch Variation von Prompt-Engineering-Techniken, Sampling-Temperaturen und Eingabeformaten. Basierend auf den Erkenntnissen der studentischen Abschlussarbeit von Lars Bissinger (Lars Bissinger 2024) wurde gezeigt, dass Prompt Engineering die Codeüberdeckung signifikant steigern kann. Maßgeblich beeinflussen dabei sowohl die Wahl der spezifischen Technik als auch die Sampling-Temperatur des LLM die Qualität der generierten Tests.

Bezüglich der Eingabeformate zeigten die Untersuchungen, dass eine Bereitstellung des vollständigen Quellcodes allein suboptimale Ergebnisse erzeugt. Als effektivster Ansatz erwies sich eine fokussierte Kombination aus Methodenkörper und zugehöriger JavaDoc-Dokumentation. Diese Beobachtung korrespondiert mit den Erkenntnissen der studentischen Abschlussarbeit von Jacob Hoffmann (2023), wonach LLMs bei der Verarbeitung präzise strukturierter Informationen bessere Resultate erzielen. Durch präzise dokumentierte Methodensignaturen und JavaDoc-Kommentare kann das LLM sowohl explizite als auch implizite funktionale Anforderungen ableiten. Diese Erkenntnis wird auch durch die studentische Abschlussarbeit von Tamino Ludwig (2024) bestätigt, der die Bedeutung einer präzisen Dokumentation für die automatisierte Testgenerierung hervorhebt.

Eine detaillierte Analyse verschiedener Prompting-Strategien in Kombination mit unterschiedlichen Sampling-Temperaturen wurde im Rahmen einer studentischen Abschlussarbeit (Lars Bissinger 2024) durchgeführt und ist in Tabelle 8.1 dargestellt.

| Prompt | Temperatur 0,0 | | Temperatur 0,5 | |
|---|---|---|---|---|
| Default | LC: 66,2%,<br>SC: 66,4%,<br>BC: 66,7% | Fehler: 5 | LC: 50,4%,<br>SC: 48,3%,<br>BC: 46,7% | Fehler: 5 |
| CoT-Detail | LC: 40,0%,<br>SC: 36,2%,<br>BC: 33,3% | Fehler: 5 | LC: 65,6%,<br>SC: 65,5%,<br>BC: 65,6% | Fehler: 5 |
| Role-CoT-1S | LC: 66,7%,<br>SC: 66,7%,<br>BC: 58,3% | Fehler: 5 | LC: 48,5%,<br>SC: 47,7%,<br>BC: 40,0% | Fehler: 7 |
| 1S | LC: 66,2%,<br>SC: 66,4%,<br>BC: 58,3% | Fehler: 5 | LC: 65,7%,<br>SC: 65,6%,<br>BC: 57,2% | Fehler: 5 |
| CoT-1S | LC: 33,3%,<br>SC: 33,3%,<br>BC: 25,0% | Fehler: 10 | LC: 30,5%,<br>SC: 29,4%,<br>BC: 21,1% | Fehler: 9 |

LC = Zeilenabdeckung, SC = Anweisungsabdeckung, BC = Zweigabdeckung

**Tabelle 8.1:** Evaluation verschiedener Prompting-Strategien (Llama 3 70B)

Aus den Ergebnissen in Tabelle 8.1 lassen sich signifikante Unterschiede in der Effektivität der verschiedenen Strategien ableiten. Während der Default-Prompt bei niedriger Temperatur (0,0) die beste durchschnittliche Codeüberdeckung bei stabiler Fehlerrate erreicht, zeigt die CoT-Variante eine deutliche Temperaturabhängigkeit. Bei einer Temperatur von 0,0 fällt die Überdeckung gering aus, steigt jedoch bei 0,5 merklich an. Dies deutet darauf hin, dass komplexere Prompting-Strategien von einer erhöhten „kreativen Freiheit“ des Modells profitieren können, allerdings mit dem Risiko inkonsistenter Ergebnisse.

Die One-Shot-Strategie zeichnet sich durch konsistente Ergebnisse über verschiedene Temperaturen hinweg aus. Im Gegensatz dazu weist die Kombination aus CoT und One-Shot (CoT-1S) die schwächste Leistung mit der höchsten Fehlerrate auf. Dies verdeutlicht die Herausforderungen bei der Integration mehrerer Prompting-Techniken.

Aufbauend auf diesen Erkenntnissen implementiert das Framework eine kombinierte Strategie mit Role Prompting als Basis, ergänzt durch CoT bei mehrschrittigen Testfällen. Eine niedrige Temperatur von 0,3 minimiert die Varianz in den generierten Tests, während das Role Prompting eine konsistente Testphilosophie sicherstellt.

#### 8.3.2.2 Evaluation der erstellten Tests

Eine systematische Evaluation der generierten Tests erfolgt auf Basis des Defects4J-Datensatzes, speziell der Commons CSV[81] Bibliothek. Der Aufbau orientiert sich an der in Abschnitt 7.3.2.4 beschriebenen Evaluationspipeline.

**Evaluationssetup**

Die Evaluation der White-Box-Test-Komponente implementiert einen Pairwise-Testing-Ansatz zur systematischen Abdeckung des Parameterraums. Evaluierte Parameter umfassen die Wahl zwischen methoden- und klassenbasierter Testgenerierung, die Anzahl der Reparaturzyklen sowie verschiedene Prompt-Konfigurationen. Die detaillierten Ergebnisse der Testdurchläufe mit unterschiedlichen Modellen und Parametern finden sich in den Tabellen A.1, A.2, A.3 und A.4 im Anhang.

Die Evaluationstabellen verwenden folgende Parameter und Metriken:

[81] Commons CSV: Eine Java-Bibliothek zum Lesen und Schreiben von CSV-Dateien in verschiedenen Formaten (`https://commons.apache.org/proper/commons-csv/`)

**Modellspezifikationen:**

- Model/Model Type: Bezeichnung des verwendeten LLM (lokal oder proprietär)
- Testgenerierungsstrategie:
  - Klassenbasiert: Generierung von Tests für die gesamte Klasse in einem Durchgang.
  - Methodenbasiert: Separate Testgenerierung für einzelne Methoden mit zwei Varianten:
    - Einzelausführung: Jeder generierte Test wird separat ausgeführt.
    - Klassenkonsolidierung: Die einzeln generierten Tests werden zu einer Testklasse zusammengeführt und gemeinsam ausgeführt.
- Prompt-Konfiguration: Zero-Shot/One-Shot kombiniert mit CoT, TaS sowie Umfang des bereitgestellten Quellcodes.

**Testgenerierungsmetriken:**

- Focal Classes: Anzahl der evaluierten Klassen.
- TC Created: Gesamtanzahl generierter Testfälle (Testcases).
- TC Passing: Anzahl erfolgreicher Testfälle.
- TC Failing: Anzahl fehlgeschlagener Testfälle.
- Classes Build Error: Anzahl der Klassen mit Kompilierungsfehlern durch defects4j, beispielsweise durch Syntaxfehler, inkompatible Java-Versionen oder fehlende Abhängigkeiten.

- Reparaturzyklen: Anzahl der durchgeführten Korrekturversuche (Entweder 0 oder 3 Reparaturversuche).

**Qualitätsmetriken:**

- Success Rate: $\frac{\text{TC Passing}}{\text{TC Created}}$
- Failure Rate: $\frac{\text{TC Failing}}{\text{TC Created}}$
- Line Coverage: Erreichte Zeilenüberdeckung
- Condition Coverage: Erreichte Zweigüberdeckung

**Effizienzmetriken:**

- Run Time: Durchschnittliche Ausführungszeit in Minuten
- Efficiency: $\frac{\text{TC Created}}{\text{Ausführungszeit}}$

Die Durchschnittswerte werden mit $\overline{x}$ gekennzeichnet, prozentuale Angaben mit %.

**Modellvergleich**

Die quantitative Analyse der verschiedenen Modelle zeigt signifikante Leistungsunterschiede. Die Evaluation umfasst die folgenden Modelle:

- die proprietären Modelle GPT-4o und GPT-4o-mini als Referenzpunkt
- das Open-Source-Modelle Llama 3.1 8B Instruct das als Basis für das feinabgestimmte Modell diente.
- das Open-Source-Modell CodeGemma 7B Instruct das ebenfalls feinabgestimmt wurde.
- die feinabgestimmten Varianten TestGen-Llama und TestGen-CodeGemma.

- die lokalen Modelle Microsoft Phi 4 14B und IBM Granite 3.1 8B als Vergleich da sie aktuelle Modelle darstellen, die hohe Benchmark-Ergebnisse erzielten[82].

Da die beiden feinabgestimmten Modelle keine validen Codegenerierungen ohne zusätzliches Training erzeugen, wurde TestGen-CodeGemma nicht mehr in den Tabellen A.3 und A.4 aufgeführt. Die Ausführungszeit pro Evaluationsdurchlauf schwankt zwischen knapp 6 Minuten bei der klassenbasierten Testgenerierung mit den proprietären Modelle, 36 Minuten mit lokalen Modellen und 28 Minuten bzw. 4 Stunden bei der klassenbasierten Konfiguration.

Die Ausführungsdauer wird maßgeblich durch die verwendete Hardware, die Systemauslastung und die Projektgröße beeinflusst. Die Messergebnisse zeigen dennoch deutliche Unterschiede zwischen proprietären und lokalen Modellen sowie zwischen methoden- und klassenbasierter Testgenerierung.

Die Evaluation der Testüberdeckung orientiert sich an den Metriken der ISO/IEC 25023:2016:

- Die Anweisungsüberdeckung (FCp-1-G) erreicht mit den proprietären Modellen bis zu 73%.
- Die Zweigüberdeckung (FCr-1-G) liegt bei bis zu 54%.

Die Wartbarkeit der generierten Tests wird durch zwei Hauptaspekte quantifiziert:

- Die Modifizierbarkeit (MMd-1-G, MMd-2-G) wird durch die Speicherung der Testfälle in einem standardisierten Format und die strukturierte Dokumentation der Tests gewährleistet.

[82] Huggingface *open_llm_leaderboard*: `https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard/`

- Die Analysierbarkeit (MAn-1-G) zeigt sich in der strukturierten Dokumentation der Tests und der nachvollziehbaren Testlogik durch CoT-Erklärungen.

**Leistungsanalyse**

Die Evaluationsergebnisse zeigen eine signifikante Korrelation zwischen Modellgröße und Testqualität. Modelle mit weniger als 13 Milliarden Parametern weisen substantielle Defizite bei der Generierung kompilierbaren Codes für komplexe Klassenstrukturen auf. Die feinabgestimmten lokalen Modelle TestGen-Llama und TestGen-CodeGemma erfordern ein zusätzliches Instruction Alignment für eine zuverlässige Codegenerierung was im Rahmen dieser Arbeit nicht mehr umgesetzt wurde. Ein Instruction Alignment befähigt das Modell, Aufgabenstellungen zu interpretieren und zielgerichtet umzusetzen, statt lediglich Textvervollständigungen vorzunehmen. Wurden die generierten Tests manuell betrachtet, zeigt sich, dass die lokalen Modelle nahezu korrekten aber nicht kompilierbaren Code generieren.

**Evaluation der Leistungseffizienz**

Eine systematische Bewertung der Leistungseffizienz nach ISO/IEC 25023:2016 erfolgt anhand der Metriken Zeitverhalten (Time-behaviour) und Ressourcennutzung (Resource utilization).

Das Zeitverhalten (PTb-1-G) zeigt signifikante Unterschiede zwischen den Modellvarianten. Bei der klassenbasierten Testgenerierung erreichen die proprietären Modelle eine durchschnittliche Ausführungszeit von 6 Minuten. Lokale Modelle benötigen mit 36 Minuten deutlich länger. Bei der methodenbasierten Testgenerierung steigt die Ausführungszeit auf bis zu 4 Stunden, bedingt durch die sequentielle Verarbeitung einzelner Methoden.

Die Arbeitsspeichernutzung bzw. VRAM-Nutzung (PRu-2-G) der quantisierten Modelle im GGUF-Format mit 4-Bit Quantisierung und einer maximalen Kontextgröße von 8000 Token variiert zwischen den Modellen:

- TestGen-Llama (5,8 GB) und das Basismodell Llama-3.1-8B-Instruct (5,8 GB) zeigen identische Speicheranforderungen - TestGen-CodeGemma (7,4 GB) und das Basismodell CodeGemma-7B-it (7,4 GB) weisen ebenfalls übereinstimmende Werte auf - IBM Granite-3.1-8B-Instruct benötigt 6,1 GB - Microsoft Phi-4 erfordert mit 10 GB den höchsten Speicherbedarf

Die parallele Nutzung mehrerer Testinstanzen auf einem Gerät wird primär durch den verfügbaren GPU-Speicher begrenzt. Bei der Verwendung eines Copilot+ Computers ist die Testkapazität mindestens 16 GB, was die Ausführung aller Modelle ermöglicht, jedoch die Ausführung mehrerer Instanzen gleichzeitig einschränken kann.

Die Leistungskennzahlen verdeutlichen den grundlegenden Kompromiss zwischen Testqualität und Ressourceneffizienz bei der LLM-basierten Testgenerierung. Die Evaluation zeigt, dass proprietäre Modelle bei einer durchschnittlichen Ausführungszeit von 6 Minuten nicht nur eine signifikant höhere Performanz, sondern auch eine bessere Testqualität aufweisen als lokale Modelle mit einer Ausführungszeit von 36 Minuten.
Demgegenüber steht der Vorteil der lokalen Modelle durch ihre geringeren Speicheranforderungen und die daraus resultierende verbesserte Parallelisierbarkeit. Basierend auf diesen Ergebnissen ergibt sich für produktive Entwicklungsumgebungen die Notwendigkeit einer differenzierten Einsatzstrategie der Modelle. Die initiale Testgenerierung sowie das Testen kritischer Komponenten sollten durch proprietäre Modelle erfolgen, da deren überlegene Testqualität den erhöhten Ressourcenbedarf rechtfertigt. Für Szenarien der kontinuierlichen Integration und iterativen Testverbesserung bieten sich lokale Modelle an, deren Parallelisierbarkeit und Unabhängigkeit von externen Diensten die längeren Ausführungszeiten kompensieren. Die methodenbasierte Testgenerierung sollte trotz erhöhter Laufzeit für komplexe Klassen präferiert werden, da sie eine präzisere Testüberdeckung ermöglicht. Diese differenzierte Herangehensweise verbessert die Effektivität des Testprozesses unter Berücksichtigung der verfügbaren Ressourcen und projektspezifischen Anforderungen. Eine zukünftige Erweiterung des Frameworks könnte eine automatisierte Auswahl der Teststrategie implementieren, die auf der Klassenkomplexität und den verfügbaren Ressourcen basiert.

### Einfluss der Reparaturstrategie

Die implementierte Reparaturstrategie zeigt einen signifikanten Einfluss auf die Testqualität. Die Erhöhung der Reparaturzyklen von einem auf drei verbessert die Erfolgsrate der Testgenerierung. Jedoch setzt das voraus, dass die Modelle überhaupt ausführbaren Code generieren, was bei den lokalen Modellen nicht immer der Fall ist. Der mehrstufige Reparaturmechanismus eliminiert dann syntaktische Fehler vollständig, sodass verbleibende Fehler ausschließlich in der Build-Phase auftreten. Diese manifestieren sich hauptsächlich durch inkorrekte Java-Versionsreferenzen, wie in Codebeispiel 8.1 exemplarisch dargestellt.

```
[javac] PATH/Csv/.../CSVRecordTest.java:25:
    error: diamond operator is not supported in -source 6
[javac]         mapping = new HashMap<>();
[javac]                               ^
[javac]   (use -source 7 or higher to enable diamond operator)
```

**Code 8.1:** Beispiel für einen Versionsfehler in der Build-Phase

### Vergleich der Testgenerierungsstrategien

Die methodenbasierte Testgenerierung erzeugt eine höhere Anzahl an Tests und erreicht eine bessere Überdeckung. Die Analyse der Ausführungszeiten zeigt eine signifikant höhere Laufzeit im Vergleich zur klassenbasierten Variante. Die durchschnittliche Ausführungszeit der methodenbasierten Konfiguration übertrifft die der klassenbasierten Variante deutlich.

**Konfigurationsanalyse**

Die systematische Evaluation verschiedener Prompt-Konfigurationen führt zu folgenden evidenzbasierten Erkenntnissen: Bei der Analyse der Prompt-Komplexität zeigen die Daten keine eindeutige Überlegenheit einfacher Konfigurationen. Das GPT-4o-mini Modell erreicht beispielsweise mit CoT und Quellcode-Kontext eine Erfolgsrate der Testfälle von 65,74% gegenüber 57,41% ohne CoT aber mit TaS. Die Schwankungen zwischen verschiedenen Konfigurationen desselben Modells liegen jedoch innerhalb einer Bandbreite von etwa 8 Prozentpunkten. Die klassenbasierte Testgenerierung zeigt bei GPT-4o eine deutlich kürzere Ausführungszeit (5:43 Minuten) verglichen mit der methodenbasierten Variante (42:08 Minuten) bei vergleichbarer Testüberdeckung (64,55% vs. 68,71% Zeilenüberdeckung). Die methodenbasierte Konfiguration generiert jedoch eine höhere Anzahl an Testfällen (150 vs. 48) und erreicht damit eine detailliertere Testüberdeckung der einzelnen Funktionalitäten aber hat auch eine größere Anzahl an Klassen für die es keine Testfälle generieren konnte.
Lokale Modelle wie testgen-llama-3.1-8b zeigen durchgehend schwache Ergebnisse mit hohen Build-Error-Raten, während proprietäre Modelle wie GPT-4o hohe Testüberdeckungen erreichen. Diese Diskrepanz verdeutlicht die aktuelle technologische Überlegenheit der proprietären Modelle bei der Testgenerierung.

**Praktische Implikationen**

Die Evaluationsergebnisse zeigen eine deutliche Diskrepanz zwischen proprietären und lokalen Modellen. Die proprietären Modelle erreichen eine durchschnittliche Codeüberdeckung von bis zu 73%, was nahe an der von Mockus, Nagappan und Dinh-Trong (2009) empfohlenen 80%-Grenze liegt. Diese Grenze wird als Indikator für eine ausreichende Testüberdeckung betrachtet, die es Entwicklern ermöglicht, sich auf die verbleibenden komplexeren Testfälle zu konzentrieren, die spezifisches Domänenwissen erfordern. Die automatisch generierten Tests bilden somit eine solide Grundlage, auf der Entwickler gezielt aufbauen können, um eine vollständige Testüberdeckung zu erreichen.

Die Wahl zwischen methoden- und klassenbasierter Testgenerierung erfordert eine projektspezifische Abwägung zwischen Überdeckung und Ausführungszeit. Die beobachtete hohe Varianz in der Testqualität zwischen verschiedenen Testläufen mit identischen Parametern unterstreicht die Notwendigkeit multipler Testiterationen.

Die lokalen Modelle zeigen trotz guter Trainingsergebnisse (gemessen an der PPL) in der praktischen Evaluation deutliche Schwächen. Dies liegt hauptsächlich an den strengen Anforderungen des Java-Compilers, die von kleineren Modellen nicht zuverlässig erfüllt werden können. Die implementierte Reparaturstrategie eliminiert zwar syntaktische Fehler, dennoch treten Build-Fehler auf, wenn das Modell Code generiert. Einige lokale Modelle generieren ohne zusätzliches Training keinen kompilierbaren bzw. ausführbaren Code. Eine weitere Feinabstimmung und vor allem ein *Instruction Alignment* der lokalen Modelle könnte deren Leistung verbessern.

Diese Ergebnisse verdeutlichen die aktuelle Eignung der White-Box-Test-Komponente primär als „automatisches“ Agentensystem in Verbindung mit proprietären Modellen. Sollen lokale Modelle eingesetzt werden, ist eine manuelle Nachbearbeitung notwendig, um die generierten Tests zu validieren und zu reparieren, was es eher als Assistenzsystem für Entwickler qualifiziert. Die automatisierte Generierung der Basistests reduziert den manuellen Aufwand signifikant und ermöglicht eine effizientere Nutzung der Entwicklerressourcen für komplexere Testszenarien.

## 8.4 Evaluation der Black-Box-Test-Komponente

Die Evaluation der Black-Box-Test-Komponente konzentriert sich auf die Trainingsphase des Agenten und Operationsphase anhand der in Abschnitt 6.3.3 definierten Qualitätskriterien. Diese Ausrichtung basiert auf den praktischen Anforderungen des GUI-Testens mobiler Applikationen.

### 8.4.1 Evaluation der Trainingsphase

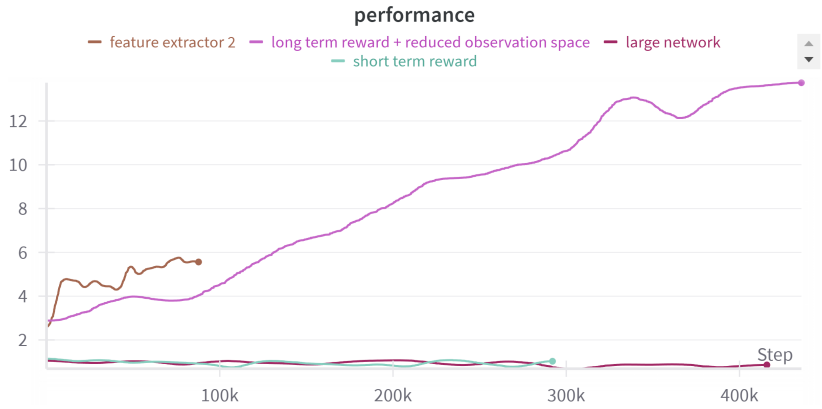



**Abbildung 8.5:** Vergleich der Trainingsverläufe für verschiedene Konfigurationen des RL-Agenten

Im Rahmen der Evaluation wurden die Trainingsverläufe verschiedener RL-Agenten-Konfigurationen analysiert. Abbildung 8.5 zeigt die Performance-Entwicklung über 400.000 Trainingsschritte für vier unterschiedliche Konfigurationen. Als Performance-Metrik dient die Anzahl korrekter Button-Interaktionen innerhalb eines definierten Zeitfensters. Eine korrekte Button-Interaktion wird definiert als erfolgreiche Identifikation und Aktivierung eines GUI-Elements, die zu einer messbaren Zustandsänderung in der Anwendung führt. Diese Messung erfolgt durch die Analyse der Zustandsübergänge und der resultierenden Belohnungssignale.

**Feature-Extraktion und Belohnungsmodell**

Die Evaluation vergleicht vier unterschiedliche Konfigurationen der in Abschnitt 7.4 implementierten Feature-Extraktions-Varianten und Belohnungsmodelle:

- **Integrierter Feature Extractor mit Basisbelohnung („*short term reward*“):** Implementiert die direkte Featureextraktion durch das Multi-Layer-Perceptron des PPO-Agenten mit der intrinsischen Belohnung $R_{base}$ basierend auf FAISS-Distanzen. Der Beobachtungsraum verwendet die ursprüngliche Bildgröße von 640×320×3 Pixeln.
- **Integrierter Feature Extractor mit Score-skalierter Belohnung und reduziertem Beobachtungsraum („*long term reward + reduced observation space*“):** Nutzt die gleiche Feature-Extraktion, erweitert jedoch die Belohnung durch $R_{baseXscore}$ mit dem kumulativen Episodenscore. Der Beobachtungsraum wird auf 84×84×1 Pixel reduziert.
- **Integrierter Feature Extractor mit Score-skalierter Belohnung („*large network*“):** Verwendet die gleiche Konfiguration wie „*long term reward + reduced observation space*“, jedoch ohne Reduktion des Beobachtungsraums.

- **YOLO-basierte Extraktion mit kombinierter Belohnung („*feature extractor 2*“):** Implementiert die YOLO-Featureextraktion mit zusätzlicher distanzbasierter Belohnungskomponente $R_{yolo}$ ohne reduziertem Beobachtungsraum.

Der Ansatz „*feature extractor 2*“ mit YOLO-basiertem Feature-Extraktor zeigt initial eine schnelle Leistungssteigerung auf einen Performance-Wert von etwa 4,5. Nach etwa 100.000 Trainingsschritten stagniert die Leistung jedoch und weist vor allem Instabilitäten auf, weshalb keine längeren Trainingsverläufe möglich waren. Diese Beobachtung deutet darauf hin, dass der spezialisierte Feature-Extraktor zwar das initiale Lernen beschleunigt, jedoch die langfristige Exploration möglicherweise durch zu starke Fokussierung auf erkannte GUI-Elemente einschränkt.

**Beobachtungsraum und Belohnungsstruktur**

Die Konfiguration „*long term reward + reduced observation space*“ demonstriert die beste Performance mit einem kontinuierlichen Anstieg auf einen Wert von über 13.
Diese Variante profitiert von der Kombination eines reduzierten Beobachtungsraums von 84×84×1 Pixeln und der Score-skalierten Belohnungsfunktion $R_{baseXscore}$. Die Dimensionsreduktion von ursprünglich 614.400 auf 7.056 Dimensionen verbessert die Effizienz des Trainingsprozesses.

Im Kontrast dazu zeigt „*short term reward*“ mit ausschließlich intrinsischer Belohnung $R_{base}$ und vollem Beobachtungsraum eine Stagnation bei einem Wert nahe 1. Auch „*large network*“ mit Score-skalierter Belohnung $R_{baseXscore}$ aber ohne Dimensionsreduktion erreicht lediglich die selbe Performance wie ein rein zufallsbasierter Agent.

Diese Ergebnisse verdeutlichen die Bedeutung einer angemessenen Dimensionsreduktion des Beobachtungsraums. Die Evaluationsergebnisse korrespondieren mit den in Abschnitt 6.3 definierten Anforderungen an die Black-Box-Test-Komponente. Die erreichte Performance der besten Konfiguration demonstriert die Fähigkeit des Systems zur GUI-Exploration und erfüllt die in Abschnitt 6.3.3 definierten Qualitätskriterien für die systematische Testdurchführung.

**Erweiterungsmöglichkeiten**

Der YOLO-basierte Ansatz könnte durch weitere Anpassungen, wie in der studentischen Abschlussarbeit von Benjamin Meyjohann (2022) vorgeschlagen, verbessert werden. In dieser Arbeit wurde gezeigt, dass ein Mask-RCNN-Modell zur automatisierten Erkennung und Positionierung von Oberflächenelementen in mobilen Anwendungen verwendet werden kann. Trainiert wurde das Modell mit automatisch generierten und annotierten Daten. Dies würde eine präzisere Elementeerkennung und damit eine gezieltere Interaktion ermöglichen.

**Parallelisierung**

Eine Parallelisierung sowohl in der Trainings- als auch in der Operationsphase wird durch die containerisierte Testumgebung und parallele Ausführung mehrerer Emulatoren ermöglicht. Mehrere Emulatoren mit unterschiedlichen Konfigurationen können während des Trainings parallel betrieben werden. Dadurch beschleunigt sich das Training durch das simultane Lernen deutlich.

### 8.4.2 Evaluation der Operationsphase

Nach erfolgreichem Training wurde der Agent auf seine Fähigkeit zur systematischen Exploration und zum Testen der GUI evaluiert. Eine systematische Bewertung erfolgte anhand der in der ISO/IEC 25023:2016 spezifizierten Metriken für Softwarequalität. Zentrale Aspekte bildeten dabei die **funktionale Eignung** zur systematischen Exploration der Benutzeroberfläche und Identifikation von Fehlerzuständen sowie die **Zuverlässigkeit** bezüglich der Robustheit der Testausführung und Wiederherstellbarkeit der Testumgebung anhand der Metrik RRe-1-G.

#### 8.4.2.1 Evaluation der Leistungseffizienz

Eine systematische Bewertung der Leistungseffizienz der Black-Box-Test-Komponente erfolgt nach den Metriken der ISO/IEC 25023:2016.

Das Zeitverhalten (PTb-1-G) der RL-basierten GUI-Tests erfolgt in Echtzeit, wobei die Testdauer primär von der Komplexität und dem Umfang der zu testenden Applikation abhängt. Die systematische Exploration der Benutzeroberfläche ermöglicht eine kontinuierliche Testausführung ohne zusätzliche Verzögerungen durch die RL-Komponente.

Bezüglich der Ressourcennutzung (PRu-2-G) benötigt jede Emulator-Instanz zwischen 2 und 4 GB Arbeitsspeicher. Diese Speicheranforderung ermöglicht einen effizienten Einsatz der verfügbaren Systemressourcen. Der Ressourcenbedarf bleibt während der Testausführung konstant, da keine zusätzlichen Modelle geladen werden müssen.

Die Testkapazität (PCa-1-G) wird hauptsächlich durch den verfügbaren Arbeitsspeicher begrenzt. Bei einem System mit ausreichend RAM können unter Berücksichtigung des Betriebssystem-Overheads und der containerisierten Testumgebung mehrere Emulator-Instanzen parallel betrieben werden. Diese Parallelisierung ermöglicht eine effiziente Testdurchführung auf verschiedenen Android-Versionen oder Gerätekonfigurationen.

Die implementierte Parallelisierung in Kombination mit der Echtzeitausführung der Tests resultiert in einer hohen Gesamteffizienz der Black-Box-Test-Komponente. Die moderate Ressourcenanforderung der einzelnen Emulator-Instanzen ermöglicht dabei eine flexible Skalierung der Testkapazität entsprechend der verfügbaren Systemressourcen.

#### 8.4.2.2 Empirische Evaluation

Für eine systematische Bewertung wurden Tests mit der in Abschnitt 4.6.1 beschriebenen Vokram-App durchgeführt. Als Grundlage dienten definierte Qualitätskriterien der ISO/IEC 25023:2016.

**Evaluationsumgebung**

Sämtliche Tests fanden in der in Abschnitt 6.1.1 beschriebenen containerisierten Testumgebung statt. Zum Einsatz kamen Android-Emulatoren mit API-Level 34 (Android 14) und einer Bildschirmauflösung von 1280×720 Pixeln. Entsprechend der in Abschnitt 7.4.1.1 beschriebenen Vorverarbeitung erfolgte eine Skalierung auf 640×360 Pixel für den RL-Agenten oder 80x80 Pixeln in Graustufen im reduzierten Beobachtungsraum.

**Parallelisierung**

Durch implementierte Parallelisierungsmechanismen aus Abschnitt 7.4 lassen sich Tests gleichzeitig auf mehreren Emulator-Instanzen ausführen. Dieser Ansatz verkürzt nicht nur die Gesamttestzeit, sondern ermöglicht auch eine umfassendere Abdeckung verschiedener Android-Versionen und Bildschirmauflösungen.

**Funktionale Eignung**

Im Fokus der Evaluation stand die systematische GUI-Exploration durch den RL-Agenten. Basierend auf den Ergebnissen aus Abschnitt 8.4.1 führt der Agent korrekte Interaktionen durch. Neue Zustände werden durch die implementierte FAISS-basierte Zustandsanalyse aus Abschnitt 7.4.1.1 effektiv erkannt und dokumentiert.

**Zuverlässigkeit**

Anhand der in Abschnitt 6.3.3 definierten Kriterien wurde die Wiederherstellbarkeit (RRe-1-G) evaluiert. Die implementierte Fehlerbehandlung aus Abschnitt 7.4 gewährleistet eine automatische Neuinstallation der Testapplikation nach kritischen Fehlern oder abgeschlossenen Episoden. Eine Episode ist dabei ein Trainingsdurchlauf des RL-Agenten, der durch eine fest definierte Anzahl von Schritten, eine definierte Anzahl von Interaktionen mit der Applikation oder einem zeitlichen Limit begrenzt ist. Eine systematische Dokumentation erfolgt durch die integrierte ADB-Schnittstelle, die sowohl Logcat-Monitoring als auch Screenshot-Erfassung umfasst. Zusätzlich ermöglicht die Darstellung des aktuellen Zustands im Frontend eine visuelle Überprüfung der Testausführung.

Insgesamt zeigen die Evaluationsergebnisse eine zuverlässige GUI-Exploration und robuste Testausführung durch die Black-Box-Test-Komponente. Die implementierte Parallelisierung und Fehlerbehandlung garantieren dabei einen effizienten und zuverlässigen Testprozess.

## 8.5 Evaluation des gesamten Testframeworks

Eine systematische Evaluation des entwickelten Testframeworks erfolgt anhand der in Abschnitt 1.2 definierten Zielsetzungen und orientiert sich an den Qualitätsmerkmalen der ISO/IEC 25023:2016. Durch Integration der in Abschnitt 6.2 und 6.3 evaluierten White-Box- und Black-Box-Komponenten werden die in Abschnitt 3.1.2 klassifizierten Fehlerarten systematisch adressiert.

**Kombinierte Testüberdeckung**

Mit proprietären Modellen erreicht die White-Box-Test-Komponente eine Anweisungsüberdeckung von bis zu 73% und eine Zweigüberdeckung von 54%. Diese Werte reichen nahe an die von Mockus, Nagappan und Dinh-Trong (2009) empfohlene 80%-Grenze für die Anweisungsüberdeckung heran und übertreffen die Überdeckung bisheriger Methoden wie beispielsweise von (Tufano, Drain u. a. 2020).

Ergänzend dazu ermöglicht die Black-Box-Test-Komponente eine systematische GUI-Exploration mit direkter Interaktion mit der Benutzeroberfläche. Durch diese unabhängig voneinander operierenden Komponenten wird eine umfassende Qualitätssicherung auf Code- und Benutzerebene gewährleistet.

**Systemarchitektur und Modularität**

Eine modulare Bauweise des Frameworks ermöglicht entsprechend der in Abschnitt 6.1.1 definierten Anforderungen flexible Anpassungen und Erweiterungen der Funktionalität. Durch eine sprachenunabhängige Konzeption lässt sich die White-Box-Test-Komponente um zusätzliche Programmiersprachen und Modelle erweitern. Der Trainingsdatensatz für das Feinabstimmen der Unit-Test-Generierung kann systematisch ausgebaut und weiterverwendet werden, während das Frontend Erweiterungsmöglichkeiten für zusätzliche Visualisierungen und Analysewerkzeuge bietet. Die containerisierte Architektur gewährleistet eine nahtlose Integration in CI/CD-Pipelines.

**Fehlererkennungskapazität**

Definierte Fehlerarten werden durch den zweistufigen Ansatz adressiert. Auf Codeebene identifiziert die LLM-basierte Unit-Test-Generierung Fehler und Defekte mit einer nachgewiesenen Erfolgsrate von bis zu 63%. Parallel dazu ermöglicht die RL-basierte GUI-Exploration die Erkennung von Störungen und resultierenden Ausfällen auf Benutzerebene durch systematische Zustandsanalyse mittels FAISS-Distanzmetriken.

**Ressourceneffizienz**

Signifikante Unterschiede zeigen sich zwischen lokalen und proprietären Modellen bezüglich Ausführungszeit und Testqualität. Proprietäre Modelle erreichen eine hohe Testqualität bei geringer Ausführungszeit von etwa 6 Minuten, während lokale Modelle zwischen 36 Minuten und 4 Stunden bei geringerer Testqualität benötigen. Bezüglich des Speicherbedarfs erfordern die quantisierten lokalen Modelle zwischen 5,8 GB (TestGen-Llama) und 10 GB (Microsoft Phi-4) VRAM. Eine obere Grenze für den Speicherbedarf der verwendbaren lokalen Modelle hängt von den lokal verfügbaren Ressourcen ab. Die containerisierte Ausführungsumgebung ermöglicht gemäß Abschnitt 7.1 eine effiziente Parallelisierung beider Komponenten.

**Grenzen und Verwendungsmöglichkeiten**

Technische Limitationen zeigen sich primär bei lokalen Modellen mit weniger als 13 Milliarden Parametern. Diese generieren in ihrer aktuellen Form keine zuverlässig kompilierbaren Tests und eignen sich daher primär als Assistenzsystem statt als autonomes Testsystem. Entwickler, die sich mit den lokalen Modellen Tests generieren lassen, sollten daher auf eine manuelle Überprüfung der generierten Tests achten.
Für die RL-Komponente zeigt sich ein erhöhter initialer Trainingsbedarf für effektive GUI-Exploration. Die YOLO-basierte Feature-Extraktion reduziert zwar die initial notwendigen Trainingsschritte und adressiert das Cold-Start-Problem, weist jedoch Instabilitäten bei längerer Ausführung auf. Fortschreitende Entwicklungen leistungsfähigerer Modelle und Hardwarekomponenten adressieren diese Limitationen zunehmend.

Anwendungsmöglichkeiten des Frameworks erstrecken sich über den mobilen Kontext hinaus. Ein Transfer der White-Box-Komponente auf Unit-Test-Generierung in verschiedenen Programmierdomänen erscheint durch die modulare Struktur realisierbar. Konzeptionell kann das Framework als Referenzarchitektur für weitere KI-gestützte Testsysteme dienen. Die Integration unterschiedlicher Modelle wird entsprechend spezifischer Anforderungen und Verfügbarkeiten ermöglicht.

Für die RL-Komponente sollte eine Konfiguration mit reduziertem Beobachtungsraum und langfristiger Belohnungsstruktur gewählt werden und das Training auf weitere Applikationen ausgeweitet werden.

**Entwicklungspotenziale und Zukunftsausblick**

Das Framework bietet vielfältige Entwicklungsmöglichkeiten. Ein Instruction Alignment könnte die Code-Generierung lokaler Modelle verbessern, während eine Integration von Mask-RCNN entsprechend Benjamin Meyjohann (2022) den YOLO-basierten Feature-Extraktor verbessern könnte. Ausbaumöglichkeiten bestehen bei der Testgenerierung für weitere Programmiersprachen, einem Ausbau des Trainingsdatensatzes für präzisere Unit-Test-Generierung für unterschiedliche Programmiersprachen sowie dem Frontend durch zusätzliche Analysetools und Visualisierungen.

Die Evaluationsergebnisse demonstrieren die Effektivität des hybriden Testansatzes bei gleichzeitiger Aufzeigung klarer Einsatzgrenzen. Modulare Architektur und systematische Konzeption ermöglichen gezielte Erweiterungen und Verbesserungen. Mit fortschreitender Entwicklung leistungsfähigerer LLMs und Rechenressourcen wird eine Steigerung der automatisierten Testgenerierungsqualität erwartet. Somit bietet das entwickelte Konzept eine fundierte Grundlage für zukünftige KI-gestützte Testframeworks.

# 9 Zusammenfassung und Ausblick

Die steigende Bedeutung mobiler Applikationen im Alltag vieler Menschen führt zu erhöhten Qualitätsanforderungen durch Industriestandards wie ISO/IEC 25010:2023 und regulatorische Vorgaben. Dies zeigt sich besonders in sicherheitskritischen Bereichen wie Finanzdienstleistungen und Gesundheitswesen, wo Softwarefehler schwerwiegende Konsequenzen haben können. Daher erweist sich eine systematische Entwicklung innovativer Ansätze zur automatisierten Softwarequalitätssicherung als essentiell. Mit der Konzeption und Implementierung eines KI-gestützten Testframeworks leistet diese Arbeit einen grundlegenden Beitrag zur systematischen Integration von White-Box-Tests - und Black-Box-Tests -Tests. Eine zentrale Innovation stellt dabei die Kombination von LLMs zur automatisierten Unit-Test-Generierung mit RL-basierten Methoden zur systematischen Exploration von Benutzeroberflächen dar.

Das Framework implementiert für beide KI-Komponenten einen zweiphasigen Ansatz, bestehend aus einer Trainings- und einer Operationsphase. In der Trainingsphase erfolgt die Feinabstimmung der LLM-Komponente durch einen spezialisierten Datensatz zur Verbesserung der Testgenerierungsqualität hinsichtlich Codeüberdeckung und Fehlererkennung. Parallel dazu wird die RL-Komponente durch ein hybrides Belohnungssystem für die effiziente GUI-Exploration trainiert, wobei der Fokus auf der systematischen Überdeckung der Benutzeroberfläche und der Identifikation von Fehlerzuständen liegt. In der anschließenden Operationsphase ermöglichen die trainierten Modelle eine automatisierte Testdurchführung mit geringer manueller Intervention.

Zur Gewährleistung reproduzierbarer Testbedingungen dient eine containerisierte Architektur, welche flexibel verschiedene Modelle integriert. Eine systematische Evaluation nach den Standards der SQuaRE-Normenreihe belegt die Effektivität des Ansatzes. Mit proprietären Modellen erreicht die White-Box-Komponente eine Anweisungsüberdeckung von bis zu 73%. Gleichzeitig demonstriert der trainierte RL-Agent in der Black-Box-Komponente eine systematische Exploration der Benutzeroberfläche.

## 9.1 Ergebnisse und Beiträge

Im Rahmen dieser Arbeit wurde ein KI-gestützter Testansatz für mobile Applikationen entwickelt, der erstmalig LLM-basierte Unit-Test-Generierung mit RL-gestützter GUI-Exploration in einer integrierten Architektur kombiniert. Aufbauend auf den ursprünglichen Forschungszielen ermöglicht dieser Ansatz eine umfassende Adressierung der in Abschnitt 3.1.2 klassifizierten Fehlerarten durch die systematische Kombination von White-Box-Tests - und Black-Box-Tests. Im Vergleich zu existierenden Ansätzen, die sich meist auf einzelne Testebenen fokussieren, bietet das Framework eine ganzheitliche Lösung zur Qualitätssicherung mobiler Applikationen.

Als technische Basis dient eine containerisierte Testinfrastruktur, welche reproduzierbare Testbedingungen sowie eine standardisierte Ausführungsumgebung gewährleistet. Durch den modularen Aufbau wird eine nahtlose Integration in bestehende CI/CD-Pipelines sowie eine flexible Erweiterung um zusätzliche Testkomponenten ermöglicht. Eine Evaluation nach ISO/IEC 25023:2016 bestätigt die Erfüllung der definierten Qualitätsanforderungen bezüglich Portabilität, Wartbarkeit und Kompatibilität.

Für die Unit-Test-Generierung wurde ein mehrstufiger Ansatz in der LLM-basierten White-Box-Komponente implementiert. Als Grundlage dient ein GitHub-Code-Korpus, der durch systematische Filterung sowie synthetische Datenanreicherung nach den Qualitätskriterien der ISO/IEC 25024:2015 verbessert wurde. Evaluationen mittels des Defects4J-Frameworks zeigen eine direkte Korrelation zwischen Modellgröße und Testqualität.
Diese Korrelation ist jedoch nicht linear ausgeprägt und kleinere Modelle erzielen in einigen Fällen bessere Ergebnisse als größere Modelle, und ein Ende der Verbesserungen durch die Erhöhung der Modellgröße zeichnet sich ab (C. Wu und Tang 2024).

Mit proprietären Modellen wurde eine Anweisungsüberdeckung von 73% und eine Zweigüberdeckung von 54% erreicht. Eine implementierte Reparaturstrategie reduziert syntaktische Fehler in den generierten Tests nahezu vollständig. Durch die nichtdeterministische Natur der Modelle ergeben sich jedoch Einschränkungen in der Testreproduzierbarkeit und es können zufällige syntaktische Fehler auftreten. Als Einschränkung erweisen sich lokale Modelle mit weniger als 13 Milliarden Parametern, welche für komplexe Klassenstrukturen nur bedingt geeignet sind und primär als Assistenzsysteme fungieren können.

Bei der RL-basierten Black-Box-Komponente wurde eine systematische Exploration der Benutzeroberfläche realisiert. Empirische Evaluationen verschiedener Konfigurationen belegen die Effektivität eines reduzierten Beobachtungsraums in Kombination mit einer langfristigen Belohnungsstruktur. Der trainierte Agent führt zielgerichtete Interaktionen mit der zu testenden App durch. Obwohl eine YOLO-basierte Feature-Extraktion das initiale Lernen beschleunigt, besteht noch Verbesserungspotenzial bezüglich der Langzeitstabilität des Systems.

Zur Integration aller Testkomponenten wurde ein Frontend nach dem MVC-Muster auf Flask-Basis entwickelt. Die Implementierung umfasst eine Echtzeitüberwachung der Testausführung durch Server-Sent Events sowie eine Visualisierung der Testergebnisse. Eine SQLite-basierte Datenpersistenz ermöglicht die systematische Dokumentation der Testüberdeckung und identifizierter Fehler. Die erstellten Unit-Tests können über das Frontend exportiert und in bestehende, externe, CI/CD-Pipelines integriert werden.

Mit diesen Entwicklungen leistet die Arbeit einen substantiellen Beitrag zur Integration von KI-Methoden in der Softwarequalitätssicherung und etabliert ein anwendungsorientiertes theoretisches Fundament für die Kombination verschiedener KI-Ansätze im Softwaretesting. Das entwickelte Framework demonstriert die grundsätzliche praktische Umsetzbarkeit dieser Konzepte und bildet eine solide Basis für weitere Forschung im Bereich KI-gestützter Testautomatisierung. Für Entwickler und Unternehmen bietet das System praktische Vorteile durch die automatisierte Generierung von Basistests, wodurch manuelle Ressourcen für komplexere Testszenarien freigesetzt werden.

## 9.2 Ausblick

Das entwickelte Framework eröffnet vielfältige Forschungsperspektiven für die Weiterentwicklung der automatisierten Qualitätssicherung mobiler Applikationen und anderer Softwaresysteme. Die zentralen Entwicklungsmöglichkeiten gliedern sich in methodische, technische und theoretische Bereiche. Dabei ist das grundlegende Konzept des Frameworks auf andere Softwaresysteme übertragbar, erfordert jedoch Anpassungen. Die Prompting-Strategien müssen für die jeweiligen Technologien angepasst und die Build- sowie Testprozesse entsprechend konfiguriert werden.

Die LLM-basierte Testgenerierung kann durch Integration spezialisierter Teststrategien wie Äquivalenzklassenanalyse und Grenzwertanalysen erweitert werden. Eine Verbesserung der Prompting-Strategien mit Fokus auf der Verwendung einer minimalen Anzahl von Testfällen könnte eine Steigerung der Testeffektivität ermöglichen.

Reinforcement Learning from Human Feedback (RLHF)-Techniken für die Feinabstimmung ermöglichen zukünftig eine präzisere Anpassung an domänenspezifische Testanforderungen (Banghua u. a. 2023). Denkbar wäre auch die Verwendung einer Reinforcement Learning durch Compiler Feedback (RLCF)-Technik, die eine gezielte Anpassung der Testgenerierung durch Feedback des Compilers ermöglicht.
Die Integration domänenspezifischer Pretrainingsmethoden für unterschiedliche Programmiersprachen sowie automatisierte Reparaturstrategien für fehlgeschlagene Tests können die Testqualität weiter verbessern.
Fortschritte in der Modellquantisierung in Kombination mit leistungsfähigerer Hardware versprechen zudem eine stetige Verbesserung lokaler Modelle (Tseng u. a. 2024).

Im Bereich der RL-basierten GUI-Tests bietet die Integration multimodaler Modelle Potenzial für eine verbesserte Oberflächenanalyse. Strukturierte Analyseansätze von UI-Screenshots könnten die Präzision der Interaktionsentscheidungen erhöhen (Y. Lu u. a. 2024).
Die Erweiterung des Aktionsraums um komplexere Touch-Gesten wie Multi-Touch oder Swipe-Bewegungen sowie die Integration von Computer-Vision-Techniken versprechen eine effizientere GUI-Element-Erkennung und Exploration.

Die technische Weiterentwicklung fokussiert sich auf die nahtlose Integration in CI/CD-Pipelines sowie die Optimierung der Containerarchitektur für verbesserte Ressourcennutzung. Die Integration weiterer Testebenen wie Integrations- und Systemtests sowie die Implementierung von Capture & Replay-Funktionalitäten ermöglichen eine umfassendere Fehlerdokumentation.
Eine Erweiterung des Frameworks auf weitere Plattformen wie Smartwatches, Tablets oder zukünftige Systeme wie Smart Glasses erfordert die Anpassung der Trainingsdatensätze und GUI-Explorationsmethoden an weitere Programmiersprachen und Interaktionsmöglichkeiten, ist aber aufgrund der modularen Struktur und inhärenten Generalisierbarkeit des Frameworks grundsätzlich realisierbar.
Dazu wäre eine Anpassung des Trainingsdatensatzes auf die dafür verwendeten Programmiersprachen für die LLM-Komponente und eine Anpassung der GUI-Exploration für die neuen Interaktionsmöglichkeiten notwendig.

Praktische Evaluationen in industriellen Umgebungen können wertvolle Erkenntnisse für die zielgerichtete Weiterentwicklung liefern.

# Literaturverzeichnis


Abdin, Marah u. a. (2024). *Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone*. arXiv: 2404.14219.

Abdulla, Waleed (2017). *Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow*.

Adamo, David u. a. (2018). „Reinforcement learning for Android GUI testing“. In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ESEC/FSE '18: 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Lake Buena Vista FL USA: ACM, S. 2–8.

Albelwi, Saleh (2022). „Survey on Self-Supervised Learning: Auxiliary Pretext Tasks and Contrastive Learning Methods in Imaging“. In: *Entropy* 24.4. S. 551.

Alshahwan, Nadia u. a. (2024). *Automated Unit Test Improvement using Large Language Models at Meta*. arXiv: 2402.09171.

Alshammari, Abdulrahman u. a. (2024). *230,439 Test Failures Later: An Empirical Evaluation of Flaky Failure Classifiers*. arXiv: 2401.15788.

Amalfitano, Domenico u. a. (2012). „Using GUI ripping for automated testing of Android applications“. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE'12: IEEE/ACM International Conference on Automated Software Engineering. Essen Germany: ACM, S. 258–261.

Andreas Spillner und Tilo Linz (2021). *Software Testing Foundations : A Study Guide for the Certified Tester Exam- Foundation Level- ISTQB® Compliant*. [S.l.]: dpunkt.verlag.

Apple (2015). *Core Services Layer*. URL: https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CoreServicesLayer/CoreServicesLayer.html (besucht am 19. 02. 2025).

– (2024). *Xcode*. Apple Developer. URL: https://developer.apple.com/xcode/ (besucht am 19. 02. 2025).

Arteca, Ellen u. a. (2022). „Nessie: automatically testing JavaScript APIs with asynchronous callbacks“. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. New York, NY, USA: Association for Computing Machinery, S. 1494–1505.

Baechler, Gilles u. a. (2024). *ScreenAI: A Vision-Language Model for UI and Infographics Understanding*. arXiv: 2402.04615.

Baker, Bowen u. a. (2020). *Emergent Tool Use From Multi-Agent Autocurricula*. arXiv: 1909.07528[cs,stat].

Baldacchino, Tara u. a. (2016). „Variational Bayesian mixture of experts models and sensitivity analysis for nonlinear dynamical systems“. In: *Mechanical Systems and Signal Processing* 66-67, S. 178–200.

Balzert, Helmut (1998). *Lehrbuch der Softwre-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Lehrbücher der Informatik Band 2. Heidelberg: Spektrum Akad. Verl.

Banghua, Zhu u. a. (2023). *Starling-7B: Increasing LLM Helpfulness & Harmlessness with RLAIF*. URL: https://starling.cs.berkeley.edu (besucht am 29. 11. 2023).

Barto, Andrew, Richard Sutton und Chris Watkins (1989). *Learning and Sequential Decision Making*.

Bauersfeld, Sebastian und Tanja E J Vos (2012). *A Reinforcement Learning Approach to Automated GUI Robustness Testing*.

Baumgartner, Manfred u. a. (2023). *Agile testing: der agile Weg zur Qualität*. Carl Hanser Verlag GmbH Co KG.

Becker, Suzanna und Mark Plumbley (1996). „Unsupervised neural network learning procedures for feature extraction and classification“. In: *Applied Intelligence* 6.3, S. 185–203.

Bell, Jonathan u. a. (2018). „DeFlaker: Automatically detecting flaky tests“. In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18: 40th International Conference on Software Engineering. Gothenburg Sweden: ACM, S. 433–444.

Bellman, Richard (1957). „A Markovian decision process“. In: *Journal of mathematics and mechanics*. S. 679–684.

Bellman, Richard und Robert Kalaba (1957). „Dynamic programming and statistical communication theory“. In: *Proceedings of the National Academy of Sciences* 43.8. S. 749–751.

Bengio, Yoshua u. a. (2003). „A Neural Probabilistic Language Model“. In: *Journal of Machine Learning Research (JMLR)* 2003.2003.

Berry, Michael J. A. und Gordon Linoff (2004). *Data mining techniques: for marketing, sales, and customer relationship management*. 2nd ed. Indianapolis, Ind: Wiley Pub.

Bertolino, Antonia (2007). „Software Testing Research: Achievements, Challenges, Dreams“. In: *Future of Software Engineering (FOSE '07)*. Future of Software Engineering (FOSE '07), S. 85–103.

Bertsekas, Dimitri (2012). *Dynamic programming and optimal control: Volume I*. Bd. 4. Athena scientific.

Black, Sid u. a. (2022). *GPT-NeoX-20B: An Open-Source Autoregressive Language Model*. arXiv: 2204.06745.

Bommasani, Rishi u. a. (2022). *On the Opportunities and Risks of Foundation Models*. arXiv: 2108.07258.

Boyarshinov, Victor (2005). *Machine learning in computational finance*. Rensselaer Polytechnic Institute.

Bratko, Andrej u. a. (2006). „Spam filtering using statistical data compression models“. In: *The Journal of Machine Learning Research* 7. S. 2673–2698.

Brown, Tom u. a. (2020). „Language Models are Few-Shot Learners“. In: *Advances in Neural Information Processing Systems* 33, S. 1877–1901.

Busch, Kiran u. a. (2023). *Just Tell Me: Prompt Engineering in Business Process Management*. arXiv: 2304.07183.

Carbonell, Jaime G., Ryszard S. Michalski und Tom M. Mitchell (1983). „An Overview of Machine Learning“. In: *Machine Learning: An Artificial Intelligence Approach*. Hrsg. von Ryszard S. Michalski, Jaime G. Carbonell und Tom M. Mitchell. Symbolic Computation. Berlin, Heidelberg: Springer, S. 3–23.

Céspedes, Daniel u. a. (2020). „Software product quality in DevOps contexts: A systematic literature review“. In: *Trends and Applications in Software Engineering: Proceedings of the 8th International Conference on Software Process Improvement (CIMPS 2019)*. Springer, S. 51–64.

Chen, Mark u. a. (2021). *Evaluating Large Language Models Trained on Code*. arXiv: 2107.03374.

Chen, Qifeng, Jia Xu und Vladlen Koltun (2017). „Fast Image Processing With Fully-Convolutional Networks“. In: Proceedings of the IEEE International Conference on Computer Vision, S. 2497–2506.

Chen, Yinghao u. a. (2024). „ChatUniTest: A Framework for LLM-Based Test Generation“. In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. FSE 2024. New York, NY, USA: Association for Computing Machinery, S. 572–576.

Choudhary, Shauvik Roy, Alessandra Gorla und Alessandro Orso (2015). „Automated test input generation for android: Are we there yet?(e)“. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, S. 429–440.

Chowdhary, Prof (2020). *Fundamentals of Artificial Intelligence*.

Chung, Junyoung u. a. (2014). *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*.

Cohn, Mike (2010). *Succeeding with agile: Software development using Scrum*. The Addison-Wesley signature series A Mike Cohn signature book. Upper Saddle River, NJ: Addison-Wesley.

Coppola, Riccardo, Maurizio Morisio und Marco Torchiano (2017). „Scripted GUI Testing of Android Apps: A Study on Diffusion, Evolution and Fragility“. In: *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE. New York, NY, USA: Association for Computing Machinery, S. 22–32.

Curtis, S.A. (2003). „The classification of greedy algorithms“. In: *Science of Computer Programming* 49.1, S. 125–157.

Daka, Ermira und Gordon Fraser (2014). „A Survey on Unit Testing Practices and Problems“. In: *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, S. 201–211.

Das, Sumit u. a. (2015). „Applications of Artificial Intelligence in Machine Learning: Review and Prospect“. In: *International Journal of Computer Applications* 115, S. 31–41.

Dehlinger, Josh und Jeremy Dixon (2011). „Mobile Application Software Engineering: Challenges and Research Directions“. In: *Workshop on Software Engineering for Mobile Application Development* 2.

Delia, Lisandro u. a. (2015). „Multi-platform mobile application development analysis“. In: *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS)*. 2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS). S. 181–186.

Deng, Jia u. a. (2009). „ImageNet: A large-scale hierarchical image database“. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009 IEEE Conference on Computer Vision and Pattern Recognition. S. 248–255.

Deng, Li, Geoffrey Hinton und Brian Kingsbury (2013). „New types of deep neural network learning for speech recognition and related applications: an overview“. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013 IEEE International Conference on Acoustics, Speech and Signal Processing. S. 8599–8603.

Dettmers, Tim, Mike Lewis u. a. (2022). „8-bit Optimizers via Block-wise Quantization“. In: *International Conference on Learning Representations*. ICLR.

Dettmers, Tim, Artidoro Pagnoni u. a. (2023). *QLoRA: Efficient Finetuning of Quantized LLMs*. arXiv: 2305.14314.

Devlin, Jacob, Ming-Wei Chang und Kristina Toutanova (2019). „Bert: Pre-training of deep bidirectional transformers for language understanding“.

In: *Proceedings of naacL-HLT*. NAACL-HLT 2019. Bd. 1. Minneapolis, Minnesota, S. 4171–4186.

Dolotta, T. A. u. a. (1976). „The LEAP load and test driver“. In: *Proceedings of the 2nd international conference on Software engineering*. ICSE '76. Washington, DC, USA: IEEE Computer Society Press, S. 182–186.

Douze, Matthijs u. a. (2024). „The Faiss library“. In:

Dubey, Shiv Ram, Satish Kumar Singh und Bidyut Baran Chaudhuri (2022). *Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark*. arXiv: 2109.14545.

Eck, Moritz u. a. (2019). „Understanding flaky tests: the developer's perspective“. In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, S. 830–840.

El Haji, Khalid, Carolin Brandt und Andy Zaidman (2024). „Using GitHub Copilot for Test Generation in Python: An Empirical Study“. In: *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)*. AST '24. New York, NY, USA: Association for Computing Machinery, S. 45–55.

Engelen, Jesper E. van und Holger H. Hoos (2020). „A survey on semi-supervised learning“. In: *Machine Learning* 109.2, S. 373–440.

European Union (2024). *Artificial Intelligence Act (Regulation (EU) 2024/1689)*. Regulation 2024/1689. European Union.

Everingham, M. u. a. (2012). *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*.

Faragó, David (2023). „Engineering A Reliable Prompt For Generating Unit Tests - Prompt engineering for QA & QA for prompt engineering“. In: *Softwaretechnik-Trends Band 43, Heft 3*. Gesellschaft für Informatik e.V.

Flora, Harleen, Xiaofeng Wang und Swati Chande (2014). „An Investigation on the Characteristics of Mobile Applications: A Survey Study“. In: *International Journal of Information Technology and Computer Science* 6.11. S. 21–27.

Frank, Mikhail u. a. (2014). „Curiosity driven reinforcement learning for motion planning on humanoids“. In: *Frontiers in Neurorobotics* 7.

Fraser, Gordon und Andrea Arcuri (2013). „Evosuite: On the challenges of test case generation in the real world“. In: *2013 IEEE sixth international conference on software testing, verification and validation*. IEEE, S. 362–369.

Fraser, Steven u. a. (2003). „Test Driven Development (TDD)“. In: *Extreme Programming and Agile Processes in Software Engineering*. Hrsg. von Michele Marchesi und Giancarlo Succi. Berlin, Heidelberg: Springer, S. 459–462.

Frister, Demian, Aleksandar Goranov und Andreas Oberweis (2021). „Industrieroboter testet Apps“. In: *German Testing Magazin* (Janaur 2021).

Frister, Demian, Andreas Oberweis und Aleksandar Goranov (2020). „Automated Testing of Mobile Applications Using a Robotic Arm“. In: *CSCI*. 2020 International Conference on Computational Science and Computational Intelligence (CSCI). IEEE, S. 1729–1735.

Frister geb. Hartmann, Demian (2019). „Sensor Integration with ZigBee Inside a Connected Home with a Local and Open Sourced Framework: Use Cases and Example Implementation“. Las Vegas, NV, USA.

Fu, Zihao u. a. (2023). *Decoder-Only or Encoder-Decoder? Interpreting Language Model as a Regularized Encoder-Decoder*. arXiv: 2304.04052.

Fujimoto, Scott, Herke van Hoof und David Meger (2018). „Addressing Function Approximation Error in Actor-Critic Methods“. In: *Proceedings of the 35th International Conference on Machine Learning*. Hrsg. von Jennifer Dy und Andreas Krause. Bd. 80. Proceedings of Machine Learning Research. PMLR, S. 1587–1596.

Gao, Leo u. a. (2020). *The Pile: An 800GB Dataset of Diverse Text for Language Modeling*.

Glorot, Xavier, Antoine Bordes und Yoshua Bengio (2011). „Deep Sparse Rectifier Neural Networks“. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. JMLR Workshop und Conference Proceedings, S. 315–323.

Goadrich, Mark H. und Michael P. Rogers (2011). „Smart smartphone development: iOS versus android“. In: *Proceedings of the 42nd ACM technical symposium on Computer science education*. SIGCSE '11. New York, NY, USA: Association for Computing Machinery, S. 607–612.

Goodfellow, Ian, Yoshua Bengio und Aaron Courville (2016). *Deep Learning*. MIT Press.

Google (2023). *UI/Application Exerciser Monkey | Android Studio*. Android Developers. URL: `https://developer.android.com/studio/test/other-testing-tools/monkey` (besucht am 25. 07. 2024).

– (2024a). *Espresso | Android Developers*. URL: `https://developer.android.com/training/testing/espresso?hl=de` (besucht am 24. 07. 2024).

– (2024b). *Leitfaden zur App-Architektur*. Android Developers. URL: `https://developer.android.com/topic/architecture?hl=de` (besucht am 20. 07. 2024).

– (2024c). *Plattformarchitektur | Platform*. Android Developers. URL: `https://developer.android.com/guide/platform?hl=de` (besucht am 20. 07. 2024).

– (2024d). *UI-Tests mit Espresso Test Recorder erstellen | Android Studio*. Android Developers. URL: `https://developer.android.com/studio/test/other-testing-tools/espresso-test-recorder?hl=de` (besucht am 24. 07. 2024).

Gu, Shixiang u. a. (2016). „Continuous Deep Q-Learning with Model-based Acceleration“. In: *Proceedings of The 33rd International Conference on Machine Learning*. Hrsg. von Maria Florina Balcan und Kilian Q. Weinberger. Bd. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, S. 2829–2838.

Guilherme, Vitor und Auri Vincenzi (2023). „An initial investigation of ChatGPT unit test generation capability“. In: *Proceedings of the 8th Brazilian Symposium on Systematic and Automated Software Testing*. SAST '23. New York, NY, USA: Association for Computing Machinery, S. 15–24.

Haarnoja, Tuomas u. a. (2018). „Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor“. In: *International conference on machine learning*. PMLR, S. 1861–1870.

Hamlet, Richard (1994). „Random testing“. In: *Encyclopedia of software Engineering* 2. S. 971–978.

Hastie, Trevor, Robert Tibshirani und Jerome H. Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer.

He, Kaiming u. a. (2016). „Deep Residual Learning for Image Recognition“. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). S. 770–778.

Herculano-Houzel, Suzana (2009). „The human brain in numbers: a linearly scaled-up primate brain“. In: *Frontiers in Human Neuroscience* 3.

Hinton, Geoffrey und Terrence J. Sejnowski (1999). *Unsupervised Learning: Foundations of Neural Computation*. MIT press.

Hochreiter, Sepp und Jürgen Schmidhuber (1997). „Long Short-term Memory“. In: *Neural computation* 9, S. 1735–80.

Hoffman, Jacob und Demian Frister (2025). „Automatische Testgenerierung mit generativer KI“. In: *German Testing Magazin* (Januar 2025).

Hoffmann, Jacob und Demian Frister (2024). „Generating Software Tests for Mobile Applications Using Fine-Tuned Large Language Models“. In: *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test*. AST '24. Lisbon, Portugal: Association for Computing Machinery, S. 76–77.

Hoffmann, Marc R. (2007). *Code Coverage Analysis for Eclipse*.

Hu, Edward J. u. a. (2021). *LoRA: Low-Rank Adaptation of Large Language Models*. arXiv: 2106.09685.

Huggingface (2022). *Training a causal language model from scratch - Hugging Face NLP Course*. URL: https://huggingface.co/learn/nlp-course/chapter7/6 (besucht am 20. 03. 2024).

– (2024). *Summary of the tokenizers*. URL: https://huggingface.co/docs/transformers/en/tokenizer_summary (besucht am 14. 08. 2024).

IEEE (1990). „IEEE Standard Glossary of Software Engineering Terminology“.
– (2010). *IEEE Standard Classification for Software Anomalies*. Version 2010.
– (2014). *IEEE Standard for Software Quality Assurance Processes*.
– (2016). *IEEE Standard for System, Software, and Hardware Verification and Validation*.
Ioffe, Sergey und Christian Szegedy (2015). „Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift“. In: *Proceedings of the 32nd International Conference on Machine Learning*. Hrsg. von Francis Bach und David Blei. Bd. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, S. 448–456.
ISO (2019). *Ergonomie der Mensch-System-Interaktion Teil 210: Menschzentrierte Gestaltung interaktiver Systeme (ISO 9241-210:2019)*. Version 2019.
ISO und IEC (2008). *ISO/IEC 25012:2008 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. Version 2008.
– (2011a). *ISO/IEC 25040:2011 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Evaluation process*. Version 2011.
– (2011b). *ISO/IEC 9126:2011 Software engineering — Product quality*. Version 2011.
– (2014). *ISO/IEC 25000:2014 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE)*. Version 2014.
– (2015). *ISO/IEC 25024:2015 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Quality measurement framework*. Version 2015.
– (2016a). *ISO/IEC 25022:2016 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Quality measurement framework*. Version 2016.

– (2016b). *ISO/IEC 25023:2016 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software product quality*. Version 2016.
– (2019a). *ISO/IEC 25020:2019 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Quality measurement framework*. Version 2019.
– (2019b). *ISO/IEC 25030:2019 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Quality requirements*. Version 2019.
– (2023). *ISO/IEC 25010:2023 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. Version 2023.
– (2024). *ISO/IEC 24030:2024 - Information technology – Artificial intelligence (AI) – Use cases*.

ISO, IEC und IEEE (2017). *ISO/IEC/IEEE 24765:2017 - Systems and software engineering – Vocabulary*.
– (2019). *ISO/IEC/IEEE 15026-1 Systems and Software engineering — Concepts and vocabulary*. Version 2019.
– (2021a). *ISO/IEC/IEEE 29119-2:2021 - Software and systems engineering – Software testing – Part 2: Test Process*.
– (2021b). *ISO/IEC/IEEE 29119-4:2021 - Software and systems engineering – Software testing – Part 4: Test Techniques*.
– (2022a). *ISO/IEC/IEEE 29119-1:2022 - Software and systems engineering – Software testing – Part 1: General concepts*.
– (2022b). *ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:General concepts*.
– (2023). *ISO/IEC/IEEE 15288 Systems and software engineering — System life cycle processes*. Version 2023.
– (2024). *ISO/IEC/IEEE 24748-1 Systems and Software engineering — Life Cycle management*. Version 2024.

itsecuritycoach (2024). *CrowdStrike: Alles Wichtige zum Vorfall*. itsecuritycoach. URL: `https://www.itsecuritycoach.com/2024/08/01/crowdstrike-vorfall-alles-wichtige/` (besucht am 18. 12. 2024).

Ivanković, Marko u. a. (2024). „Productive Coverage: Improving the Actionability of Code Coverage“. In: *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* 2024, S. 58.

Jain, Ramesh, Rangachar Kasturi, Brian G Schunck u. a. (1995). *Machine vision*. Bd. 5. McGraw-hill New York.

Jakub, Cameron und Mihai Nica (2023). *Depth Degeneracy in Neural Networks: Vanishing Angles in Fully Connected ReLU Networks on Initialization*. arXiv: 2302.09712[cs,math,stat].

Jarrett, Kevin u. a. (2009). „What is the Best Multi-Stage Architecture for Object Recognition?“ In: *ICCV* 12.

Jiang, Albert Q., Alexandre Sablayrolles, Arthur Mensch u. a. (2023). *Mistral 7B*. arXiv: 2310.06825.

Jiang, Albert Q., Alexandre Sablayrolles, Antoine Roux u. a. (2024). *Mixtral of Experts*. arXiv: 2401.04088.

Joorabchi, Mona Erfani, Ali Mesbah und Philippe Kruchten (2013). „Real Challenges in Mobile App Development“. In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, S. 15–24.

Just, René, Darioush Jalali und Michael D. Ernst (2014). „Defects4J: a database of existing faults to enable controlled testing studies for Java programs“. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, S. 437–440.

Kaelbling, Leslie, Michael Littman und Andrew Moore (1996). „Reinforcement Learning: A Survey“. In: *Journal of Artificial Intelligence Research* 4.

Kalman, B.L. und S.C. Kwasny (1992). „Why tanh: choosing a sigmoidal function“. In: *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*. [Proceedings 1992] IJCNN International Joint Conference on Neural Networks. Bd. 4, 578–581 vol.4.

Kaplan, Jared u. a. (2020). *Scaling Laws for Neural Language Models*. arXiv: 2001.08361[cs,stat].

Khemka, Mansi und Brian Houck (2024). „Toward Effective AI Support for Developers: A survey of desires and concerns“. In: *Queue* 22.3, Pages 60:53–Pages 60:78.

Kingma, Diederik P. und Jimmy Ba (2015). „Adam: A Method for Stochastic Optimization“. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Hrsg. von Yoshua Bengio und Yann LeCun.

Kirillov, Alexander u. a. (2019). „Panoptic Segmentation“. In: *2019 CVPR*. Conference on Computer Vision and Pattern Recognition (CVPR). Long Beach, CA, USA: IEEE, S. 9396–9405.

Kocetkov, Denis u. a. (2022). *The Stack: 3 TB of permissively licensed source code*. arXiv: 2211.15533.

Konda, Vijay und John Tsitsiklis (1999). „Actor-Critic Algorithms“. In: *Advances in Neural Information Processing Systems*. Hrsg. von S. Solla, T. Leen und K. Müller. Bd. 12. MIT Press.

Kong, Aobo u. a. (2024). *Better Zero-Shot Reasoning with Role-Play Prompting*. arXiv: 2308.07702.

Krizhevsky, Alex, Ilya Sutskever und Geoffrey E Hinton (2012). „ImageNet Classification with Deep Convolutional Neural Networks“. In: *Advances in Neural Information Processing Systems*. Bd. 25. Curran Associates, Inc.

Kulal, Sumith u. a. (2019). „SPoC: Search-based Pseudocode to Code“. In: *Advances in Neural Information Processing Systems*. Bd. 32. Curran Associates, Inc.

Kunz, Christopher (2024). *Crowdstrike-Ausfall: Analyse zeigt trivialen Programmierfehler*. Heise. URL: https://www.heise.de/ (besucht am 18. 12. 2024).

Laurençon, Hugo u. a. (2022). „The BigScience ROOTS Corpus: A 1.6TB Composite Multilingual Dataset“. In: *Advances in Neural Information Processing Systems*. Hrsg. von S. Koyejo u. a. Bd. 35. Curran Associates, Inc., S. 31809–31826.

LeCun, Yann, Yoshua Bengio und Geoffrey Hinton (2015). „Deep learning“. In: *Nature* 521.7553. S. 436–444.

Li, Raymond u. a. (2023). *StarCoder: may the source be with you!* arXiv: 2305.06161.

Li, Yujia u. a. (2022). „Competition-level code generation with AlphaCode“. In: *Science (New York, N.Y.)* 378.6624, S. 1092–1097.

Li, Zhenmin u. a. (2006). „Have things changed now?: an empirical study of bug characteristics in modern open source software“. In: *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ASPLOS06: Architectural Support for Programming Languages and Operating Systems. San Jose California: ACM, S. 25–33.

Liang, Chieh-Jan u. a. (2014). „Caiipa: Automated large-scale mobile app testing through contextual fuzzing“. In: *Proceedings of the Annual International Conference on Mobile Computing and Networking, MOBICOM*.

Liggesmeyer, Peter (1990). *Modultest und Modulverifikation: state of the art*. BI-Wiss.-Verlag.

– (2009). *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. 2. Heidelberg: Spektrum Akademischer Verlag.

Lin, Tsung-Yi u. a. (2014). „Microsoft COCO: Common Objects in Context“. In: *Computer Vision – ECCV 2014*. Hrsg. von David Fleet u. a. Cham: Springer International Publishing, S. 740–755.

Linares-Vásquez, Mario u. a. (2017). „How do Developers Test Android Applications?“ In: *2017 IEEE International Conference on Software Maintenance and Evolution*. ICSME, S. 613–622.

Lingampally, Raghu, Atul Gupta und Pankaj Jalote (2007). „A Multipurpose Code Coverage Tool for Java“. In: *2007 40th Annual Hawaii International Conference on System Sciences*. HICSS. 261b–261b.

Liu, Baozheng u. a. (2020). „FANS: Fuzzing Android Native System Services via Automated Interface Analysis“. In: *SEC'20: Proceedings of the 29th USENIX Conference on Security Symposium*. Conference on Security Symposium.

Liu, Chien Hung u. a. (2014). „Capture-Replay Testing for Android Applications“. In: *2014 International Symposium on Computer, Consumer and Control*. 2014 International Symposium on Computer, Consumer and Control, S. 1129–1132.

Liu, Pei u. a. (2024). „Understanding the quality and evolution of Android app build systems“. In: *Journal of Software: Evolution and Process* 36.5. e2602.

Liu, Pengfei u. a. (2023). „Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing“. In: *ACM Computing Surveys* 55.9, 195:1–195:35.

Liu, Xiao u. a. (2021). „Self-supervised learning: Generative or contrastive“. In: *IEEE Transactions on Knowledge and Data Engineering* 35.1. S. 857–876.

Liu, Yiheng u. a. (2024). *Understanding LLMs: A Comprehensive Overview from Training to Inference*. arXiv: 2401.02038.

Llama Team, AI @ Meta (2024). „The Llama 3 Herd of Models“. In.

Lu, Lu u. a. (2020). *Dying ReLU and Initialization: Theory and Numerical Examples*. arXiv: 1903.06733.

Lu, Yadong u. a. (2024). *OmniParser for Pure Vision Based GUI Agent*. arXiv: 2408.00203.

Ludewig, Jochen und Horst Lichter (2013). *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. 3., korrigierte Aufl. Heidelberg: dpunkt.verl.

Luo, Qingzhou u. a. (2014). „An empirical analysis of flaky tests“. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. New York, NY, USA: Association for Computing Machinery, S. 643–653.

Ma, Ji und Yuyu Yuan (2019). „Dimension reduction of image deep feature using PCA“. In: *Journal of Visual Communication and Image Representation* 63, S. 102578.

Marvin, Ggaliwango u. a. (2024). „Prompt Engineering in Large Language Models“. In: S. 387–402.

Memon, Atif M; Martha E; Pollack und Mary Lou Soffa (1999). „Using a goal-driven approach to generate test cases for GUIs“. In: International Conference on Software Engineering: ICSE 99. Hrsg. von Barry Boehm, David Garlan und Jeff Kramer. ACM, S. 257–266.

Microsoft (2024). *Copilot+ PC hardware requirements - Microsoft Support*. URL: https://support.microsoft.com/en-us/topic/copilot-pc-hardware-requirements-35782169-6eab-4d63-a5c5-c498c3037364 (besucht am 20. 02. 2025).

Miillerburg, M (1970). „Systematic Stepwise Testing: A Method For Testing Large Complex Systems“. In: *WIT Transactions on Information and Communication Technologies* 14.

Miller, Barton P., Mengxiao Zhang und Elisa R. Heymann (2022). „The Relevance of Classic Fuzz Testing: Have We Solved This One?“ In: *IEEE Transactions on Software Engineering* 48.6, S. 2028–2039.

Min, Sewon u. a. (2022). *Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?* arXiv: 2202.12837.

Mitchell, T.M. (1997). *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill.

Mockus, Audris, Nachiappan Nagappan und Trung T. Dinh-Trong (2009). „Test coverage and post-verification defects: A multiple case study“. In: *International Symposium on Empirical Software Engineering and Measurement*. ESEM. Bd. 3. S. 291–301.

Mondal, Saikat, Suborno Deb Bappon und Chanchal K. Roy (2024). „Enhancing User Interaction in ChatGPT: Characterizing and Consolidating Multiple Prompts for Issue Resolution“. In: *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. 2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR). S. 222–226.

Montavon, Grégoire, Wojciech Samek und Klaus-Robert Müller (2018). „Methods for interpreting and understanding deep neural networks“. In: *Digital Signal Processing* 73, S. 1–15.

Muccini, Henry, Antonio Di Francesco und Patrizio Esposito (2012). „Software testing of mobile applications: Challenges and future research directions“. In: *2012 7th International Workshop on Automation of Software Test (AST)*, S. 29–35.

Muldal, Alistair u. a. (2019). *dm_env: A Python interface for reinforcement learning environments*.

Myers, Glenford J. (1979). *The Art of Software Testing*.

Myers, Glenford J., Tom Badgett und Corey Sandler (2012). *The Art of Software Testing*. 3. Aufl. Hoboken, NJ: Wiley.

Nair, Vinod und Geoffrey E. Hinton (2010). „Rectified linear units improve restricted boltzmann machines“. In: *Proceedings of the 27th International Conference on Machine Learning*. ICML. Bd. 10. ICML'10. Madison, WI, USA: Omnipress, S. 807–814.

Nakajima, Shin und Takako Nakatani (2021). „AI Extension of SQuaRE Data Quality Model“. In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C). S. 306–313.

Negara, Stas, Naeem Esfahani und Raymond Buse (2019). „Practical Android Test Recording with Espresso Test Recorder“. In: *IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP, S. 193–202.

OpenAI u. a. (2023). *GPT-4 Technical Report*. arXiv: 2303.08774.

Ouédraogo, Wendkûuni C. u. a. (2024). *Large-scale, Independent and Comprehensive study of the power of LLMs for test case generation*. arXiv: 2407.00225.

Ouyang, Long u. a. (2022). „Training language models to follow instructions with human feedback“. In: *Advances in Neural Information Processing Systems* 35, S. 27730–27744.

Pan, Minxue u. a. (2020). „Reinforcement learning based curiosity-driven testing of Android applications“. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, S. 153–164.

Pan, Sinno Jialin und Qiang Yang (2010). „A Survey on Transfer Learning“. In: *IEEE Transactions on Knowledge and Data Engineering* 22.10, S. 1345–1359.

Papineni, Kishore u. a. (2002). „Bleu: a Method for Automatic Evaluation of Machine Translation“. In: *Proceedings of the 40th Annual Meeting of the*

*Association for Computational Linguistics*. ACL 2002. Hrsg. von Pierre Isabelle, Eugene Charniak und Dekang Lin. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, S. 311–318.

Parry, Owain u. a. (2022). „A Survey of Flaky Tests“. In: *ACM Transactions on Software Engineering and Methodology* 31.1, S. 1–74.

Patel, Priyam u. a. (2018). „On the effectiveness of random testing for Android“. In: *Proceedings of the 13th International Workshop on Automation of Software Test*. AST. Bd. 13.

Pathak, Deepak u. a. (2017). „Curiosity-driven Exploration by Self-supervised Prediction“. In: *Proceedings of the 34th International Conference on Machine Learning*. ICML. S. 2778–2787.

Pecorelli, Fabiano u. a. (2021). „Software testing and Android applications: a large-scale empirical study“. In: *Empirical Software Engineering* 27.2, S. 31.

Qualcomm (2024). *Snapdragon X Elite Product Brief*.

Radford, Alec, Karthik Narasimhan u. a. (2018). *Improving Language Understanding by Generative Pre-Training*. OpenAI. URL: `https://api.semanticscholar.org/CorpusID:49313245`.

Radford, Alec, Jeffrey Wu, Dario Amodei u. a. (2019). *Better language models and their implications*. OpenAI blog. URL: `https://openai.com/index/better-language-models/`.

Radford, Alec, Jeffrey Wu, Rewon Child u. a. (2019). *Language Models are Unsupervised Multitask Learners*. URL: `https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`.

Raffel, Colin u. a. (2020). „Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer“. In: *Journal of Machine Learning Research*. JMLR. Bd. 21.

Raffin, Antonin u. a. (2021). „Stable-Baselines3: Reliable Reinforcement Learning Implementations“. In: *Journal of Machine Learning Research* 22.268, S. 1–8.

Redmon, Joseph u. a. (2016). „You Only Look Once: Unified, Real-Time Object Detection“. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition*. CVPR. S. 779–788.

Reenskaug, Trygve (1979). „A note on DynaBook requirements“. In: *Xerox PARC*.

Ren, Shuo u. a. (2020). *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis*. arXiv: 2009.10297.

Reynolds, Laria und Kyle McDonell (2021). *Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm*. arXiv: 2102.07350.

Ridnik, Tal, Dedy Kredo und Itamar Friedman (2024). *Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering*. arXiv: 2401.08500.

Romdhana, Andrea u. a. (2022). „Deep Reinforcement Learning for Black-box Testing of Android Apps“. In: *ACM Transactions on Software Engineering and Methodology* 31.4, 65:1–65:29.

Rumelhart, David E., Geoffrey E. Hinton und Ronald J. Williams (1986). „Learning representations by back-propagating errors“. In: *Nature* 323.6088. S. 533–536.

Sahoo, Pranab u. a. (2024). *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. arXiv: 2402.07927.

Said, Kabir S. u. a. (2021). „GUI testing for mobile applications: objectives, approaches and challenges“. In: *Proceedings of the 12th Asia-Pacific Symposium on Internetware*. Internetware '20. New York, NY, USA: Association for Computing Machinery, S. 51–60.

Samuel, A. L. (1959). „Some Studies in Machine Learning Using the Game of Checkers“. In: *IBM Journal of Research and Development* 3. S. 210–229.

Sandler, Mark u. a. (2019). *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. arXiv: 1801.04381.

Scao, Teven Le u. a. (2022). *BLOOM: A 176B-Parameter Open-Access Multilingual Language Model*. arXiv: 2211.05100.

Schäfer, Max u. a. (2024). „An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation“. In: *IEEE Transactions on Software Engineering* 50.1. S. 85–105.

Schulman, John u. a. (2017). *Proximal Policy Optimization Algorithms*. arXiv: 1707.06347.

Sennrich, Rico, Barry Haddow und Alexandra Birch (2016). „Neural Machine Translation of Rare Words with Subword Units“. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Hrsg. von Katrin Erk und Noah A. Smith. Berlin, Germany: Association for Computational Linguistics, S. 1715–1725.

Shazeer, Noam u. a. (2017). *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. arXiv: 1701.06538.

Shorten, Connor und Taghi M. Khoshgoftaar (2019). „A survey on Image Data Augmentation for Deep Learning“. In: *Journal of Big Data* 6.1, S. 60.

Sommerville, Ian (2011). *Software engineering*. 9th ed. Boston: Pearson.

Song, Xinying u. a. (2021). „Fast WordPiece Tokenization“. In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. EMNLP 2021. Hrsg. von Marie-Francine Moens u. a. Online und Punta Cana, Dominican Republic: Association for Computational Linguistics, S. 2089–2103.

Sousa, Érica, Carla Bezerra und Ivan Machado (2023). „Flaky Tests in UI: Understanding Causes and Applying Correction Strategies“. In: *Proceedings of the XXXVII Brazilian Symposium on Software Engineering*. SBES 2023: XXXVII Brazilian Symposium on Software Engineering. Campo Grande Brazil: ACM, S. 398–406.

Spasić, Aleksandar J. und Dragan S. Janković (2023). „Using ChatGPT Standard Prompt Engineering Techniques in Lesson Preparation: Role, Instructions and Seed-Word Prompts“. In: *International Scientific Conference on Information, Communication and Energy Systems and Technologies*. ICEST. Bd. 58, S. 47–50.

Srivastava, Nitish u. a. (2014). „Dropout: a simple way to prevent neural networks from overfitting“. In: *J. Mach. Learn. Res.* 15.1, S. 1929–1958.

StatCounter (2024a). *Mobile OS market share worldwide 2009-2024*. StatCounter. URL: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (besucht am 20. 02. 2025).

– (2024b). *Percentage of mobile device website traffic worldwide from 1st quarter 2015 to 4th quarter 2023*. StatCounter. URL: https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/ (besucht am 20. 02. 2025).

Su, Xiaogang, Xin Yan und Chih-Ling Tsai (2012). „Linear regression“. In: *WIREs Computational Statistics* 4.3, S. 275–294.

Sutton, Richard S. und Andrew Barto (2018). *Reinforcement learning: an introduction*. Nachdruck. Adaptive computation and machine learning. Cambridge, Massachusetts: The MIT Press.

Tahchiev, Petar u. a. (2010). *JUnit in Action, Second Edition*. 2nd. USA: Manning Publications Co.

Team, Gemma u. a. (2024). *Gemma: Open Models Based on Gemini Research and Technology*. arXiv: 2403.08295.

Thoppilan, Romal u. a. (2022). *LaMDA: Language Models for Dialog Applications*. arXiv: 2201.08239.

Touvron, Hugo, Thibaut Lavril u. a. (2023). *LLaMA: Open and Efficient Foundation Language Models*. arXiv: 2302.13971.

Touvron, Hugo, Louis Martin u. a. (2023). *Llama 2: Open Foundation and Fine-Tuned Chat Models*.

Toyama, Daniel u. a. (2021). *AndroidEnv: A Reinforcement Learning Platform for Android*.

Tseng, Albert u. a. (2024). *QTIP: Quantization with Trellises and Incoherence Processing*. arXiv: 2406.11235.

Tufano, Michele, Shao Kun Deng u. a. (2022). „Methods2Test: a dataset of focal methods mapped to test cases“. In: *Proceedings of the 19th International Conference on Mining Software Repositories*. MSR '22. New York, NY, USA: Association for Computing Machinery, S. 299–303.

Tufano, Michele, Dawn Drain u. a. (2020). *Unit Test Case Generation with Transformers and Focal Context.*

V. B, Shereena und Julie M. David (2015). „Significance of Dimensionality Reduction in Image Processing“. In: *Signal & Image Processing : An International Journal* 6.3, S. 27–42.

Vaswani, Ashish u. a. (2017). „Attention is all you need“. In: *Proceedings of the International Conference on Neural Information Processing Systems.* NIPS. Bd. 31. Red Hook, NY, USA, S. 6000–6010.

Wang, Hongyu u. a. (2023). *BitNet: Scaling 1-bit Transformers for Large Language Models.* arXiv: 2310.11453.

Wasserman, Anthony I. (2010). „Software engineering issues for mobile application development“. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research.* FoSER '10. New York, NY, USA: Association for Computing Machinery, S. 397–400.

Watkins, Christopher J. C. H. und Peter Dayan (1992). „Q-learning“. In: *Machine Learning* 8.3, S. 279–292.

Wei, Jason u. a. (2023). *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.* arXiv: 2201.11903.

Weiss, Karl, Taghi M. Khoshgoftaar und DingDing Wang (2016). „A survey of transfer learning“. In: *Journal of Big Data* 3.1, S. 9.

Winter, Mario u. a. (2013). *Der Integrationstest: Von Entwurf und Architektur zur Komponenten- und Systemintegration.* München: Hanser.

Wirth, Rüdiger und Jochen Hipp (2000). „CRISP-DM: Towards a standard process model for data mining“. In: *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining.* Bd. 1. Manchester, S. 29–39.

Wu, Chuhan und Ruiming Tang (2024). *Performance Law of Large Language Models.* arXiv: 2408.09895.

Yang, Chengrun u. a. (2024). *Large Language Models as Optimizers.* arXiv: 2309.03409.

Ye, Hui u. a. (2013). „DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag“. In: *Proceedings of International Conference on Advances in*

*Mobile Computing & Multimedia - MoMM '13*. International Conference. Vienna, Austria: ACM Press, S. 68–74.

Ye, Junjie u. a. (2023). *A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models*. arXiv: 2303.10420.

Yosinski, Jason u. a. (2014). „How transferable are features in deep neural networks?“ In: *Proceedings of the International Conference on Neural Information Processing Systems*. NIPS. Bd. 27. Curran Associates, Inc.

Zhang, Aston u. a. (2023). *Dive into Deep Learning*. arXiv: 2106.11342.

Zhang, Chi u. a. (2023). *AppAgent: Multimodal Agents as Smartphone Users*. arXiv: 2312.13771.

Zhang, Dongsong und Boonlit Adipat (2005). „Challenges, Methodologies, and Issues in the Usability Testing of Mobile Applications“. In: *International Journal of Human-Computer Interaction* 18.3. S. 293–308.

Zhang, Zhilu und Mert Sabuncu (2018). „Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels“. In: *Advances in Neural Information Processing Systems*. Bd. 31. Curran Associates, Inc.

Zhao, Heri u. a. (2024). *CodeGemma: Open Code Models Based on Gemma*. arXiv: 2406.11409.

Zheng, Huaixiu Steven u. a. (2024). *Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models*. arXiv: 2310.06117.

Zhu, Hong, Patrick A. V. Hall und John H. R. May (1997). „Software unit test coverage and adequacy“. In: *ACM Computing Surveys* 29.4, S. 366–427.

Zhu, Xiaojin (2008). *Semi-Supervised Learning Literature Survey*. University of Wisconsin-Madison Department of Computer Sciences.

# A Anhang

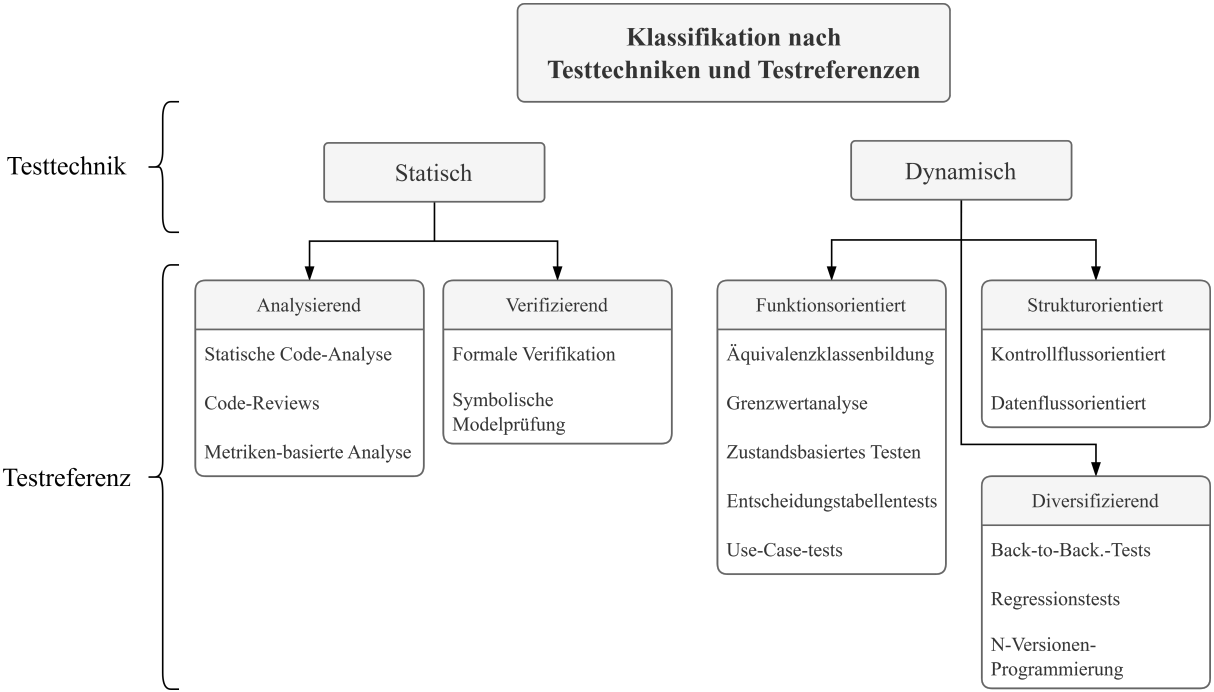



**Abbildung A.1:** Überblick über die Klassifikation der Testtechniken und Testreferenzen

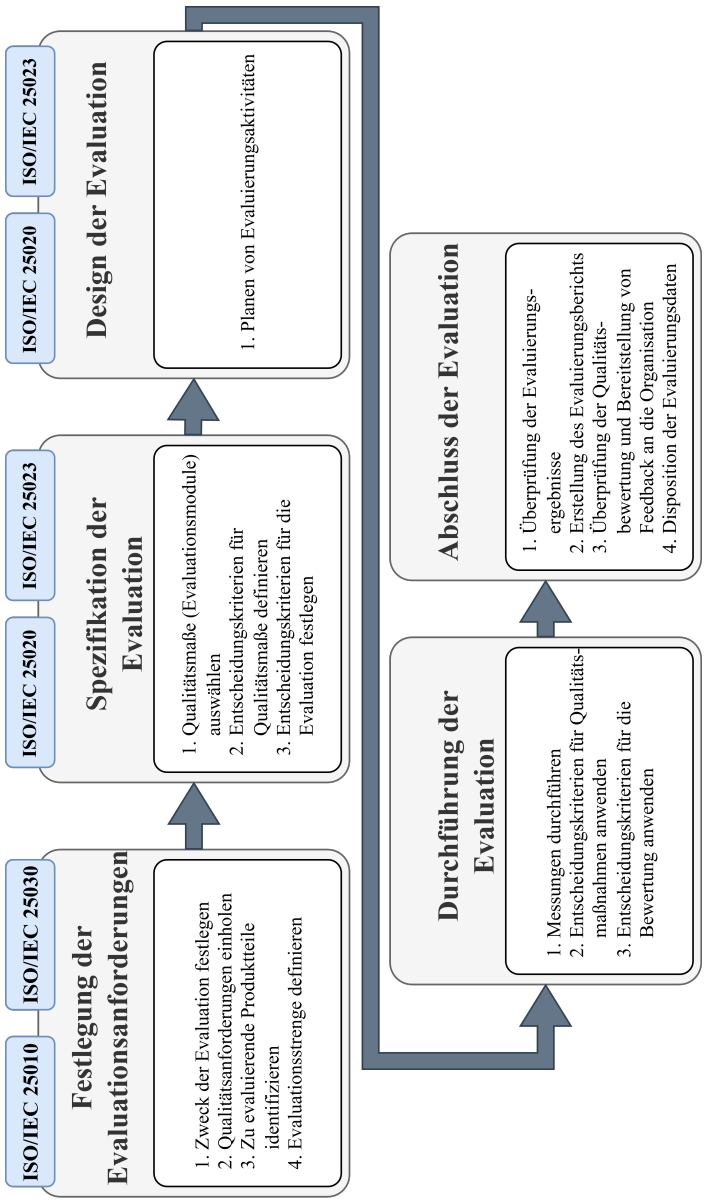



**Abbildung A.2:** Ablauf der ISO/IEC 25040:2011 mit den dazugehörigen Unternormen

| Model | Run Time | Method Config | Repairs | Focal Classes | TC Created |
|---|---|---|---|---|---|
| **codegemma-7b-it** | 00:08:59 | Combined Method[0,No CoT,TaS,No Source] | 3 | 0/8 (0.00%) | 0 |
| **gpt-4o** | 00:42:08 | Combined Method[1,CoT,TaS,Source] | 3 | 4/8 (50.00%) | 150 |
| **gpt-4o** | 00:05:43 | Class | 3 | 7/8 (87.50%) | 48 |
| **gpt-4o-mini** | 00:35:28 | Combined Method[0,CoT,No Tas,Source] | 3 | 7/8 (87.50%) | 108 |
| **gpt-4o-mini** | 00:31:01 | Single Method[0,CoT,TaS,Source] | 0 | 0/8 (0.00%) | 0 |
| **gpt-4o-mini** | 00:28:33 | Combined Method[0,No CoT,TaS,No Source] | 3 | 4/8 (50.00%) | 54 |
| **gpt-4o-mini** | 00:06:29 | Class | 3 | 7/8 (87.50%) | 56 |
| **gpt-4o-mini** | 01:21:16 | Single Method[1,CoT,No Tas,No Source] | 3 | 8/8 (100.00%) | 0 |
| **gpt-4o-mini** | 00:37:17 | Combined Method[1,CoT,TaS,Source] | 3 | 7/8 (87.50%) | 102 |
| **granite-3.1-8b-it** | 00:16:03 | Class | 3 | 0/8 (0.00%) | 0 |
| **llama-3.1-8b-it** | 00:50:11 | Combined Method[0,No CoT,TaS,No Source] | 3 | 3/8 (37.50%) | 4 |
| **llama-3.1-8b-it** | 00:45:45 | Combined Method[1,CoT,TaS,No Source] | 3 | 5/8 (62.50%) | 5 |
| **llama-3.1-8b-it** | 02:26:36 | Single Method[0,CoT,TaS,Source] | 3 | 3/8 (37.50%) | 0 |
| **llama-3.1-8b-it** | 00:18:29 | Class | 3 | 0/8 (0.00%) | 0 |
| **phi-4** | 00:36:53 | Class | 3 | 1/8 (12.50%) | 7 |
| **testgen-llama-3.1-8b** | 00:20:17 | Class | 3 | 0/8 (0.00%) | 0 |
| **testgen-llama-3.1-8b** | 02:26:13 | Combined Method[0,No CoT,TaS,No Source] | 3 | 0/8 (0.00%) | 0 |
| **testgen-llama-3.1-8b** | 03:59:05 | Single Method[0,CoT,TaS,Source] | 3 | 0/8 (0.00%) | 0 |
| **testgen-llama-3.1-8b** | 02:07:34 | Combined Method[1,CoT,TaS,No Source] | 3 | 0/8 (0.00%) | 0 |

**Tabelle A.1:** Testdurchläufe mit unterschiedlichen Modellen und Parametern (Teil 1)

| Model | Classes Build Error | TC Passing | TC Failing | Line Coverage | Condition Coverage |
|---|---|---|---|---|---|
| **codegemma-7b-it** | N/A | N/A | N/A | 0/457 (0.00%) | 0/292 (0.00%) |
| **gpt-4o** | 4/8 (50.00%) | 142/150 (94.67%) | 8/150 (5.33%) | 314/457 (68.71%) | 149/292 (51.03%) |
| **gpt-4o** | 1/8 (12.50%) | 38/48 (79.17%) | 10/48 (20.83%) | 295/457 (64.55%) | 149/292 (51.03%) |
| **gpt-4o-mini** | 1/8 (12.50%) | 71/108 (65.74%) | 37/108 (34.26%) | 325/457 (71.12%) | 161/292 (55.14%) |
| **gpt-4o-mini** | N/A | N/A | N/A | N/A | N/A |
| **gpt-4o-mini** | 4/8 (50.00%) | 31/54 (57.41%) | 23/54 (42.59%) | 211/457 (46.17%) | 92/292 (31.51%) |
| **gpt-4o-mini** | 1/8 (12.50%) | 35/56 (62.50%) | 21/56 (37.50%) | 328/457 (71.77%) | 156/292 (53.42%) |
| **gpt-4o-mini** | N/A | 358/533 (67.17%) | 175/533 (32.83%) | N/A | N/A |
| **gpt-4o-mini** | 1/8 (12.50%) | 61/102 (59.80%) | 41/102 (40.20%) | 332/457 (72.65%) | 158/292 (54.11%) |
| **granite-3.1-8b-it** | 8/8 (100.00%) | N/A | N/A | N/A | N/A |
| **llama-3.1-8b-it** | 5/8 (62.50%) | 3/4 (75.00%) | 1/4 (25.00%) | 0/457 (0.00%) | 0/292 (0.00%) |
| **llama-3.1-8b-it** | 3/8 (37.50%) | 1/5 (20.00%) | 4/5 (80.00%) | 0/457 (0.00%) | 0/292 (0.00%) |
| **llama-3.1-8b-it** | N/A | 0/3 (0.00%) | 3/3 (100.00%) | N/A | N/A |
| **llama-3.1-8b-it** | 8/8 (100.00%) | N/A | N/A | N/A | N/A |
| **phi-4** | 7/8 (87.50%) | 6/7 (85.71%) | 1/7 (14.29%) | 13/457 (2.84%) | 5/292 (1.71%) |
| **testgen-llama-3.1-8b** | N/A | N/A | N/A | N/A | N/A |
| **testgen-llama-3.1-8b** | 8/8 (100.00%) | N/A | N/A | 0/457 (0.00%) | 0/292 (0.00%) |
| **testgen-llama-3.1-8b** | N/A | N/A | N/A | N/A | N/A |
| **testgen-llama-3.1-8b** | 8/8 (100.00%) | N/A | N/A | N/A | N/A |

**Tabelle A.2:** Testdurchläufe mit unterschiedlichen Modellen und Parametern (Teil 2)

| Model | Run Time | Method Config | Repairs | Focal Classes | TC Created |
|---|---|---|---|---|---|
| **codegemma-7b-it** | 00:08:59 | Combined Method[0,No CoT,TaS,No Source] | 3 | 0/8 (0.00%) | 0 |
| **gpt-4o** | 00:42:08 | Combined Method[1,CoT,TaS,Source] | 3 | 4/8 (50.00%) | 150 |
| **gpt-4o** | 00:05:43 | Class | 3 | 7/8 (87.50%) | 48 |
| **gpt-4o-mini** | 00:35:28 | Combined Method[0,CoT,No Tas,Source] | 3 | 7/8 (87.50%) | 108 |
| **gpt-4o-mini** | 00:31:01 | Single Method[0,CoT,TaS,Source] | 0 | 0/8 (0.00%) | 0 |
| **gpt-4o-mini** | 00:28:33 | Combined Method[0,No CoT,TaS,No Source] | 3 | 4/8 (50.00%) | 54 |
| **gpt-4o-mini** | 00:06:29 | Class | 3 | 7/8 (87.50%) | 56 |
| **gpt-4o-mini** | 01:21:16 | Single Method[1,CoT,No Tas,No Source] | 3 | 8/8 (100.00%) | 0 |
| **gpt-4o-mini** | 00:37:17 | Combined Method[1,CoT,TaS,Source] | 3 | 7/8 (87.50%) | 102 |
| **granite-3.1-8b-it** | 00:16:03 | Class | 3 | 0/8 (0.00%) | 0 |
| **llama-3.1-8b-it** | 00:50:11 | Combined Method[0,No CoT,TaS,No Source] | 3 | 3/8 (37.50%) | 4 |
| **llama-3.1-8b-it** | 00:45:45 | Combined Method[1,CoT,TaS,No Source] | 3 | 5/8 (62.50%) | 5 |
| **llama-3.1-8b-it** | 02:26:36 | Single Method[0,CoT,TaS,Source] | 3 | 3/8 (37.50%) | 0 |
| **llama-3.1-8b-it** | 00:18:29 | Class | 3 | 0/8 (0.00%) | 0 |
| **phi-4** | 00:36:53 | Class | 3 | 1/8 (12.50%) | 7 |
| **testgen-llama-3.1-8b** | 00:20:17 | Class | 3 | 0/8 (0.00%) | 0 |
| **testgen-llama-3.1-8b** | 02:26:13 | Combined Method[0,No CoT,TaS,No Source] | 3 | 0/8 (0.00%) | 0 |
| **testgen-llama-3.1-8b** | 03:59:05 | Single Method[0,CoT,TaS,Source] | 3 | 0/8 (0.00%) | 0 |
| **testgen-llama-3.1-8b** | 02:07:34 | Combined Method[1,CoT,TaS,No Source] | 3 | 0/8 (0.00%) | 0 |

**Tabelle A.3:** Testdurchläufe mit unterschiedlichen Modellen und Parametern (Teil 1)

| Model | Classes Build Error | TC Passing | TC Failing | Line Coverage | Condition Coverage |
|---|---|---|---|---|---|
| **codegemma-7b-it** | N/A | N/A | N/A | 0/457 (0.00%) | 0/292 (0.00%) |
| **gpt-4o** | 4/8 (50.00%) | 142/150 (94.67%) | 8/150 (5.33%) | 314/457 (68.71%) | 149/292 (51.03%) |
| **gpt-4o** | 1/8 (12.50%) | 38/48 (79.17%) | 10/48 (20.83%) | 295/457 (64.55%) | 149/292 (51.03%) |
| **gpt-4o-mini** | 1/8 (12.50%) | 71/108 (65.74%) | 37/108 (34.26%) | 325/457 (71.12%) | 161/292 (55.14%) |
| **gpt-4o-mini** | N/A | N/A | N/A | N/A | N/A |
| **gpt-4o-mini** | 4/8 (50.00%) | 31/54 (57.41%) | 23/54 (42.59%) | 211/457 (46.17%) | 92/292 (31.51%) |
| **gpt-4o-mini** | 1/8 (12.50%) | 35/56 (62.50%) | 21/56 (37.50%) | 328/457 (71.77%) | 156/292 (53.42%) |
| **gpt-4o-mini** | N/A | 358/533 (67.17%) | 175/533 (32.83%) | N/A | N/A |
| **gpt-4o-mini** | 1/8 (12.50%) | 61/102 (59.80%) | 41/102 (40.20%) | 332/457 (72.65%) | 158/292 (54.11%) |
| **granite-3.1-8b-it** | 8/8 (100.00%) | N/A | N/A | N/A | N/A |
| **llama-3.1-8b-it** | 5/8 (62.50%) | 3/4 (75.00%) | 1/4 (25.00%) | 0/457 (0.00%) | 0/292 (0.00%) |
| **llama-3.1-8b-it** | 3/8 (37.50%) | 1/5 (20.00%) | 4/5 (80.00%) | 0/457 (0.00%) | 0/292 (0.00%) |
| **llama-3.1-8b-it** | N/A | 0/3 (0.00%) | 3/3 (100.00%) | N/A | N/A |
| **llama-3.1-8b-it** | 8/8 (100.00%) | N/A | N/A | N/A | N/A |
| **phi-4** | 7/8 (87.50%) | 6/7 (85.71%) | 1/7 (14.29%) | 13/457 (2.84%) | 5/292 (1.71%) |
| **testgen-llama-3.1-8b** | N/A | N/A | N/A | N/A | N/A |
| **testgen-llama-3.1-8b** | 8/8 (100.00%) | N/A | N/A | 0/457 (0.00%) | 0/292 (0.00%) |
| **testgen-llama-3.1-8b** | N/A | N/A | N/A | N/A | N/A |
| **testgen-llama-3.1-8b** | 8/8 (100.00%) | N/A | N/A | N/A | N/A |

**Tabelle A.4:** Testdurchläufe mit unterschiedlichen Modellen und Parametern (Teil 2)