GUISpector: An MLLM Agent Framework for Automated Verification of Natural Language Requirements in GUI Prototypes

Kristian Kolthoff*
Institute for Software and Systems Engineering, Clausthal University of Technology
Clausthal-Zellerfeld, Germany
kristian.kolthoff@tu-clausthal.de

Felix Kretzer*
Human-Centered Systems Lab, Karlsruhe Institute of Technology
Karlsruhe, Germany
felix.kretzer@kit.edu

Simone Paolo Ponzetto
Data and Web Science Group, University of Mannheim
Mannheim, Germany
simone@informatik.uni-mannheim.de

Alexander Maedche
Human-Centered Systems Lab, Karlsruhe Institute of Technology
Karlsruhe, Germany
alexander.maedche@kit.edu

Christian Bartelt
Institute for Software and Systems Engineering, Clausthal University of Technology
Clausthal-Zellerfeld, Germany
christian.bartelt@tu-clausthal.de

Abstract

Graphical user interfaces (GUIs) are foundational to interactive systems and play a pivotal role in early requirements elicitation through prototyping. Ensuring that GUI implementations fulfill natural language (NL) requirements is essential for robust software engineering, especially as LLM-driven programming agents become increasingly integrated into development workflows. Existing GUI testing approaches, whether traditional or LLM-driven, often fall short in handling the complexity of modern interfaces, and typically lack actionable feedback and effective integration with automated development agents. In this paper, we introduce *GUISpector*, a novel framework that leverages a multi-modal (M)LLM-based agent for the automated verification of NL requirements in GUI prototypes. First, *GUISpector* adapts a MLLM agent to interpret and operationalize NL requirements, enabling to autonomously plan and execute verification trajectories across GUI applications. Second, *GUISpector* systematically extracts detailed NL feedback from the agent's verification process, providing developers with actionable insights that can be used to iteratively refine the GUI artifact or directly inform LLM-based code generation in a closed feedback loop. Third, we present an integrated tool that unifies these capabilities, offering practitioners an accessible interface for supervising verification runs, inspecting agent rationales and managing the end-to-end requirements verification process. We evaluated *GUISpector* on a comprehensive set of 150 requirements based on 900 acceptance criteria annotations across diverse GUI applications, demonstrating effective detection of requirement satisfaction and violations and highlighting its potential for seamless integration of actionable feedback into automated LLM-driven development workflows. The video presentation of *GUISpector* is available at: https://youtu.be/JByYF6BNQeE, showcasing its main capabilities.

CCS Concepts

• **Computing methodologies** → **Intelligent agents**.

Keywords

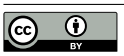
Automated GUI Verification, Natural Language Requirements, Multi-modal LLM GUI Agent, Automated GUI Feedback Generation

ACM Reference Format:
Kristian Kolthoff, Felix Kretzer, Simone Paolo Ponzetto, Alexander Maedche, and Christian Bartelt. 2026. GUISpector: An MLLM Agent Framework for Automated Verification of Natural Language Requirements in GUI Prototypes. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3774748.3787611

1 Introduction

Graphical user interfaces (GUIs) are central to effective interaction with modern software systems and have become ubiquitous across domains. Beyond their role in end-user interaction, GUIs are frequently employed as prototypes during requirements elicitation [24], providing stakeholders with concrete, interactive artifacts that facilitate clearer communication and more precise specification of requirements [4, 25]. As with other critical software artifacts, rigorous GUI testing is essential to ensure both functional correctness and stakeholder alignment. However, GUIs often exhibit significant complexity and dynamic behavior, making comprehensive testing a challenging and resource-intensive task. Moreover, human test data is often ambiguous and existing approaches rely on often scattered tools [17]. Ensuring that GUIs correctly implement elicited requirements is particularly important in the prototyping phase, as discrepancies can lead to costly misunderstandings and rework [4]. The recent surge in large language model (LLM)-driven GUI generation [8, 14] has accelerated automated GUI development, making equally robust automated testing essential.

*Both authors contributed equally to the paper.



ICSE-Companion '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2296-7/26/04
https://doi.org/10.1145/3774748.3787611

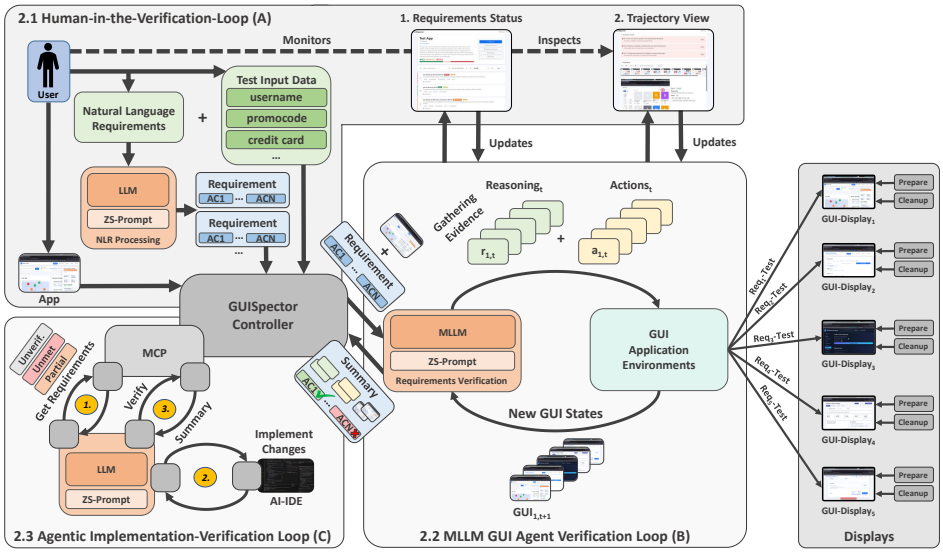


Figure 1: Overview of the *GUISpector* architecture with Human-in-the-Verification-Loop, MLLM GUI agent verification loop with reasoning, actions and GUI state trajectories and agentic implementation-verification loop for fully autonomous mode

To alleviate the burden of manual GUI testing, numerous automated approaches have been proposed, including automated test scripts [27], random [19] and model-based exploration for improved coverage [11, 28], and scripted bug replay techniques [7, 10]. However, these methods often struggle with low coverage due to the complexity and dynamic nature of GUIs. Recent advances leverage LLMs to enable more sophisticated and semantically meaningful GUI testing. For example, LLMs have been used to predict user actions for state exploration [18] and to generate tests from natural language descriptions [5, 6]. Yet, these approaches typically rely on textual abstractions of GUI hierarchies, missing critical dynamic and visual aspects and are unable to address non-functional and fine-grained requirements.

In this paper, we address this gap by introducing *GUISpector*, a multi-modal (M)LLM-based agent adapted for the automated verification of natural language requirements in GUI prototypes. *GUISpector* leverages vision-language reasoning to interpret and operationalize NL requirements, autonomously explore GUI applications and extract actionable summaries from test trajectories. Crucially, our approach enables a closed feedback loop, where insights from GUI verification can be directly fed back into LLM-driven GUI generation, a capability that remains largely unexplored despite its demonstrated benefits in code generation workflows [23]. As LLM-driven GUI development becomes widely adopted [8, 14], *GUISpector* provides an accessible, integrated tool that empowers practitioners to seamlessly incorporate requirements-driven verification and iterative improvement into automated GUI development. Code, datasets and prototype are available at our repository [1].

2 Approach: GUISpector

GUISpector comprises three main components: *(A)* a human-in-the-verification-loop, enabling users to specify the target application, natural language requirements, input data, and monitoring the verification process, *(B)* the core MLLM GUI agent verification loop, iteratively generating reasoning for planning and evaluation, selecting actions, and recording GUI screenshot trajectories, and *(C)* an agentic implementation-verification loop that enables LLM-based programming agents to seamlessly interact with *GUISpector*, for fully automated workflows. The architecture is shown in Fig. 1.

2.1 Human-in-the-Verification-Loop

In *GUISpector*, users begin by specifying the application for verification, providing unstructured NL requirements and supplying relevant test input data. Requirements are automatically processed using a zero-shot prompted LLM, which extracts a structured representation of each requirement. This acts as a generic NL interface, capturing key elements such as the description and particularly individual acceptance criteria, with the option to infer missing details from context. *GUISpector* offers an intuitive user interface that presents an overview of all requirements along with their current fulfillment status and allows users to inspect the verification trajectories executed by the MLLM GUI agent. Each agent interaction can be examined in detail, including the MLLM reasoning and a breakdown of acceptance criteria fulfillment or violation, supported by evidence collected during verification runs. Overall, *GUISpector* supports efficient, parallelized NL requirements-driven GUI testing.

2.2 MLLM GUI Agent Verification Loop

The core component of *GUISpector* is the MLLM agent verification loop, which adapts a pretrained multi-modal LLM optimized for computer use by employing a zero-shot prompt that instructs the agent to autonomously verify whether each requirement is implemented in the application, starting from a given URL and interacting with the GUI to gather evidence. The prompt enforces fully autonomous operation, minimal actions, use of user-provided input test data and requires the agent to evaluate each acceptance criterion individually, justifying its decision with concrete evidence and outputting a structured JSON summary. For each verification run, the agent receives the requirement, acceptance criteria and test data along with a GUI screenshot representing the current

Table 1: Evaluation results for requirements (Req) and acceptance criteria (AC) across five applications, with averages of steps, time and token consumption. Token counts are reported in thousands (k). Costs using *OpenAI CUA*: $3/M input, $12/M output. Number of steps, time, input and output-token consumption, and costs are reported per requirement, all other metrics per app.

	Met (Req)			Unmet (Req)			Partial (Req)			Met (AC)			Unmet (AC)			#Steps		Time (s)		#In-Tok. (k)		#Out-Tok. (k)		Cost ($)	
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	Av.	SD	Av.	SD	Av.	SD	Av.	SD	Av.	SD
App-1	.789	.938	.857	1.00	.714	.833	.500	.429	.462	.908	.967	.937	.920	.793	.852	25.1	17.4	336.0	291.2	229.8	165.5	2.394	1.323	.689	.029
App-2	1.00	.944	.971	1.00	1.00	1.00	.833	1.00	.909	.968	.984	.976	.958	.920	.939	21.1	14.9	322.3	316.0	189.7	127.8	1.984	0.817	.569	.024
App-3	.933	.824	.875	.833	.833	.833	.625	.833	.714	.931	.900	.915	.793	.852	.821	19.5	16.6	223.1	240.5	174.6	149.2	2.002	1.020	.524	.024
App-4	1.00	.824	.903	.750	.857	.800	.500	.667	.571	.946	.898	.922	.824	.903	.862	22.3	12.4	293.4	182.3	202.7	107.7	2.115	0.822	.608	.025
App-5	.850	1.00	.919	1.00	.857	.923	.667	.400	.500	.909	1.00	.952	1.00	.778	.875	34.2	19.8	412.9	266.7	311.9	188.0	2.850	1.399	.936	.034
Avg.	.915	.906	.905	.917	.852	.878	.625	.666	.631	.933	.950	.940	.899	.849	.870	24.4	16.2	317.6	259.3	221.7	147.6	2.269	1.076	.665	.027
SD	.093	.079	.044	.118	.102	.082	.138	.258	.183	.026	.048	.025	.088	.064	.043	5.8	2.8	68.8	51.4	54.3	31.4	0.364	0.274	.163	.004

application state. The agent then reasons about the most plausible next actions to verify acceptance criteria, predicts the next action (e.g., *click*, *double click*, *scroll*, *type*, with parameters such as *x,y* coordinates), and executes it in the GUI environment, which updates the GUI state. Subsequently, a new screenshot is captured and returned to the model, closing the loop. This iterative process produces a trajectory of GUI states, reasoning steps, and actions (GUI_1, r_1, a_1, …, GUI_n, r_n, a_n), culminating in a final output JSON containing a detailed summary, explanations and evidence for each acceptance criterion. These summaries (particularly important for violated requirements) can be accessed by users and fed back into LLM-driven programming agents to support iterative GUI development and improvement. Before and after each verification run, the GUI environments are cleaned up to avoid contaminating states.

2.3 Agentic Implementation-Verification Loop

Beyond manual user-driven requirements verification, *GUISpector* also supports a fully automatic mode that integrates LLM-based programming agents with the proposed GUI verification approach closing the development loop. With the emergence and recent popularity of LLM-powered IDEs such as *Cursor* [3] and *Copilot* [9], developers can easily integrate their large-scale code bases into LLMs, enabling to rapidly implement changes with LLMs and potentially increase productivity. To facilitate seamless integration, we implemented a *Model Context Protocol (MCP)* [12] server, allowing external agents to directly interact with *GUISpector*. LLM-driven programming agents can retrieve requirements, including acceptance criteria and their current verification states for different verification setups previously created by the user and selectively process unverified or violated requirements. Agents can then invoke verification runs through *GUISpector*, receiving actionable explanations for any *unmet* or *partially met* requirements. This enables iterative improvement: after addressing identified issues, the agent re-runs verification until the requirement is *met*, then proceeds to the next one. We employ a zero-shot prompt to instruct programming agents to follow this fully automated implementation-verification loop, representing a novel approach that leverages actionable feedback from automated MLLM GUI agents to drive iterative, requirements-driven development and enhance the capabilities of LLM agents.

2.4 Implementation

GUISpector is built as a *Django* web application with an *HTML/CSS* and *JavaScript* frontend and uses *MySQL* for data storage. Background tasks, such as executing parallel MLLM agent verifications, are managed with *Celery* across multiple worker nodes, coordinated by *Redis* for efficient and scalable processing. To support simultaneous verification runs, we implemented a display resource manager in *Redis* that dynamically allocates and tracks *Xfce* desktop environments running on *Xvfb* virtual displays. *xdotool* is used to simulate user interactions within GUI applications. For MLLM-based verification, we currently employ the *OpenAI CUA* [22] model, while for other LLM tasks, the framework offers support for *OpenAI GPT* [20], *Google Gemini* [26], and *Anthropic Claude Sonnet* [2], allowing flexible model selection. Other MLLM computer-use agents can seamlessly be integrated into the proposed *GUISpector* framework.

3 Experimental Evaluation

GUI-Requirements Evaluation Dataset. To evaluate the ability of the proposed approach to recognize requirements fulfillment, we first created a new dataset, due to the lack of available datasets of GUIs paired with requirements. Using *Codex* [21], we generated 30 functional requirements with three acceptance criteria (ACs) each for five domain-diverse applications (*Park-and-Pay, Budget Tracker, Recipe Generator, Fitness Quests, Cleaning Booking*). In a second step, *Codex* was instructed to create the corresponding apps (single HTML files with local state, no external CDNs/APIs). All prompts, requirements, ACs and apps are available in our repository [1].

We then evaluated whether the generated apps conformed to the specifications. For each app, the target distribution was 18 *met*, six *unmet* and six *partially met* requirements (30 per app). Two authors independently labeled 450 acceptance criteria (ACs) as *met* or *unmet*. Annotations were independent but not equally blinded: one annotator had not seen the detailed specification; the other had prior exposure but did not consult it during annotation. Requirement states were derived from AC labels (all ACs *met* = Req. *met*; all ACs *unmet* = Req. *unmet*; otherwise = Req. *partially met*).

Before resolving disagreements, we measured inter-coder reliability across 150 requirements (5 apps × 30) for three raters (*LLM-Specification*, *Annotator 1*, *Annotator 2*), indicating very high agreement (*Krippendorff's* α : 0.929, ordinal, 95% CI [0.880, 0.966]). We adopt the agreed human evaluations as the gold standard; eight disagreements (out of 150) occurred between the LLM and humans. At the acceptance-criteria level (5 apps x 30 requirements x 3 ACs), agreement between the two human annotators was similarly high (*Cohen's* κ : 0.921, pooled, nominal, with accuracy: 96.7%, 435 out of 450 ACs). Stratified by app, κ ranged from 0.841 (*App 1*) to 0.972 (*App 2*), indicating high overall agreement with some heterogeneity.

Evaluation Setup and Results. Based on our ground truth for each requirement and its acceptance criteria, we evaluated how well *GUISpector* recognizes fulfillment and violations. For this, each app was analyzed in single evaluation runs. Requirements were classified into three categories (*met*, *partially met*, *unmet*), while acceptance criteria were assigned two categories (*met*, *unmet*). We then calculated *precision*, *recall*, and *F1* scores for the three-class requirements task and the binary acceptance-criteria task.

Table 1 summarizes per-app and average results. For the acceptance criteria task, both labels met (*F1* avg. = 0.94) and unmet (*F1* avg. = 0.87) achieve high scores, indicating that the framework reliably identifies fulfillment and violations at the criterion level. For the three-class requirements task, *F1* remains high for *met* (avg. = 0.905) and *unmet* (avg. = 0.878), whereas *partially met* is slightly lower (avg. = 0.631), which may result from boundary confusions due to greater semantic ambiguity. Execution effort varies by app (averg. *steps* 19.5 for *App 3* to 34.2 for *App 5*). The observed spread is expected because each run starts from a clean state and certain apps require longer prerequisite flows before a requirement can be verified The elapsed *time* per requirement likewise differs across apps for the same reasons. Although per-requirement runtime may, in some instances, exceeds that of human testers, the framework can make up for it by supporting straightforward parallelization. Costs are dominated by MLLM input tokens and average $0.67 per requirement, we expect reductions by merging verification runs.

4 Related Work

Automated GUI testing has evolved from early approaches such as automated test scripts [27], random exploration [19], and model-based techniques [11, 28], to more recent scripted bug replay methods [7, 10]. While these methods have improved testing efficiency, they often suffer from limited coverage due to the inherent complexity and dynamic behavior of GUIs. More recently, large language models (LLMs) have been leveraged to generate more human-like and semantically meaningful GUI interactions. For example, *GPTDroid* [18] frames GUI testing as a Q&A task, using LLMs to iteratively explore app states, while *CAT* [5] combines retrieval-augmented generation and machine learning to automate UI test generation, and multi-agent approaches have been proposed for testing multi-user features [6]. However, these LLM-based methods typically rely on textual representations of GUI hierarchies, which neglect the visual and dynamic aspects of GUIs, limiting their ability to verify non-functional requirements and fine-grained acceptance criteria. In parallel, LLM-assisted prototyping tools [13, 15, 16] have introduced requirements matching and traceability features, but focus primarily on static, mid-fidelity GUI prototypes rather than fully interactive applications. Unlike these prior works, *GUISpector* enables multi-modal, requirements-driven verification directly on interactive GUIs and uniquely introduces a closed feedback loop by providing actionable summaries from agent-based verification to inform and iteratively improve LLM-driven GUI implementation.

5 Conclusion & Future Work

GUISpector introduces a multi-modal LLM agent framework for automated verification of NL requirements in GUIs, enabling both interactive and fully automated workflows. Our approach shows effective detection of requirement satisfaction and violations, and provides actionable feedback for iterative development. As future work, we aim to improve efficiency by merging related verification runs to avoid redundant exploration of shared subpaths.

Acknowledgments

This work is supported by the *Federal Ministry of Research, Technology and Space (BMFTR)* of Germany and by the *Federal Ministry for Economic Affairs and Energy (BMWE).*

References

[1] 2025. GUISpector. https://github.com/kristiankolthoff/GUISpector.
[2] Anthropic. 2025. Claude 4. (2025). https://www.anthropic.com/news/claude-4
[3] Inc. Anysphere. 2025. Cursor. https://cursor.com
[4] Michel Beaudouin-Lafon and WE Mackay. 2002. Prototyping development and tools. *Handbook of Human-Computer Interaction* (2002), 1006–1031.
[5] Sidong Feng et al. 2024. Enabling Cost-Effective UI Automation Testing with Retrieval-Based LLMs: A Case Study in WeChat. arXiv:2409.07829 [cs.SE]
[6] Sidong Feng et al. 2025. Agent for User: Testing Multi-User Interactive Features in TikTok. *arXiv preprint arXiv:2504.15474* (2025).
[7] Sidong Feng and Chunyang Chen. 2022. Gifdroid: Automated replay of visual bug reports for android apps. In *Proceedings of the 44th International Conference on Software Engineering*. 1045–1057.
[8] Lennart Fiebig, Kristian Kolthoff, Christian Bartelt, and Simone Paolo Ponzetto. 2025. Effective GUI Generation: Leveraging Large Language Models for Automated GUI Prototyping. (2025).
[9] Inc. GitHub. 2025. GitHub Copilot. https://github.com/features/copilot
[10] Lorenzo Gomez et al. 2013. Reran: Timing-and touch-sensitive record and replay for android. In *35th International Conference on Software Engineering*. IEEE, 72–81.
[11] Tianxiao Gu et al. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.
[12] Xinyi Hou et al. 2025. Model context protocol (mcp): Landscape, security threats, and future research directions. *arXiv preprint arXiv:2503.23278* (2025).
[13] Kristian Kolthoff et al. 2024. Interlinking user stories and GUI prototyping: A semi-automatic LLM-based approach. In *2024 IEEE 32nd International Requirements Engineering Conference (RE)*. IEEE, 380–388.
[14] Kristian Kolthoff et al. 2024. Zero-Shot Prompting Approaches for LLM-based Graphical User Interface Generation. *arXiv preprint arXiv:2412.11328* (2024).
[15] Kristian Kolthoff et al. 2025. GUIDE: LLM-Driven GUI Generation Decomposition for Automated Prototyping. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 1–4.
[16] Felix Kretzer, Kristian Kolthoff, et al. 2025. Closing the loop between user stories and GUI prototypes: an LLM-based assistant for cross-functional integration in software development. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–19.
[17] Felix Kretzer and Alexander Maedche. 2025. TESY: A Usability Test-Driven Prototyping Assistant Connecting Designers with Crowd-Testers. *Proc. ACM Hum.-Comput. Interact.* 9, 2, Article CSCW184 (May 2025), 24 pages.
[18] Zhe Liu et al. 2023. Chatting with gpt-3 for zero-shot human-like mobile automated gui testing. *arXiv preprint arXiv:2305.09434* (2023).
[19] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th international symposium on software testing and analysis*. 94–105.
[20] OpenAI. 2023. GPT-4 Technical Report. *arXiv preprint arXiv:2303.08774* (2023).
[21] OpenAI. 2025. Codex. (2025). https://openai.com/codex/
[22] OpenAI. 2025. CUA: Introducing a universal interface for AI to interact with the digital world. (2025). https://openai.com/index/computer-using-agent
[23] Yun Peng et al. 2025. Perfcodegen: Improving performance of llm generated code with execution feedback. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 1–13.
[24] Klaus Pohl. 2010. *Requirements engineering: fundamentals, principles, and techniques*. Springer Publishing Company, Incorporated.
[25] Alon Ravid and Daniel M Berry. 2000. A method for extracting and stating software requirements that a user interface prototype contains. *Requirements Engineering* 5, 4 (2000), 225–241.
[26] Gemini Team, Rohan Anil, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).
[27] Qing Xie and Atif M Memon. 2007. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16, 1 (2007), 4–es.
[28] Xia Zeng et al. 2016. Automated test input generation for android: Are we really there yet in an industrial case?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 987–992.