A Road Map for Tamed Interactions with Large Language Models

VINCENZO SCOTTI, JAN KEIM, and TOBIAS HEY, KASTEL, Karlsruhe Institute of Technology, Karlsruhe, Germany
ANDREAS METZGER, Ruhr Institute for Software Technology (paluno), University of Duisburg-Essen, Essen, Germany
ANNE KOZIOLEK and RAFFAELA MIRANDOLA, KASTEL, Karlsruhe Institute of Technology, Karlsruhe, Germany

Large Language Models are increasingly embedded in software systems (GenAIware), enabling new forms of automation and interaction. However, their probabilistic nature and reliance on prompt programming challenge reliability, robustness, and maintainability. In current practice, prompt-related concerns (e.g., context management, interaction logic, output validation) are embedded in general-purpose code, leading to implicit, hard-to-analyze systems. We argue that prompt programming should be treated as a first-class Software Engineering (SE) concern and propose LLM Scripting Language (LSL), a Domain-Specific Language (DSL) for structuring LLM interactions as analyzable programs. LSL introduces abstractions for interaction blocks, context scopes, output constraints, and control flow, separating deterministic logic from probabilistic model behavior while ensuring syntactic compliance. From an SE perspective, LSL supports disciplined development by making interaction logic explicit, analyzable, and amenable to verification and validation. It also acts as cognitive scaffolding, externalizing prompt design into programmable artifacts that reduce implicit reasoning and support systematic debugging, evolution, and reuse. We illustrate these properties in a structured generation scenario, showing improved failure localization and interaction transparency. While LSL does not guarantee semantic correctness or factual accuracy, it provides a principled foundation for more analyzable and maintainable prompt-based systems.

CCS Concepts: • **Computing methodologies** → **Natural language processing**; • **Software and its engineering** → **Domain specific languages**;

Additional Key Words and Phrases: SE4AI, LLM, LSL, DSL, NLP

This work was supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF) and the Engineering Secure Systems (KASTEL) (Grant No. 46.23).
Authors' Contact Information: Vincenzo Scotti, KASTEL, Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: vincenzo.scotti@kit.edu; Jan Keim, KASTEL, Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: jan.keim@kit.edu; Tobias Hey, KASTEL, Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: hey@kit.edu; Andreas Metzger, Ruhr Institute for Software Technology (paluno), University of Duisburg-Essen, Essen, Germany; e-mail: andreas.metzger@paluno.uni-due.de; Anne Koziolek, KASTEL, Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: koziolek@kit.edu; Raffaela Mirandola (corresponding author), KASTEL, Karlsruhe Institute of Technology, Karlsruhe, Germany; e-mail: raffaela.mirandola@kit.edu.
*For correspondence, please contact Vincenzo Scotti.



This work is licensed under Creative Commons Attribution International 4.0.

© 2026 Copyright held by the owner/author(s).
ACM 1557-7392/2026/9-ART279
https://doi.org/10.1145/3832776

ACM Reference format:
Vincenzo Scotti, Jan Keim, Tobias Hey, Andreas Metzger, Anne Koziolek, and Raffaela Mirandola. 2026. A Road Map for Tamed Interactions with Large Language Models. *ACM Trans. Softw. Eng. Methodol.* 35, 9, Article 279 (September 2026), 40 pages.
https://doi.org/10.1145/3832776

1 Introduction

The advancements in **Generative AI (GenAI)** [17], and in particular LLMs [66], are rapidly reshaping how **software engineering (SE)** addresses software systems design and development. Beyond their role as development-time assistants or copilots [47], LLMs are increasingly embedded directly into software products to realize functionality that would otherwise be infeasible or too expensive to implement [17]. In such *GenAIware* systems [17], foundation models are invoked at runtime as integral software components, powering capabilities such as conversational interfaces, information retrieval, content generation, and adaptive decision support [93]. As a result, software systems are no longer defined solely by code and data but also by *prompts* (i.e., natural language instructions) that govern the behavior of these models. This shift introduces new SE challenges, as developers must now reason about artifacts that are inherently non-deterministic, difficult to validate, and hard to maintain. In practice, this leads to issues such as inconsistencies in generated outputs, propagation of invalid results, and increased difficulty in debugging and evolving LLM-based systems [14, 26, 37].

Recent research has observed that prompts in GenAIware are no longer simple textual inputs but are becoming meticulously engineered parts of the codebase, effectively functioning as programs themselves [17, 61]. These prompts encode *task definitions*, *interaction protocols*, *constraints*, and *expectations about model outputs*, and they critically shape the behavior of the overall system. More broadly, GenAIware development increasingly involves context engineering [63], which consists of the construction, scoping, and evolution of the information provided to an LLM, and output constraining [46, 74], which consists of specifying structural and contractual expectations on generated results (either to validate the output or to enforce specific behavior). For simplicity, we refer to this combination of prompt design, context management, and output constraints as *prompt programming*.

Together, the prompt programming components determine how foundation models behave within software workflows. Recent work in **human–computer interaction (HCI)** further reinforces this view by showing that prompt programming exhibits properties analogous to software development, despite lacking a formal grammar or programming language structure [41]. However, despite their importance, these elements are still largely treated as *ad hoc* strings or implicit conventions, embedded in general-purpose code and refined through trial and error. This practice makes these prompt-based systems fragile, hard to debug, difficult to evolve, and challenging to reason about from a SE perspective [17].

This situation exposes a fundamental SE gap, and the root of the problem is not expressiveness but *analyzability*. Existing languages and frameworks are expressive enough: prompt templates can be written in Python, output validators can be defined with libraries such as Pydantic, and orchestration pipelines can be assembled with frameworks such as LangChain. What these approaches cannot provide is a clear separation between deterministic software logic and probabilistic model behavior. When prompt programming is embedded in general-purpose code, interaction logic becomes implicit, verification targets become unclear, and static analysis is practically infeasible, because conventional control flow is interleaved with probabilistic LLM invocations in ways the host language cannot reason about.

In GenAIware systems, developers therefore face increased difficulty in inspecting the structure of interactions with LLMs, enforcing invariants on generated outputs, and reasoning about the behavior of prompt-driven workflows over time. As systems grow in complexity, this lack of analyzable structure undermines reliability, maintainability, and trustworthiness and prevents the systematic application of good SE practices such as verification, validation, and lifecycle management.

In this article, we argue that addressing these challenges requires treating prompt programming as a *first-class* SE concern and that doing so demands dedicated programming abstractions, not merely libraries, that make prompt behavior (i) *separable*: the LLM interactions (prompts, scaffolding code, etc.) should be clearly separated from code that is not related to prompt programming; (ii) *inspectable*: the structure and intent of each interaction is explicit and human-readable; (iii) *analyzable*: static and runtime analysis can reason about control flow, reachability, and contracts; and (iv) *governable*: output constraints and execution invariants are part of the language semantics, not bolted on as external validators. As such, we propose the concept of **LLM Scripting Language (LSL)**: a **Domain-Specific Language (DSL)** designed to express prompt-based interactions as structured programs. Rather than replacing general-purpose programming languages or existing GenAI frameworks, LSL defines a *restricted interaction layer* that sits between deterministic software and probabilistic foundation models. In this sense, LSL is intended to play a role analogous to SQL for databases or regular expressions for pattern matching: it does not supersede host languages but provides a specialized abstraction that makes reasoning, optimization, and verification amenable in ways that would be impractical in unrestricted general-purpose code.

LSL is intended to define prompts as executable specifications. It provides constructs to structure interactions with LLMs, manage and scope context, constrain outputs, and orchestrate multi-step workflows involving both model invocations and conventional software components. By making prompt logic explicit and declarative, LSL enables reasoning about the structure and control flow of GenAIware interactions and opens the door to applying established SE techniques (such as static analysis, runtime validation, testing, and monitoring) at the level of prompt programs. While LSL does not aim to guarantee semantic correctness or factual truthfulness of model outputs, it offers structural guarantees about interactions, contracts, and execution paths, thereby improving engineering control over prompt-driven systems.

The novelty of this work lies exactly in reframing prompt programming into a first-class programming model. While existing frameworks provide mechanisms for prompt templating, structured outputs, and workflow orchestration, they treat these concerns as library-level constructs embedded in general-purpose code. In contrast, LSL elevates interactions with the LLM to explicit, analyzable program artifacts, where context management, output constraints, and control flow are integrated into the language semantics. This enables forms of reasoning—such as static analysis of interaction structure, verification of workflow properties, and systematic enforcement of interaction contracts—that are not achievable when prompt logic remains implicit within host languages. In this sense, LSL is not a convenience layer over existing tooling but a shift in abstraction boundary from "using LLMs inside programs" to "programming interactions with LLMs as programs."

From a human–**AI** interaction perspective, LSL follows a mixed-initiative approach. Pure natural language is flexible but semantically unstable across executions; pure code is rigorous but too rigid to concisely express domain intent directed at a probabilistic model. LSL occupies the productive middle ground: natural language remains central for expressing task intent and domain knowledge, while lightweight formal structure stabilizes and constrains that intent, making it amenable to analysis and enforcement.

This article presents a vision and research road map for LSL as a prompt-programming language for GenAIware. We position LSL within the broader landscape of GenAIware development, clarify

its scope and intended guarantees, and outline how it can support reliability, robustness, and trustworthiness in prompt-based systems. By reframing prompts as programs and treating prompt programming as a core SE concern, we aim to contribute to the foundations needed to engineer GenAIware systems with the same discipline expected of traditional software.

We divide the rest of this vision article into the following sections. In Section 2, we recap the essentials about LLMs. In Section 3, we summarize the main work related to ours and position our contribution. In Section 4, we describe LSL in detail. In Section 5, we present an application example to illustrate the use of LSL. In Sections 6 and 7, we comment, respectively, on the future opportunities and challenges we foresee. In Section 8, we report the design and results of our evaluation investigating the need for abstractions like LSL. In Section 9, we present a research road map towards realizing the vision of LSL. Finally, in Section 10, we summarize our considerations.

2 Background

In this section, we provide an overview of LLMs (Section 2.1), highlighting capabilities (Section 2.2) and limitations (Section 2.3).

2.1 Overview

LLMs are **Deep Neural Networks (DNNs)** trained on massive amounts of data to learn a probabilistic generative model of text. The underlying DNN of an LLM is a Transformer [92], a DNN designed to process sequential data, like sequences of text tokens,[1] through the *self-attention mechanisms* [92]. This Transformer architecture enables LLMs to capture long and complex dependencies throughout the input text as well as memorize and recall a large amount of information, allowing the generation of highly coherent and contextually relevant text.

We mainly distinguish these LLMs based on their size, computed in the number of parameters of the underlying DNN, since, usually, their capabilities depend on the size of the model and the amount of training data. Bigger and more capable models, including closed-access models such as *GPT* or *ChatGPT* [52, 72] and *Gemini* [18, 79], are hundreds or thousands of billions of parameters large. Smaller models, on the other hand, have only up to tens of billions of parameters. These include open-access models such as *DeepSeek* [21, 33, 34], *Llama* [38, 90], *Vicuna* [62], *Mistral* [55, 56], and *Gemma* [65, 80]. The latter group represents a set of highly valuable alternatives for research and application development that, under proper tweaking, can rival the bigger ones. Moreover, these open models are often also released in larger versions [34, 38, 56].

Open-access LLMs are mainly distributed, sharing the neural network weights learned during the pre-training. Often, to make these pre-trained models usable as assistants (like ChatGPT), they are *fine-tuned* (i.e., refined via further training) to behave as *instruction-following* agents or *chatbot-assistants* [28, 82, 84] by training on conversations (or other interactions) where a user and an agent communicate via natural language to solve some tasks. Typically, the fine-tuning data is formatted using a specific template, where the input is a sequence composed of *system message* (main task instructions, which can also be chatbot directives); *user message* (user's input to be processed, question, or request); *agent response* (agent's output or response to the user's question or request). Each sample in the dataset contains one or more message-response exchanges between the user and the agent (LLM) covering multiple tasks, multiple steps within a task, or to handle user corrections. Alternatively, other models are distributed as coding assistants, like in the case of *Copilot* [47] or *Code Llama* [81], where fine-tuning is used to refine the model on source code generation, which is, however, a capability most pre-trained or assistant LLMs already have. Differently from the past years, where fine-tuning was used to adapt a pre-trained model to a very

[1] Tokens are the basic units processed by a (e.g., words, sub-words, or characters).

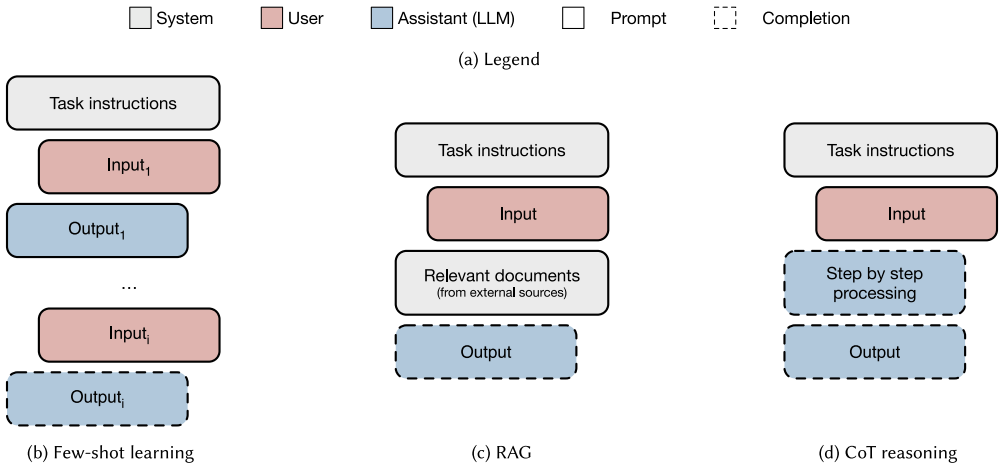


Fig. 1. LLM prompting approaches.

specific task like classification (often modifying the architecture) [35, 77], nowadays prompting (i.e., instructing the model via natural language) is preferred to solve most problems with LLMs, and fine-tuning is applied very rarely.

2.2 Capabilities

Auto-regressive (or causal) LLMs, which represent the vast majority of current models, are designed to generate text autoregressively. They yield the probability distribution from which to sample the next token one at a time, conditioning the distribution of this next token on all the preceding ones (the context). From a practical perspective, LLMs are used by prompting with some prefix text, which may include some instructions for a task or some user input, and generating an output completion by sampling iteratively from the predicted probability distribution.

An LLM can be prompted to tackle problems in different ways. The main prompting approaches, which we report in Figure 1, are *few-shot learning* (sometimes called in-context learning) [23], **Retrieval Augmented Generation (RAG)** [45], and **Chain of Thought (CoT)** *reasoning* [59]. Few-shot learning, depicted in Figure 1(b), consists of providing additional examples of input–output pairs (the shots) together with the task instructions as part of the prompt, helping the LLM better "understand" the task and the expected output format. This is an example of the ability of LLMs to generalize from limited data. RAG, depicted in Figure 1(c), consists of automatically extending the input context by integrating information coming from external knowledge sources and, thus, allowing the model to ground its output responses in this relevant information. CoT reasoning, depicted in Figure 1(d), consists of forcing the model to generate a step-by-step reasoning process before the actual output, giving the model more space for its internal reasoning processes and, thus, improving the performance on complex tasks.

All these typical approaches are orthogonal to each other, meaning that they can be combined for better performance. Moreover, none of them are specific to instruction-following or chatbot assistant fine-tuning and can be used with pre-trained models [23].

Beyond their role as passive text generators, LLMs increasingly support deployment in *agentic* configurations, where they operate as (semi-)autonomous decision-makers [15]. In such settings, an LLM is prompted with explicit goals, contextual system information, and access to external tools such as search, simulation, or code execution.

2.3 Limitations

LLMs come with very general capabilities and high flexibility, but they are prone to many known issues. These issues range from architectural limitations to problems with the generated content.

From the DNN architecture side, we identify two main issues: context size (i.e., the maximum number of consecutive tokens a model can manage) and computational complexity. Even though in the last few years we have seen an increase in techniques designed to extend physically or virtually the maximum context, the problem of having a finite context still remains, and, especially in smaller models, the attention doesn't keep up well when the context grows excessively. Moreover, the computation of the *self-attention mechanism* (core of the Transformer) is quadratic in time to the length of the input. There are caching mechanisms and alternative implementations [31, 32, 76] to cope with these issues, but they aren't always usable. Besides their architectural limitations, LLMs stay probabilistic models. This nature makes them robust to noise and uncertainty but also prone to errors, as they rely on probability rather than sound values. These errors can be caused by different factors, including sensitivity to input variations or ambiguities and overfitting to training knowledge and patterns.

As a result of the technical issues, the models end up relying on more recent context and may end up generating contradictions, struggling with multi-step reasoning, or simply deviating from the original context and instructions. Moreover, due to the probabilistic nature, it may end up generating biased content based on unreliable, outdated, or even unethical information. At the same time, this probabilistic nature may lead to similar phenomena called hallucinations. An LLM creates hallucinations to simply generate a completion that is more statistically likely, but the output consists of made-up facts and details that are neither grounded in world knowledge, causing problems with factuality [54], nor in the current context, causing problems with faithfulness [54].

The combination of these issues makes these models unsuitable for tasks that require high reliability, robustness, and trustworthiness, like automating a pipeline using the LLM. Consider the case where we are interested in interaction with a user and generate a call to some external API, and the model fails to close a parenthesis. The use of techniques like few-shot learning may help the model to stick to a given pattern (e.g., generate well-formed JSON). Similarly, RAG helps to integrate more updated and reliable knowledge (e.g., refer to the latest news), and CoT reasoning may help improve accuracy in multi-step reasoning tasks (e.g., solving an equation). Yet, all these techniques are nothing but palliatives, as they only help to reduce the probability that an error occurs but without any guarantee that the problem won't show up.

3 Related Work

In this section, we present the state of the art of SE support for GenAIware (Section 3.1); then, we focus on the existing frameworks designed to manage the interaction with LLMs (Section 3.2) and on the known challenges arising from GenAIware development; and, finally, we position the proposed LSL framework with respect to the state-of-the-art (Section 3.4).

3.1 SE for AI

This article elaborates on how the introduction of AI-powered components within software systems (GenAIware), particularly those relying on LLMs, should be paired with appropriate tools for the development of such software systems that allow for the correct use of such AI components. As mentioned above, realizing this potential at scale requires a paradigm shift in how these applications are developed, deployed, and maintained. Yet, current practices, often characterized by *ad hoc* prompt engineering and fragmented toolchains, hinder productivity, reproducibility, and the development of robust, enterprise-grade GenAIware [19, 67, 85]. Empirical studies show that

developers struggle to engineer GenAIware in practice, with low resolution rates for prompt-related issues and long debugging cycles, highlighting the lack of appropriate abstractions for structuring interactions [26, 27, 61].

A range of tools and approaches has been proposed to facilitate the development of AI-based software systems, including libraries for prompt templating, retrieval tools that integrate semantic search, and plug-ins for AI-assisted coding. These tools address specific aspects of AI integration, such as handling prompts [27, 61], leveraging curated data through techniques like RAG [19, 85], and embedding AI capabilities within traditional software workflows. However, existing solutions often treat these components in isolation. In contrast, our approach seeks to provide a more unified framework that considers *prompts* and *data*, alongside *code*, as first-class citizens in the software development process.

In the 1980s, the concept of *literate programming*, where computers could be instructed purely through natural language, seemed science fiction [58]. Yet, we are now moving in that direction: tools like **Natural Language Outlines (NLO)** offer a novel modality and interaction surface for providing AI assistance to developers throughout the software development process [86]. Such NLOs enable a bidirectional sync between code and natural language, where a developer can change either code or NL and have the LLM automatically update the other. In contrast, LSL tries to go beyond this iterative approach managed by an assistant and tries to mix together structured code with natural language so that, where necessary, the AI component can be easily "programmed" using a prompt. Moreover, the suite coming with LSL is thought to also integrate tools for verification and validation, rather than the "bare" programming components.

While LLMs are increasingly used in SE, there is still no fully integrated approach that embeds them across the DevOps lifecycle. Software engineers are dealing with the following problems: (1) *Fragmented tooling*: Current tools are often disparate, requiring developers to stitch together multiple components, leading to inefficiencies and compatibility issues [67]. (2) *Lack of standardization*: There's a dearth of industry-wide standards for prompt engineering, promptware architecture, interfaces, and lifecycle management, leading to inconsistent development practices [61, 67, 89]. (3) *Limited verification and debugging capabilities*: Debugging LLM-based applications, especially hallucination-related issues, remains challenging due to their black-box nature and probabilistic outputs [36, 67]. (4) *Immature lifecycle management*: While *MLOps* addresses model lifecycle [39], the specific nuances of prompt and data versioning, testing, and deployment within a combined software-prompt-data paradigm are not fully mature [67].

Problems like the fragmented tooling (1) and the lack of standardization (2) are not new for SE research. In fact, there were similar issues when several cloud providers started appearing, which led to the development of multi-cloud research to help deal with the different interfaces exposed by the vendors [22].

Nevertheless, the SE community is moving to improve several aspects related to the development of GenAIware. Practice to take AI components into account now starts from the early requirement elicitation stage [40] and spans throughout the entire lifecycle process [42]. Prompt engineering and prompt programming are currently evolving to produce robust solutions that can keep up with the ever-changing nature of LLMs, also becoming integral components of software applications [27, 61], also known as *promptware*. Data management is strongly influencing the development as well, leading to architectures specifically designed for properly integrating LLMs and data within the software system [19, 85]. Recent work has explored IDE-inspired tooling for prompt programming, including semantic highlighting, structured editing, and automated detection of prompt structure [41]. In contrast to such tool-centric approaches, our work focuses on providing language-level abstractions that make interaction structure explicit and analyzable.

Beyond engineering practices, there are also emerging efforts to rethink programming models for GenAIware at a more fundamental level. One example is Oracular Programming [60], which proposes a paradigm where LLMs act as runtime oracles that resolve high-level non-deterministic strategies. This approach emphasizes modularity, evolvability, and extensibility, treating calls to LLMs as structured programming constructs rather than *ad hoc* prompts. While our proposal of LSL focuses on scripting, constraining, and verifying interactions to provide stronger guarantees, Oracular Programming highlights a complementary research direction in which language abstractions are designed to embrace non-determinism while still supporting systematic reasoning about GenAIware systems.

3.2 Existing Frameworks

With our work, we are interested in covering and unifying several aspects related to the adoption of LLMs and LLM-powered components within software systems. There are several key aspects we are interested in. First, we want to look into application development, looking into prompt management, function calling, agentic AI, and data management. Second, we want to improve reliability and robustness through standard interfaces, automatic input templating and formatting, and output constraints specification. Third, we want to create trustworthiness through verification and validation and through explainability.

Looking at the state of the art in terms of tools and frameworks addressing these issues, we could not help but notice a strong fragmentation and a lack of standardization that are preventing the automation of many aspects of the AI-powered software lifecycle.

About application development frameworks and tools to offer the different functionalities, *LangChain* and *LangGraph* [2] are the earliest and most famous examples. Yet, through the last years, alternatives like *Semantic Kernel* [13], *AutoGen* [96], *Flowise* [4]. Some of these frameworks, like *CrewAI* [3], are specifically designed for agentic or multi-agent application development. The most recent contribution in this sense is the NLO tool [86], which brings the development closer to literate programming. Unlike inference engines, many solutions for context management attempt to provide a unified interface to all models; however, they often lack the ability to constrain the generated output.

On the data side of context management, in recent years, frameworks like *Llama Index* [12] have helped automate most of the work involved in building a vector document store. These frameworks allow to enhance data-intensive applications requiring RAG to offer their functionalities. In this sense, *Haystack* [8] represents a valuable tool that provides resources for both data management and model orchestration.

With a specific focus on reliability and robustness, we mostly depend on inference engines or frameworks. Solutions like *Llama CPP* and *Llama CPP Python* [10, 11] offer many low-level functionalities for templating the input and capturing elements of the output. Unlike other existing inference engines and tools, Llama CPP allows specifying a *generative grammar* to constrain the generated completions. We talk in these cases of *grammar-constrained decoding* [46]. The closest tool in this sense is *Guidance* [7] that offers a templating system well blended into Python syntax and the possibility to specify capture groups to be automatically applied to the output and to specify a finite list of alternative completions to select from. However, most of these tools rely only on HuggingFace's Transformers API or use their own custom interfaces. Prior work shows that LLM-generated artifacts frequently contain structural defects, including malformed code, incomplete outputs, and inconsistencies across steps, even when they are functionally plausible [14].

There are some tools specialized in output constraints and validation. *NeMo Guardrails* [78] was designed to impose programmable guardrails to make sure a dialogue with an assistant follows a

specific template or that certain harmful topics are avoided. Many of these tools are thought to impose safety guardrails [53], and in fact, there are specialized solutions to capture or prevent toxic and harmful content generation, like *Guardrails AI* [6].

Concerning trustworthiness, we have two main aspects being touched: verification and validation on one side and explainability on the other. About verification and validation, we have that now testing tools and frameworks like *LangTest* [68] or *EvoTox* [30], which are designed to take into account the black-box and probabilistic nature of LLMs, offering a robust evaluation of both model performance or model compliance to certain behaviors. However, to the best of our knowledge, no testing suite exploits the information coming out of the tests to quantify and help improve the trustworthiness of the overall software system. Adopting AI-based components and their integration in the development environment is still at an early stage when available.

Similarly, while we have access to tools and techniques like *InTraVisTo* [24], *Tuned Lens* [20], or *LM Transparency Tool* [91], which are designed specifically for the explainability of LLMs, we hardly see these tools integrated in any development platform. However, these tools can turn out to be very helpful for "debugging a prompt," a prompt for example, showing how the instructions given to the model affect the predicted output.

3.3 Known Challenges

Recent empirical and conceptual studies highlight fundamental challenges in engineering GenAIware and, more broadly, AI-based software [83]. Although these works address different aspects, they collectively reveal consistent patterns that point to deeper structural issues in how LLM interactions are currently designed and implemented. A key difficulty is that prompts lack a predefined grammar, making it hard to reason about their structure and to support standard programming operations such as refactoring or reuse [41]. Recent studies highlight that testing and debugging LLM-based systems are fundamentally challenging due to non-determinism and implicit interaction logic, requiring new abstractions beyond traditional SE approaches [37, 67].

Unlike traditional software components, LLM-based components exhibit inherent non-determinism, producing different outputs even under identical inputs and configurations. As a result, correctness can no longer be assessed as a binary property, but rather as a distribution of outcomes across repeated executions, challenging established assumptions in SE, particularly in testing, verification, and reproducibility [37]. Furthermore, behavior in GenAIware is not solely defined by code but emerges from the interaction between prompts, model inference, and surrounding control logic. Prompt construction acts as both input specification and behavioral modifier, effectively encoding parts of the system logic in natural language, yet this interaction logic is typically embedded in general-purpose code, where it remains implicit and difficult to analyze or reason about [37].

Empirical studies show that developers face persistent and diverse challenges when developing GenAIware, including prompt design, integration, reproducibility, and output interpretation. Many of these issues are difficult to resolve in practice, with a low proportion of questions receiving accepted solutions and long response times in developer communities [26]. These findings indicate that current development practices lack adequate abstractions and methodologies to support reliable engineering of LLM-based systems.

Taken together, these findings point to a fundamental gap: the primary source of complexity in GenAIware lies not only in the models themselves but also in the interaction logic that governs their use. This logic, expressed through prompt design, context management, and multi-step interactions, is typically implicit, scattered across code, and difficult to analyze, test, and maintain. From these observations, we derive the following requirements for engineering GenAIware:

R1 *Explicit interaction structure*: Interaction logic should be explicitly represented rather than embedded implicitly in host language code.
R2 *Separation of concerns*: Deterministic control flow and probabilistic model behavior should be clearly separated.
R3 *Analyzability and verifiability*: Interaction structures should be amenable to validation, verification, and systematic debugging.
R4 *Support for variability*: The programming model should explicitly account for non-deterministic outputs and variability across executions.

3.4 Positioning of This Work

A natural question is why a new DSL is needed at all. Powerful general-purpose languages such as Python already exist, alongside a rich ecosystem of libraries for LLM orchestration, validation, and guardrails. Indeed, many of the individual techniques discussed in this article, like prompt templating, structured outputs, or output validation, can be implemented already using existing frameworks. Across multiple empirical studies, developers consistently report difficulties in prompt design, integration, and failure diagnosis, indicating that current approaches do not scale to complex workflows [41].

Our position is that the core limitation of current approaches is not expressiveness, but analyzability through an appropriate abstraction level. Existing approaches to building GenAIware largely rely on general-purpose programming languages and framework-based abstractions to orchestrate model interactions. While these approaches provide flexibility and prototyping capabilities, they typically treat prompt construction and interaction logic as *ad hoc* elements embedded in application code. As a result, they do not make interaction structure explicit, nor do they provide mechanisms for analyzing or verifying the behavior arising from such interactions.

Recent research has highlighted the need for new SE approaches that account for the non-deterministic and interaction-driven nature of LLM-based systems. In particular, challenges related to variability, testing, and developer experience suggest that prompt programming cannot be viewed merely as string manipulation but requires principled abstractions to manage complexity.

In this context, we position LSL as a DSL designed to address the structural limitations of current practices by introducing first-class abstractions for modeling LLM interactions. Rather than embedding prompts and interaction logic implicitly within general-purpose code, LSL represents interactions as explicit, structured programs.

Specifically, LSL addresses the requirements identified in Section 3.3 with the following features:

F1 LSL provides explicit constructs for defining interaction blocks and workflows, making prompt-based logic visible and structured.
F2 LSL separates deterministic control logic from probabilistic model execution, enabling clearer reasoning about system behavior.
F3 By enforcing structural constraints and well-defined interaction boundaries, LSL enables static analysis and systematic validation of interaction logic.
F4 LSL exposes interaction boundaries and constraints that make variability explicit and manageable within the programming model.

LSL plays a role analogous to SQL for databases or regular expressions for pattern matching: it does not supersede host languages but provides a specialized abstraction that enables reasoning, optimization, and verification that would be impractical in unrestricted code. Overall, LSL shifts prompt programming from an implicit and ad hoc practice to an explicit, structured, and analyzable SE activity, providing a principled foundation for developing and maintaining GenAIware.

4 LSL

In this section, we detail our proposed DSL, LSL, as a way to control LLMs within pipelines and workflows (Section 4.1), we explain how we envision the approach to constrain and exploit LLMs generative capabilities (Section 4.2) and the interaction structure (Section 4.3) in this framework. Moreover, we provide overview of the LSL core language model (Section 4.4). Finally, we comment on the expected user base of our DSL (Section 4.5).

4.1 Proposed Framework

We propose LSL: a DSL to format and constrain the input and output of an LLM. The idea is to develop a scripting language, its interpreter, and the surrounding development tools to improve the guarantees on many aspects concerning the use of LLMs and make them more autonomous. The interpreter running the script should interact with the LLM, either communicating with it locally or remotely in a seamless way to the user.

Drawing from HCI theory, LSL can be understood as a *cognitive scaffold* that restructures how users (SE practitioners) formulate, execute, and reason about interactions with language models. Rather than relying on *ad hoc* prompt engineering, the LSL provides explicit abstractions for control flow, data dependencies, and intermediate reasoning steps, thereby externalizing mental processes that would otherwise remain implicit. In line with theories of *external cognition* (i.e., cognitive offloading) [69] and distributed cognition [49, 71], the language acts as a representational medium that reduces cognitive load, improves transparency, and supports more systematic debugging and reuse.

Prior work shows that existing prompt engineering tools provide little support for forming accurate mental models of LLM behavior, forcing users to rely on trial-and-error and intuition, further motivating structured abstractions such as LSL as cognitive scaffolding [41]. By constraining interaction patterns and making reasoning structure visible, LSL helps bridge the semantic gap between user intent and LLM behavior, enabling more reliable and cognitively tractable human–AI collaboration.

The intended uses of the scripting language are (i) defining interaction blocks, (ii) managing the context processed by the LLM, (iii) formatting the input prompt to the LLM, and (iv) constraining the output completion of the LLM.

We are designing this DSL to be used by experts who need to integrate AI in their applications or services. The experts use LSL to write scripts to automate some pipeline, part of an application, using LLMs. The end user, or other services, can benefit from these AI-based applications using these pipelines. In Figure 2, we provide a high-level view of the development environment. An LSL script already contains all the essential information to generate some form of executable (which basically is the GenAIware) together with the information to constrain the input and output and to validate the interactions automatically. All this can be generated and packed together by the language interpreter into an LSL application. The runtime environment and the verification and validation environment work together during execution to run the application while enforcing and monitoring the runtime behavior during the interactions. Interfaces are provided both on the LLM side, to provide inference, and on the data side, to allow the use of both datasets (for testing, calibration, few-shot learning, etc.) and external resources (like document bases, knowledge graphs, etc.).

LSL is thought to be used in any kind of interaction, from unstructured chit-chats to more complex structured or semi-structured use cases, like knowledge-grounded conversations and automating the creation of databases, as we show in the examples of Figures 3 and 4. Considering the semi-structured use case of the knowledge-grounded chat, as represented in Figure 3, we can

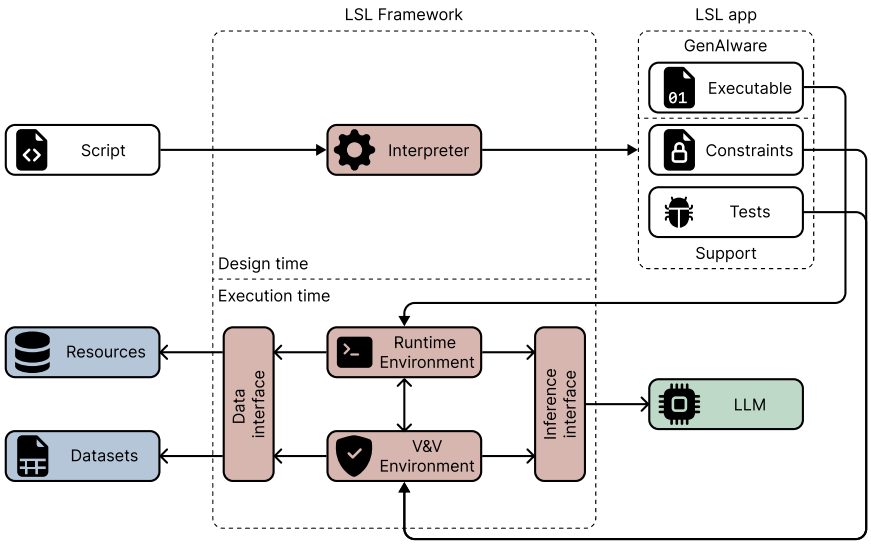


Fig. 2. Idea behind LSL: the script is converted into an application bundling an executable (which is ingested by the runtime environment) and the support material composed of the constraints to satisfy and the tests to validate the program behavior (which are ingested by the verification and validation environment). The data interface provides access to datasets (for benchmarking, performance estimate, etc.) and to resources (document bases, knowledge bases, etc.), the inference interface warps the access to the LLM.

see how LSL can be used to provide a global structure for the dialogue, alternating user messages and model responses. Before triggering the LLM completion to get the response, we can issue a call to a sub-routine that takes care of gathering relevant documents. This sub-routine, in turn, issues two consecutive calls to as many sub-routines for query identification and extraction. Lastly, the approach uses the LLM to further filter retrieved documents, all in a separate context. Conversely, structured use cases find a lot of compatible applications in process automation, like in the example of database creation from informal specifications of Figure 4. In this second use case, we can see how LSL can be used to issue three consecutive calls to as many sub-routines to identify entities, attributes, and relationships, all in separate contexts. From the extracted information, LSL can inject another piece of the prompt similar to CoT to ease the generation of the SQL code with the definitions of the tables; moreover, with LSL, we can constrain the last completion to stick to the SQL syntax to enforce well-formed code.

When we mention "constraining the output of the LLM," we mean limiting the output to a subset of possible strings, whether they are enumerated or specified with some formalism, as we detail in Section 4.2. Instead, when we mention "controlling the interaction with the LLM" we mean properly adding or removing pieces of the context whenever prompting the LLM, as we further outline in Section 4.3. In Figures 3 and 4, we show how we plan to constrain the output: We enforce the generated answer to some questions to be "yes" or "no," as shown in Figures 3 and 4, or to stick to some specific syntax, as displayed in Figure 4. Similarly, we show how we plan to control the scope of the interaction to a single question and its answer rather than retaining all user input when selecting documents, like in Figure 3, or when extracting entities and their relationships, like in Figure 4.

LSL is therefore not proposed as a competitor to existing orchestration frameworks but as a complementary abstraction that enables principled engineering analysis of LLM interactions. We summarize the core distinctions of LSL from existing Python-based frameworks in Table 1.

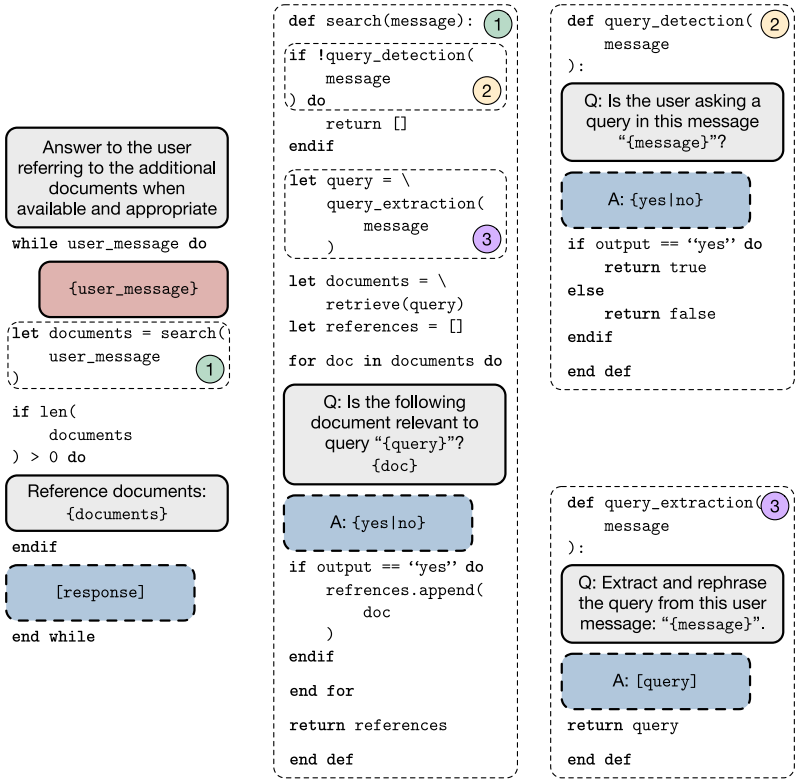


Fig. 3. Example semi-structured use case (system messages, instructions, data, and subroutine outputs): knowledge-grounded chat.

4.2 Beyond Prompting

Engineering the prompt to trigger the correct model completion is a non-trivial task [29, 82]. Besides phrasing the request to the LLM correctly in natural language, there are other difficulties. One difficulty is making sure to provide all necessary information (e.g., task description, target labels, output format). Another crucial difficulty concerns using correct techniques like few-shot learning to increase the probability of triggering the desired model's behavior, whether it is solving the correct task or sticking to the correct output format. Nevertheless, all these techniques cannot give any formal guarantee on the structure of the generated output.

Generative grammars can be the solution to tackle this issue. In fact, they offer a formalism that can be exploited to constrain the decoding of the LLM output probabilities to well-formed strings according to a given specification. These grammars can be applied freely to the generation process of an LLM. There are simple use cases, like generating only digit tokens if computing the value of a mathematical expression. Other use cases are more complex, like the example of generating syntactically correct SQL code for a query to be run in Figure 4 or generating JSON strings that respond to a given JSON schema. As we discuss later in Sections 6 and 7, this constrained decoding still does not ensure the soundness of the output but at least gives guarantees on the fidelity to the desired structures.

With these grammars, we can specify the set of all acceptable and desirable completions for the current prompt. The proper way to select the completion would be to pick the one that maximizes

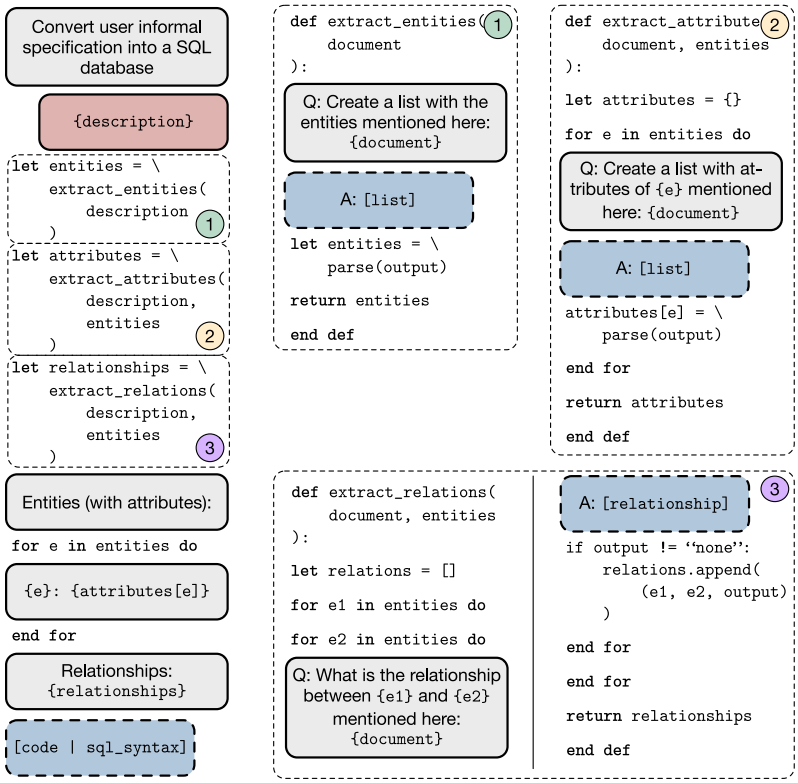


Fig. 4. Example structured use case (system messages, instructions, data, and subroutine outputs): database creation.

Table 1. Key Differences between Existing Python-Based Frameworks and the Proposed LSL

Aspect	**Python-based framework**	**LSL**
Interaction logic	Embedded in host language	First-class scripted artifact
Output constraints	Runtime validation	Part of execution semantics
Static analysis	Limited	Possible (termination, reachability)
Verification	External, *ad hoc*	Integrated into the language model
Intended user	Application developer	SE practitioner defining contracts
Primary goal	Productivity	Reliability and analyzability

the joint probability of its tokens according to the LLM. However, this may not be feasible either because the set of valid strings for the grammar is infinite or because it is still too large to be processed in a reasonable time. In these cases, resorting to search strategies like *greedy search* or *beam search* may help reduce the computational burden at the cost of introducing sub-optimal, but still syntactically valid, solutions.

The use of these grammars does not exclude the combined use with typical prompt engineering techniques that are still useful to make sure the probabilities in the output correlate with the task to solve. Additionally, LSL and its interpreter can be designed to help in prompt programming, offering tools for templating the context in a way that is more suitable to the LLM or by automating

some passages like the retrieval of relevant information of examples for the few-shot learning from known and reliable sources.

4.3 Beyond Linear Interactions

Most applications that integrate chatbots based on LLMs see the interaction with the user as a linear process where the user and chatbot alternate sending messages to each other. However, the stateless nature of LLMs makes them compatible with non-linear interaction, whether they are chats or other more structured processes [43, 75]. This means that we can integrate an LLM in a workflow characterized by (i) multiple, possibly independent, steps to solve the main task or its sub-tasks, (ii) isolated steps that do not require processing the entire context, (iii) states that needs to be passed from one step to another, and (iv) forward and backward evolutions of the interaction.

Figure 4 shows how both a semi-structured task, such as the knowledge-grounded chat of the example in Figure 3, and a structured task like the generation of SQL code to build a database of the example in Figure 4 require multiple steps. Moreover, some passages are iterative, some steps are self-contained, and many of them require defining and moving around variables: these are all premises for the development of a DSL. Nowadays there are tools and frameworks like *Langchain* [2] to help define pipelines and automate processes or like *NeMo Guardrails* to impose some constraints on the dialogue flow and content [78], but (i) they often lack expressivity to deal with complex problems, (ii) they do not take into account possible optimizations to enhance performances, (iii) they are bound to one or a small set of domains (e.g., programming languages), (iv) they do not necessarily allow for rapid prototyping and deployment, and (v) they do not exploit formal tools to provide some form of verification or validation. With LSL, we aim at easing the implementation of all these steps through a unified interface.

Moreover, some passages, like the relevant document selection in the search function of the example in Figure 3 or the attributes and relations extractions of the example in Figure 4, are independent and can be parallelized. A proper integration of this feature in LSL can allow automatic exploitation of the high degree of parallelism that DNNs naturally enable. In fact, they can process samples in batches independently of each other, allowing, where applicable, a better utilization of the computational resources.

Going further with non-linear interactions, we also consider error management. In fact, despite the use of LSL, there still may be faults caused either by the underlying LLM or by the environment. For these cases, we foresee the integration of tools to trace the source of the errors and apply correction actions, either by asking for user feedback, when applicable, or by prompting the LLM to provide a revised completion [87], given the previous faulty completion and the error.

4.4 Core Language Model

LSL is designed for *prompt programming in GenAIware systems*, where the primary abstraction is not computation over data but structured interaction with a probabilistic model. In such systems, prompts, context, and output constraints jointly determine the behavior of the underlying foundation model and must therefore be treated as first-class software artifacts [17].

The design of LSL is guided by three core principles: *interaction-centricity*, *explicit context and constraints*, and *analyzability over expressiveness*. LSL programs describe *interaction workflows* rather than purely algorithmic computations. Each program consists of a sequence of interactions between deterministic software components and an LLM, where each interaction is explicitly defined and scoped. Moreover, LSL treats context construction, scoping, and evolution as explicit

language constructs and similarly elevates output constraints to first-class elements. This reflects the central role of context engineering and output constraining in GenAIware development [17]. Finally, consistent with DSL design principles, LSL prioritizes analyzability over unrestricted expressiveness, enabling inspection and reasoning about prompt-driven behavior as software artifacts [64].

The fundamental unit of computation is the *interaction block*, representing a single prompt–response exchange:

```
def classify_sentiment(text: str)
    write Classify the sentiment of the given text  # Preamble
    write Q: '''\n{text}\n'''                       # Input
    read A: {positive|negative|neutral}             # Output
end def
```

This interaction encapsulates the context, instruction, and output constraint as a single analyzable unit.

Context scopes define which information is visible to the model:

```
@context(sys_message, history: list[mapping[str, str]])
def chat()
    while true
        {sys_message}                                          # Preamble
        {history[-2:]}                                         # Keep only last two messages
        let message: str = read User: {user}
        append(history, {"role": "user", "content": message})
        let response: str = read Assistant: {}
        append(history, {"role": "assistant", "content": response})
        clear                                                  # Clear context
    end while
```

This prevents unintended accumulation of context and enables reasoning about information dependencies.

Outputs are constrained using formal specifications, such as enumerations, schemas, or grammars.

```
import json

...

def extract_entities(document: str): json[list[str]]
    Extract entities from the given document                          # Preamble (write implicit)
    '''\n{text}\n'''                                                  # Input (write implicit)
    let entities: json[list[str]] = read Entities:\n{json[list[str]]}  # Output "```json\n...\n```"
    return entities
end def
```

More expressive constraints can be defined using grammars:

```
import sql

...

def generate_sql(
    entities: list(str), attributes: mapping(str, list[str], relationships: list[tuple[str, str, str]
)
    Create a SQL database given the following entities and relationships.
    Entities
    for let e: str in entities
        {e}:
        for let a: str in attributes[e]
        \t- {a}
         end for
     end for
     for let e1: str, e2: str, r: str in relationships
         {r}({e1}, {e2})
     end for
     let code: sql[str] = read {sql[str]}
end def
```

LSL provides constructs for sequencing, branching, and iteration.

```
def search(query: str): list[str]
    ...
end def

def answer_with_docs(message: str, docs: list[str]): str
    ...
end def

def answer(message: str): str
    ...
end def

...

let response: str
if detect_query(message) do
    let docs: list[str] = search(message)
    response = answer_with_docs(message, docs)
else
    response = answer(message)
endif

...
```

Execution can be monitored and adapted dynamically.

```
import python

def correct(spec: str, code: python[str], error str): python[str]
    ...
end def

...

def generate_code(spec: str, tests: list[callable[python[str]]): str
    Write a Python function with the following specifications.  # Preamble (write implicit)
    {spec}                                                     # Input (write implicit)
    let code: python[str] = read {python[str]}                 # Output
    let counter: int = 0
    try
        label retry
        for let t: callable in test:
            t(code)
        on error:
            code = correct(spec, code, error)
            if counter <= 2 do
                counter += 1
                goto retry
            end if
        end on
    end try
end def
```

In our intended execution model, an LSL program is interpreted as a sequence of steps alternating between deterministic computation and probabilistic model invocation. Each interaction block is evaluated through the following stages:

(1) *Context construction*, assembling relevant inputs and state;
(2) *Prompt instantiation*, materializing the instruction template;
(3) *Constrained generation*, invoking the LLM under output constraints;
(4) *Validation and binding*, checking conformance and storing results.

This execution model introduces a clear separation between conventional control flow and probabilistic behavior, which is essential for enabling analysis and reasoning [17].

Consider a two-step workflow:

```
def classify_sentiment(text: str): [positive|negative|neutral]
    ...
end def

def summarize_issue(text: str): str
    ...
end def

...

let sentiment: [positive|negative|neutral] = classify_sentiment(review)
if sentiment == "negative" do
    let summary: str = summarize_issue(review)
    ...
end if

...
```

This corresponds to two interaction blocks evaluated sequentially, with explicit dataflow between them.

LSL semantics are defined in terms of structured interactions with explicit control over both context and output space.

Context is explicitly constructed and scoped for each interaction.

```
@context(history: list[mapping[str, str])
def answer_with_docs(documents: list[str]): str
    let message str: history[-1]["content"]
    new                                                                          # Start new context
    Answer the following message referring to the given documents when appropriate.  # Preamble
    Reference documents:
    for let doc: str in documents:
        '''\n{doc}\n'''
    Q: {message}                                                                 # Input
    let response: str = read {response}                                          # Output
    return response
end def
```

This avoids interference between unrelated steps and ensures predictable behavior.

Output validity is defined by conformance to constraints. LSL enforces *structural correctness* by restricting the set of admissible outputs.

```
def detect_query(message): bool
    Q: Is the user asking a query in this message: "{message}"?
    let response: [yes|no] = A: {yes|no}
    if response == "yes" do
        return true
    else
        return false
    endif
end def
```

The model is constrained to produce one of the allowed values, ensuring compatibility with downstream control flow.

Our objective with LSL is to enable reasoning about: (i) control-flow properties (e.g., reachability, termination), (ii) structural validity of outputs, (iii) consistency of interaction contracts, and (iv) presence of recovery strategies.

For example, given a grammar-constrained output:

```
import json
import service_xyz

...

def call_api(intent: str, params: json[mapping[str, str]])
    Generate the API call for the given intent and parameters.  # Preamble (write implicit)
    Intent: {intent}                                           # Input (write implicit)
    Parameters: {params}
    let call: service_xyz[json] = read {service_xyz[json]}     # Output
    call()
end def
```

One can ensure that all generated API calls conform to the expected syntax.

However, LSL does not guarantee: (i) factual correctness, (ii) absence of hallucinations, and (iii) semantic validity of outputs.

These limitations arise from the probabilistic nature of LLMs, whose outputs are generated based on learned distributions rather than deterministic computation.

4.5 Stakeholders

LSL is designed primarily for software engineers who need to integrate LLM-powered functionality into applications and services but who may lack deep expertise in machine learning or natural language processing. These practitioners understand their application domains and requirements well but face barriers when attempting to prompt LLMs effectively: they must navigate complex machine learning or deep learning frameworks, manage probabilistic outputs, and ensure reliability in production systems. By providing a scripting interface that abstracts away low-level LLM implementation details while still offering fine-grained control over interactions, LSL gives these professionals the opportunity to focus on solving domain-specific problems rather than (i) dealing

with the machine learning infrastructure and (ii) managing the several tools necessary to build the intended component. Secondary stakeholders include DevOps engineers responsible for deploying and maintaining AI-powered services, as well as quality assurance professionals who need to verify and validate LLM-based components within larger software systems.

We assume that LSL users possess core SE competencies rather than specialized AI expertise. Specifically, users should be proficient in programming under imperative or procedural paradigms (e.g., using languages like C, Bash, or Python) and comfortable with standard programming constructs such as control flow, functions, and error handling. They should have practical knowledge of SE practices, including testing, debugging, and verification techniques, as well as basic familiarity with DevOps workflows like *continuous integration* or *continuous delivery* pipelines. While users need not be machine learning researchers, they should understand LLMs at a conceptual level; this includes LLMs capabilities and limitations (e.g., hallucinations and probabilistic outputs), as well as basic prompt engineering techniques such as few-shot learning. Experience with DSLs (e.g., regular expressions and build scripts) and APIs is beneficial, as is the ability to design multi-step workflows and data pipelines. Finally, users should be willing to engage with lightweight formalism, such as defining constraints through JSON schemas or grammar-like specifications, without requiring expertise in formal verification methods.

5 Illustrative Example

To illustrate the implications of the proposed LSL programming model, we analyze a representative application example: the generation of SQL code from informal user specifications, as introduced in Figure 4. This scenario is representative of GenAIware systems where LLMs are used to produce structured artifacts that are subsequently consumed by deterministic software components.

The SQL generation pipeline in Figure 4 involves multiple steps, including: (i) entity extraction, (ii) attribute identification, (iii) relationship extraction, and (iv) SQL code generation.

Each step relies on LLM outputs that must conform to expected structures to ensure correct downstream processing.

We consider two alternative implementations of the same workflow. The former is a *Python-based implementation*, combining a general-purpose language with existing frameworks (e.g., LangChain for orchestration and Pydantic for output validation). The latter is an *LSL-based implementation*, where interactions, context management, and output constraints are expressed as first-class constructs in a dedicated prompt-programming language.

Rather than evaluating model accuracy, which depends on the underlying LLM, we focus on *engineering-relevant failure modes* that affect the reliability and maintainability of GenAIware systems. In particular, we consider:

- *Invalid output rate*: The frequency with which generated outputs do not conform to the expected structure (e.g., malformed SQL).
- *Pipeline breakage*: The extent to which invalid or unexpected outputs propagate and disrupt downstream steps.
- *Recovery effort*: The effort required to detect, localize, and correct failures during execution.

Analysis. The SQL generation scenario exposes several concrete failure cases: (i) *Malformed SQL syntax*, e.g., missing parentheses or invalid keywords, causing execution errors in downstream systems; (ii) *Inconsistent intermediate representations*, e.g., entities extracted without corresponding attributes, leading to incomplete schema definitions; (iii) *Missing or incorrect relationships*, resulting in logically inconsistent database schemas; and (iv) *Incorrect control decisions*, e.g., failure to detect whether a user query requires structured processing, leading to incorrect pipeline execution.

Table 2. Comparison of Failure Modes in Python-Based Frameworks and LSL

Scenario	Python-based framework	LSL
Malformed output	Runtime exception or invalid data propagation	Structurally prevented (or detected at interaction boundary)
Missing function call	Silent failure or incorrect branching	Enforced by interaction specification
Error localization	Distributed across program logic	Localized to interaction block
Recovery handling	Ad hoc, application-specific	Explicit, integrated in the execution model

In Python-based implementations, these failures are managed in an *ad hoc* manner. Moreover, output validation is typically performed *post hoc*, and control-flow decisions depend on possibly invalid or missing outputs. For instance, malformed SQL is typically detected only at execution time, while inconsistencies between entities and attributes may propagate across multiple steps before being detected. Validation logic (e.g., JSON schema checks or type validation) is applied after generation, and control flow decisions depend on outputs that may be missing, malformed, or semantically incorrect. Often these failures are caught by some mechanism that feeds back the error to the LLM in an attempt to fix it, leading to the so-called "Code Roulette" [73].

In contrast, LSL externalizes these concerns into the interaction model. Intermediate steps such as entity extraction and relationship detection are associated with explicit output constraints (e.g., lists, schemas), while final SQL generation can be constrained using formal grammars. As a result, structural violations can be detected or prevented through constrained decoding at the boundary of each interaction block, reducing the likelihood of error propagation.

Even in the absence of formal syntax, prompts exhibit implicit semantic structure (e.g., preambles, examples, templates), which can be inferred and operationalized [41]. LSL builds on this intuition by making such structure explicit and enforceable.

Table 2 summarizes key differences between the two approaches.

Discussion and Limitations. This application example shows that LSL improves robustness in structured generation scenarios such as SQL synthesis. By constraining outputs and isolating interactions, LSL reduces both the likelihood and the impact of structural failures. In particular, grammar-based constraints are effective in ensuring syntactically valid SQL, while explicit interaction boundaries prevent the propagation of invalid intermediate representations. Moreover, LSL improves interpretability by making interaction logic explicit.

However, improvements introduced by LSL come with tradeoffs. In fact, the improvements we reported in this example scenario are limited to *structural correctness*. LSL does not guarantee that the generated SQL is semantically correct with respect to the user's intent, nor that the extracted entities and relationships reflect the true meaning of the input specification. Moreover, introducing a DSL requires developers to learn new abstractions and integrate an additional execution layer. Additionally, LSL addresses structural and interaction-level failures but does not eliminate model-level issues such as hallucinations or ambiguity in natural language instructions.

Overall, the application example suggests that LSL shifts failure modes from *implicit and distributed* (in Python-based implementations) to *explicit and localized* (in prompt programs), thereby improving analyzability and maintainability without altering the underlying probabilistic nature of LLMs.

6 Opportunities

In this section, we report our considerations about the opportunities given by LSL for reliability (Section 6.1), robustness (Section 6.2), and trustworthiness (Section 6.3).

6.1 Reliability

Concerning the reliability, which we define as the ability of the LLM to "perform as required, without failure" [16], we focus on the use cases where the LLM has to generate a precise output, like an exact string out of k known strings (e.g., in classification), or, more importantly, it has to deal with the external environment (e.g., calling a function from an API or writing some code). Often, functionalities to interact with the environment are implemented with *function calling*: the LLM generates a response matching a specific pattern that triggers the function call. The information about the functions is provided to the LLM in the context as part of the task instructions.

Classification problems require that the LLM generate the correct name of the class, and the function calling approach requires that the model (1) would generate the call, (2) that it would generate it when required, and (3) that it would generate it correctly. However, despite the fine-tuning of the few-shot learning on samples for classification or requiring function calls, we cannot ensure that the output will comply with the desired one. As a result, a system that relies on the LLM taking care of, for example, calling a function to complete autonomously some workflow would lose reliability.

Conversely, using LSL can ensure that, for example, if a function call is needed, the scripted parts of the interaction will trigger that call. Moreover, the possibility to constrain the output will ensure that the parameters passed to the function call are syntactically correct. Similarly, if we need the LLM to generate some code at runtime (e.g., run some bash command to check the status of a server), we can ensure that it is at least syntactically correct and that the script is executed. As a result, we would have more autonomous LLMs running inside more reliable systems.

Despite the problem staying, the LLM introduces errors (as we further detail in Section 7.3). We can ensure that everything the LLM generates will comply with the required outputs to the following steps by constraining the output to follow specific patterns. This can at least ensure that one of the classes to choose from is actually selected or that a function is called whenever it is needed without failing because of, e.g., missing a `';'`. This avoids breaking down the entire pipeline and makes it more reliable.

Reliability

We plan to tackle the reliability of generated content by constraining the decoding to obey a given structure or to target only a known, possibly finite, set of outputs.

6.2 Robustness

LLMs are DNNs, and as such, they are designed to be robust, where robustness is intended as the ability to avoid spurious correlations and patterns [23, 48]. Nevertheless, these models are not error-proof, and often, ambiguous or poorly formulated inputs can cause them to fail. The same applies to use cases where there is the need for some multi-step reasoning or some additional or more updated information is needed.

A first measure to improve robustness in these cases is to automate the use of the techniques we presented in Section 2.2. For example, given a known classification task where there are some reference datasets available, LSL can offer support for automatically retrieving proper prompts and examples for few-shot learning or CoT reasoning. In the case of few-shot learning, it can also automatically tune the number of shots. Similarly, LSL can offer tools to automatically search trusted documents and knowledge bases to add useful information to the context and help the LLM generate the correct response. All these features are thought to improve the robustness by automating the use of techniques designed to help the accuracy of LLMs.

A complementary robustness measure, especially in the interaction with the external environment, can include error management. We can fall back to a semi-automatic process by introducing a human in the loop in case of failures. Otherwise, we can have the LLM try to fix the issue itself. If we can trace back to the point where the error was created, we can prompt the LLM to revise its response in light of the error message. In this sense, there already exist frameworks to use textual feedback (i.e., the error message) to have the LLM correct itself [87]. As before, integrating these solutions in LSL will not ensure the absence of errors, but it can improve the robustness of a system using the LLM.

Robustness

We plan to tackle robustness in the use of LLMs (i) by automating context management for few-shot learning, RAG, and CoT reasoning and (ii) by introducing error-handling subroutines (either with self-correction or with user feedback).

6.3 Trustworthiness

In the following, we discuss trustworthiness, defined as the ability of an LLM to "reflect characteristics such as truthfulness, safety, fairness, robustness, privacy, machine ethics, transparency, and accountability" [51] of applications interfacing with LLMs using LSL in terms of verification and validation (Section 6.3.1) and in terms of explainability (Section 6.3.2).

6.3.1 Verification and Validation. Introducing a structured overlay to wrap the access to the LLM with LSL opens the chance of using formal methods to verify and validate the scripted pipelines via model-checking tools. In fact, while we cannot have guarantees that the model will not make errors in the predicted output, the constraints on the interaction structure (e.g., branches in the workflow) and on the generated output (e.g., code) introduced by LSL are compatible with this kind of analysis. Using formal tools for verification and tools for validation allows us to add guarantees on the constraints imposed on the LLM and on the outcome of the scripts realized with LSL [16]. For instance, one can verify that the interaction scripted with LSL always terminates or that there are no unreachable or dead nodes in the computation graph. Looking at more concrete use cases, we can also verify the generated JSON to call a remote API always matches the API constraints by looking at the grammar constraining the LLM output.

Moreover, we can take into account the probabilistic nature of the LLM and, where applicable, integrate testing methodologies based on (1) fuzzying, adversarial, or evolutionary testing [30, 36] to expand the coverage automatically and (2) statistical tests, to provide a confidence bound on the results of the tests [36]. The latter would additionally allow the use of results on the model's accuracy on a specific task or sub-task and apply probabilistic model checking to measure the likelihood of certain behaviors or outcomes. For example, if we need to apply sentiment analysis on a piece of text as part of a pipeline and we have a reference dataset of documents, possibly coming from the same domain, we can use it to test the model, get the likelihood of committing a classification error, and use this information to verify properties of our system. Additionally, given insights on model calibration (i.e., how well the predicted probabilities match the model's confidence), we can perform runtime analysis for a specific input, extending the set of guarantees.

6.3.2 Explainability. Explainability with DNNs, in general, and LLMs, in particular, poses an important requirement to make AI-based applications more *transparent* [51]. Besides transparency, which helps users understand the application's behavior, explainability can also serve as a debugging tool by highlighting which parts of the prompt caused faulty output and allowing users to correct such parts, for example, by rephrasing ambiguous instructions.

For the scope of LSL, we see explainability adopted in two different ways. The former solution is to plug in external frameworks, thus reusing pre-existing tools and approaches [97]. The latter solution, more specific to LSL, is to use the structure in the interaction imposed by the script to trace back the outcome. The two proposed solutions are not mutually exclusive and can be combined to get better insights into what led to a specific result or prediction.

Trustworthiness

We plan to tackle trustworthiness in terms of (i) verification and validation, offering support for verification of the script code and probabilistic checking of the procedure described by the script, and (ii) explainability, offering support for internal solutions based on the script structure and external explainability tools.

7 Challenges

In this section, we report the challenges that we foresee to develop LSL in terms of technical difficulties (Section 7.1), standardization of the interfaces (Section 7.2), threats related to the use of LLMs (Section 7.3), and the ethical concerns (Section 7.4).

7.1 Technical Barrier

Developing a scripting language like LSL is complex, encompassing multiple SE areas, including language and API design, concurrency, and performance [70].

Designing the language with the appropriate level of abstraction is critical to ensure the right balance between expressivity, abstraction, and usability. The syntax should allow the integration of script code, input templating, and output constraining in a seamless way from the developer's point of view, easing the management of context and the interaction with the model. Particularly, output constraints, which are critical to reliability, should be analyzed carefully to allow specifying both simple constraints, like alternative completions, and more complex constraints, like the syntax of a programming language.

Similarly, designing the APIs to simplify the access to functionalities like likelihood prediction, generation, embedding, or to access other resources is fundamental to offer a complete suite to the developers. Thus, we need to understand which are the essential operations required to interact with the LLM and the environment. This is strictly connected to the standardization challenge we introduce in Section 7.2, as providing a unified interface to models and resources is essential to effectively communicate with all the components.

Regarding concurrency and, more generally, performance, given the computational demand of running an LLM, being able to automatically parallelize the passages in the pipeline and pack those requests that can be run concurrently in batches is essential. The concurrency aspects are not limited to a single process but also to concurrent processes characterized by different sessions and contexts, requiring careful management of which contexts are forwarded to the LLM and when to ensure optimal resource utilization. Another aspect to consider is the causal nature of LLMs, which enables the exploitation of caching mechanisms by reusing partially pre-processed sequences.

There are also aspects related to error handling, testing, verification, and validation. Integrating explainability with error management and testing can help track down and solve issues by understanding whether they are related to the script code, the LLM, or both. Similarly, designing the tools to apply (statistical) model checking to verify and validate the script code at compile time and runtime poses a non-trivial task, even if we want to resort to external tools.

Technical Barrier

We expect the main technical challenges to come from (i) the design of LSL to be sufficiently abstract and usable plus (ii) the implementation of the interpreter for LSL when addressing automatic LLM context management and resource usage optimization.

7.2 Standardization

Standardizing the interfaces is crucial for several reasons, including *interoperability* of different modules, *consistency and predictability* of behaviors, and *ease of integration*. In this context, we are interested in standardizing two main interfaces: (1) the interface to the LLM, obviously and (2) the interface to external resources, like datasets or documents and knowledge bases. The main challenge in these cases is either finding a good standard compatible with the most adopted solution or enforcing the adoption of a new one.

From an LLM perspective, the most adopted interface is the one provided by *HuggingFace* with the *Transformer* library [95]. Yet, there are other valid alternatives to self-host models, like llama.cpp [11], or to access them remotely via commercial APIs, like *watsonx.Ai* [9]. Moreover, larger commercial models like GPT [1] or Gemini [5] have their own separate interfaces. Large benchmark frameworks like *Big Bench* and *Language Model Evaluation Harness* [89] tried to propose wrappers to unify the interface to LLMs, yet not all APIs or inference servers support the necessary functionalities. In fact, it is crucial to have access to a scoring method to get the likelihoods predicted by the model to enforce decoding strategies (e.g., with a grammar) independently of the LLM implementation.

Standardizing the interface to external resources is crucial to automate many tasks. In fact, having standard interfaces to data sets can help automate the retrieval of samples for few-shot learning or templates for RAG. At the same time, having standard interfaces to document on knowledge bases can help with (i) making content generated by LLMs more factual and based on evidence and (ii) using LLMs for semantic ranking and re-ranking of search results. Given standardized templates for the tasks to which LLMs can be applied to this can enable the automation of hyper-parameter tuning (e.g., number of examples in few-shot learning) and the computation of performances for later verification and validation (see Section 6.3.1). To our knowledge, there are resources with a large and diverse pool of datasets covering different domains and tasks, but none provide a standard that allows for automated evaluation. Moreover, the introduction of these external sources leads to the problem of identifying trusted sources, which we discuss in Section 7.4.

Standardization

We expect standardization challenges to be mainly related to the interfaces (i) towards the LLM due to existing frameworks and (ii) to external resources, like data sets and document collections.

7.3 Inherent Threats

Throughout this article, we have highlighted that LSL cannot eliminate all threats and risks associated with the use of LLMs [16]. Nevertheless, some issues can be addressed (e.g., it is possible to enforce an output template to ensure the generated content complies with a standard), and the likelihood of others occurring can be reduced (e.g., using few-shot learning to improve classification accuracy).

Identifying and quantifying all the threats that cannot be neutralized poses a significant challenge. In fact, these threats depend on multiple factors, including the specific LLM being used and the specific tasks and subtasks that compose pipelines relying on that model.

Undoubtedly, the problem of hallucinations, either in terms of factuality or in terms of faithfulness, poses the biggest risk. The possibility of the LLM generating falsehoods or not sticking to some request makes the overall system seem unreliable. However, research in the context of hallucination detection and mitigation is advancing, although there is no definitive solution to the problem yet [50, 54].

Inherent Threats

We expect the main challenges in dealing with long-running threats related to the use of LLMs, namely erroneous predictions and hallucinations, to arise from the difficulty in identifying and quantifying the risk of their occurrence.

7.4 Ethical Concerns

Introducing AI in any application opens several ethical issues. LLMs in particular introduce a well-defined set of ethical and social problems one should account for when using this technology [94].

Technologies aimed at automating processes using LLMs often introduce problems related to *job replacement* (i.e., when human tasks are delegated to some piece of software) and *deskilling* (i.e., when humans lose their expertise on some tasks by stopping doing that task) [88]. The intention of LSL is to automate processes using LLMs, but instead of replacing human experts, the intention is to use their expertise to help automate such processes, leveraging their domain knowledge and experience. Moreover, human feedback and supervision are always crucial to ensure the proper functioning of the system. However, it is up to the developer of the specific application to give support to these aspects. LSL and its ecosystem, through scripting interactions with LLMs, can provide resources to easily integrate feedback and supervision components.

Additionally, content generated by LLMs exposes to risks related to *bias* and *toxicity* [44], besides *misinformation* and *hallucination*. These issues pose a threat towards the user who would be exposed to such content. While there is no proper way to ensure that such content is never generated, there are solutions to (i) test the predisposition of LLMs to generate such content [30] and measure quantitatively this predisposition and (ii) to recognize this kind of harmful content, preventing sharing it with the user. These tools should be integrated into the LSL ecosystem to quantify and mitigate these risks.

Ethical Concerns

We expect the main challenges related to ethical concerns to arise from problems (i) of replacement and deskilling, (ii) of bias and toxicity, and (iii) of misinformation and hallucinations.

8 Evaluation

In this section, we describe the setup of our qualitative evaluation (Section 8.1), the results of the evaluation (Section 8.2), and the mitigations for the possible threats to the validity (Section 8.3).

8.1 Design

To investigate the perceived need for the abstractions introduced by LSL, we conducted a survey-based study, a commonly adopted empirical method for capturing practitioners' opinions, experiences, and expectations through structured questionnaires. Following established guidelines for survey-based research in SE, the study was designed as a cross-sectional survey, aiming to collect

a snapshot of current practices and perceptions among practitioners working with GenAIware [25, 57].

The overall objective of our survey-based study was to assess the relevance and usefulness of the features supported by LSL from the perspective of practitioners. In line with this goal, the survey was structured to capture three complementary aspects: current development practices for GenAIware, the challenges encountered in practice, and the perceived usefulness and potential adoption of structured approaches to defining LLM interactions. This design is consistent with the use of survey-based studies to describe existing conditions and to assess attitudes and perceived needs within a target population [57].

The target population of the study consisted of practitioners and researchers with experience in developing or experimenting with GenAIware. Participants were recruited through convenience-based channels, including professional and academic networks, which is a common approach in exploratory surveys where the target population is difficult to enumerate precisely. While this approach does not support strict statistical generalization, it enabled the collection of informed opinions from possible stakeholders with relevant experience, which is appropriate given the exploratory nature of the study [57].

The survey instrument was designed iteratively to ensure clarity and relevance of the questions. The questionnaire includes closed questions, primarily based on Likert-scale responses. These questions were used to capture consistent assessments of perceived difficulty and usefulness across key dimensions. The design of the questions aimed to avoid ambiguity and bias, using clear and neutral wording and focusing on a single concept per item, as recommended in survey design guidelines [25, 57].

The questionnaire was organized to follow a logical progression. It first collected background information about participants' roles and experience with GenAIware, ensuring that respondents were in a position to meaningfully answer the subsequent questions. It then investigated current engineering practices and common pain points, providing a contextual grounding for the evaluation of LSL. Finally, the survey introduced the concept of a structured language for defining LLM interactions and elicited participants' perceptions regarding its usefulness, benefits, and potential limitations. This progression is intended to reduce bias by avoiding early priming effects and by ensuring that participants reflect on their own experience before evaluating the proposed approach.

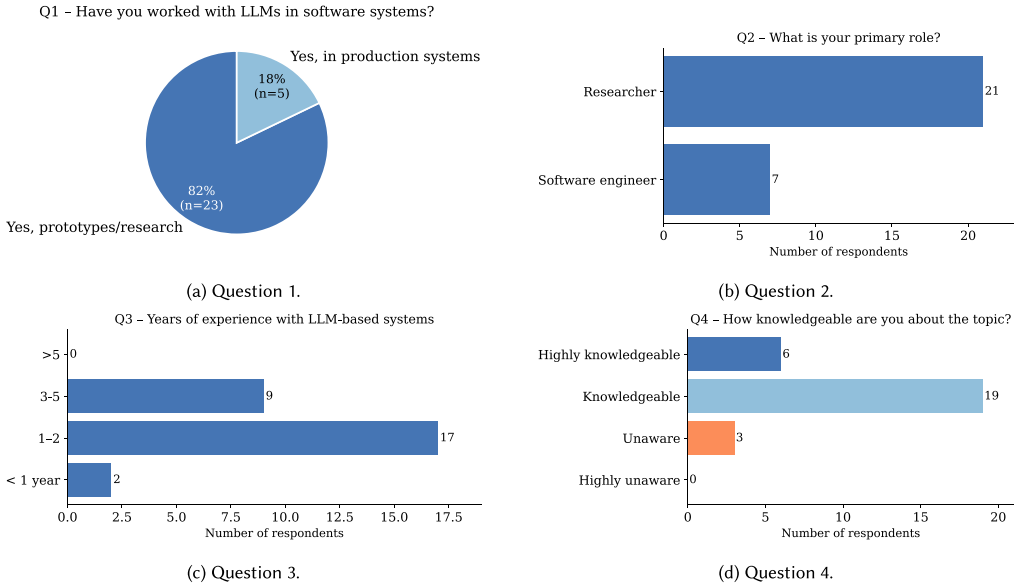
Data collection was performed through a self-administered online questionnaire, enabling participants to complete the survey asynchronously. The questionnaire was designed to be concise, requiring only a few minutes to complete, in order to reduce respondent burden and improve response quality, in line with best practices for survey-based research. Participation was voluntary and anonymous, and no incentives were provided. [57] The analysis of the collected data was performed looking at descriptive statistics of the closed questions.

Overall, the study design prioritized the collection of practitioner perspectives and perceived needs rather than objective performance measurements. As such, it provided qualitative and perception-based evidence regarding the relevance of the LSL approach, complementing the illustrative examples presented earlier in the article.

8.2 Results

The survey received 30 responses; 2 were discarded as the respondent has never worked with LLMs in software systems, $n = 28$ valid responses were retained for analysis. Hereafter we report descriptive results for each section of the questionnaire, with figures organized according to the survey sections.

Figure 5 summarizes the background profile of the respondents. All 28 participants reported having worked with LLMs in software systems (Q1): 23 (82%) in a research or prototyping context,



(a) Question 1.

(b) Question 2.

(c) Question 3.

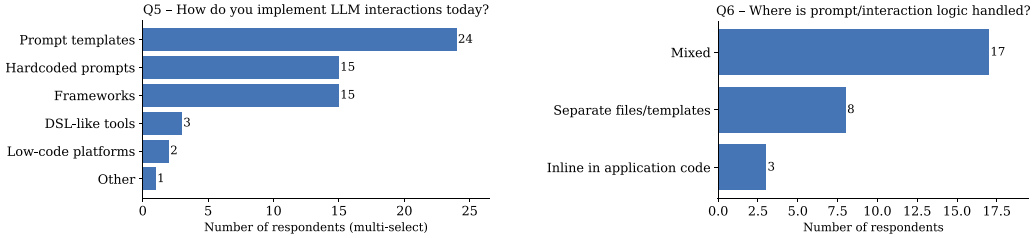
(d) Question 4.

Fig. 5. Screening and background section responses.

and 5 (18%) in production systems. Regarding the primary role (Q2), the sample comprised 21 researchers (75%) and 7 software engineers (25%). In terms of experience with GenAIware (Q3), more than half of the respondents (17, 61%) reported 1–2 years, and 9 (32%) reported 3–5 years; only 2 (7%) had less than 1 year, and none reported more than five. Regarding self-reported domain knowledge (Q4), 19 respondents (68%) described themselves as *knowledgeable* and 6 (21%) as *highly knowledgeable*; 2 respondents (11%) reported being *unaware* of the topic. Overall, the sample reflects a technically informed population with moderate-to-high familiarity with LLM-based development.

Figure 6 shows the responses regarding current development practices. Prompt templates (e.g., f-strings, Jinja2) are the most widely adopted technique (Q5), used by 24 respondents (86%), followed by hardcoded prompts embedded in application code (15, 54%) and LLM orchestration frameworks such as LangChain or Semantic Kernel (15, 54%). Only three respondents (11%) reported using DSL-like tools for structured generation, two respondents (7%) relied on low-code or no-code platforms, and only one respondent (4%) marked other approaches. This confirms that the use of structured, language-level abstractions remains uncommon in practice. Regarding where interaction logic is managed (Q6), the majority (17, 61%) handle it in a mixed fashion, combining different locations; eight respondents (28%) use separate files or templates, and three (11%) keep it inline in application code.

Figure 7 reports perceived pain points when developing GenAIware (Q7). The strongest agreement was recorded for Q7f (*Separating deterministic logic from LLM behavior would help*; $\bar{x} = 4.14$, SD = 0.71), with 82% of respondents agreeing or strongly agreeing, directly motivating one of the core design choices of LSL. Q7e (*Maintaining and evolving LLM interaction logic is challenging*; $\bar{x} = 3.71$) and Q7b (*Debugging LLM pipelines is difficult*; $\bar{x} = 3.68$) also received predominantly positive agreement, with 61% and 57% of respondents rating them *Agree* or *Strongly agree*, respectively. Q7c (*Invalid outputs propagate and break downstream components*; $\bar{x} = 3.50$) showed similarly high agreement (64%), while Q7d (*Hard to enforce structure or constraints*; $\bar{x} = 3.18$) drew more mixed responses. In contrast, Q7a (*Prompt logic is difficult to understand and reason about*; $\bar{x} = 2.71$) was



(a) Question 5. (b) Question 6.

Fig. 6. Current practice section responses.

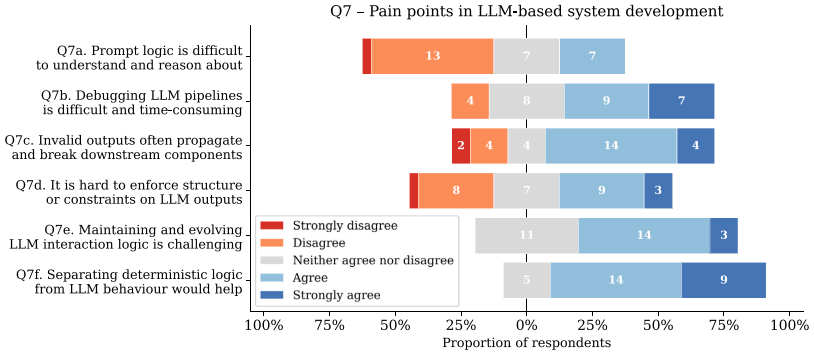


Fig. 7. Pain points section responses—Question 7.

the only item where a majority (50%) disagreed, suggesting that practitioners do not yet perceive the readability of prompt text itself as a primary challenge.

Question 8 served as an embedded attention check (see Section 8.3); all 20 retained respondents answered it correctly, confirming the attentiveness of the sample.

Figure 8 shows the perceived usefulness of specific LSL features (Q9, rated 1–5 with higher values indicating greater usefulness). Built-in validation of outputs (Q9d) received the highest mean rating ($\bar{x} = 4.32$, SD = 0.98), with 82% of respondents assigning a score of 4 or 5. Output constraints via schemas or grammars (Q9c) were also rated highly ($\bar{x} = 4.14$), with 79% of respondents scoring 4 or above. The separation between LLM calls and control logic (Q9e; $\bar{x} = 3.82$) and an explicit representation of interaction workflows (Q9a; $\bar{x} = 3.54$) similarly scored above the neutral midpoint. Static analysis of interaction logic (Q9b) received the lowest mean rating ($\bar{x} = 3.36$), though still above neutral, with 46% of respondents rating it 4 or above. Overall, all five features were rated above neutral, indicating broad appreciation of the LSL feature set.

Figure 9 reports participants' reactions to the LSL concept. Regarding overall perceived usefulness (Q10), the mean rating was 3.70 (SD = 0.80, median = 4), with 61% of respondents assigning a score of 4 or 5 and no respondent rating the concept as not useful at all (1). Regarding expected benefits (Q11), respondents were broadly positive: ease of maintenance (Q11c; $\bar{x} = 3.82$) and improvements to testing and verification (Q11d; $\bar{x} = 3.82$) received the highest endorsement, with 68% of respondents agreeing or strongly agreeing on both items. Improvements to debugging (Q11a; $\bar{x} = 3.54$) were supported by 54% of respondents, and reliability improvements (Q11b; $\bar{x} = 3.43$) by 39%. The two negatively framed items, serving as internal consistency indicators, showed consistently low agreement: Q11e (*No benefit is expected*; $\bar{x} = 1.93$) was rejected by 75% of respondents, and Q11f (*Finding bugs will be harder*; $\bar{x} = 2.32$) by 61%, confirming coherent response

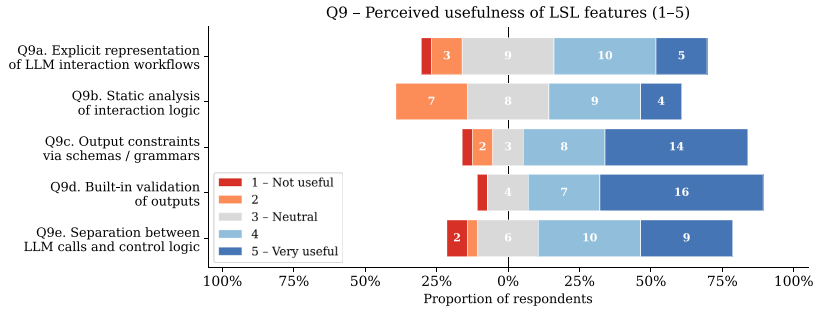


Fig. 8. Desired capabilities section responses—Question 9.

patterns. Regarding adoption concerns (Q12), integration with existing tools (Q12b; $\bar{x} = 4.18$, SD = 0.86) emerged as the primary concern, with 86% of respondents reporting being *concerned* or *very concerned*. Limited flexibility of the DSL (Q12d; $\bar{x} = 3.89$) was the second most prominent concern (75% concerned or very concerned). The learning curve (Q12a; $\bar{x} = 3.18$) and the question of whether such a language is actually necessary (Q12e; $\bar{x} = 2.79$) raised moderate concerns, while performance overhead (Q12c; $\bar{x} = 2.82$) showed the most mixed distribution of responses.

Finally, a Mann-Whitney U test comparing researchers ($n = 21$) and software engineers ($n = 7$) found no statistically significant differences across any of the four construct blocks (Q7: $U = 65.0$, $p = 0.67$; Q9: $U = 83.5$, $p = 0.61$; Q11: $U = 73.0$, $p = 1.00$; Q12: $U = 45.5$, $p = 0.14$), indicating consistent perceptions across roles; statistical power is however limited by the sample size, and these results should be interpreted accordingly.

8.3 Threats Mitigation

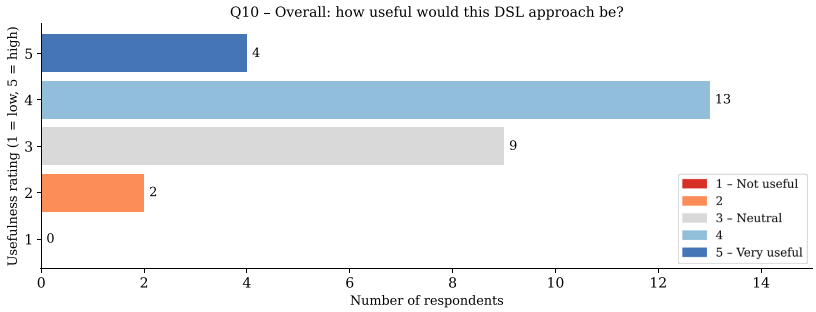
To ensure the integrity of the collected data and mitigate the risk of careless responding, we adopted several established quality control mechanisms. Given the relatively small scale and limited length of the survey instrument, we did not apply question randomization, as fatigue and ordering effects were considered unlikely to substantially impact participant responses. Instead, we focused on incorporating *internal consistency* and *attention checks* directly into the questionnaire design and subsequent analysis.

Specifically, one of the survey items was intentionally negatively framed (with respect to another item in the same group) to allow the identification of inconsistent response patterns, such as uniformly selecting the same agreement level across all questions. In addition, an explicit attention check was embedded within the survey, asking participants to select a particular response option to confirm that they were carefully reading the questions. During data analysis, we further examined response reliability by computing the intra-individual variance across all answers; submissions exhibiting near-zero variance, indicative of repetitive or inattentive behavior, were excluded from the final dataset. We also assessed the internal consistency of the measured constructs using Cronbach's alpha.

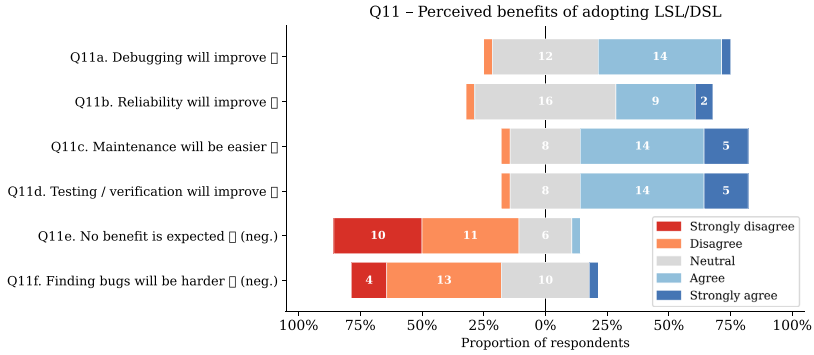
To analyze differences in perceived alignment across roles, we applied a Mann-Whitney U test. We selected this tests because the collected responses represented only two roles, *Researcher* ($n = 21$) and *Software engineer* ($n = 7$), with no respondent selecting any of the remaining options.

9 Road Map

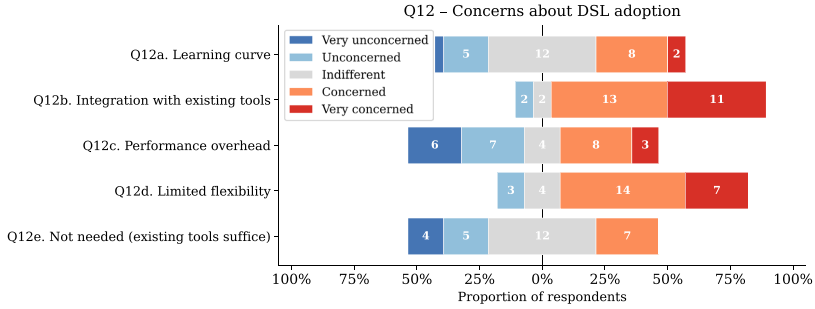
Building on the opportunities identified in Section 6 and the challenges outlined in Section 7, we envision a broad research road map towards reliable, robust, and trustworthy GenAIware. This road map, visualized in Figure 10, outlines research directions for the community at large and highlights



(a) Question 10



(b) Question 11



(c) Question 12

Fig. 9. Reaction to LSL concept section responses.

how language design, API development, interpreter infrastructures, usability, and ethical oversight must come together.

Foundational Work. The development of languages and abstractions will allow for structured interactions with LLMs. This includes the design of DSLs such as LSL, which can provide formalisms for scripting, constraining, and reasoning about LLM interactions. The community will need to explore various levels of abstraction, ranging from low-level APIs for fine-grained control to high-level declarative interfaces that prioritize usability and accessibility. Alongside the language itself, the definition of open and standardized APIs is critical. Such interfaces should unify access to prompt management, structured outputs, context handling, and integration with external resources such as datasets or knowledge bases, thereby fostering interoperability and extensibility.

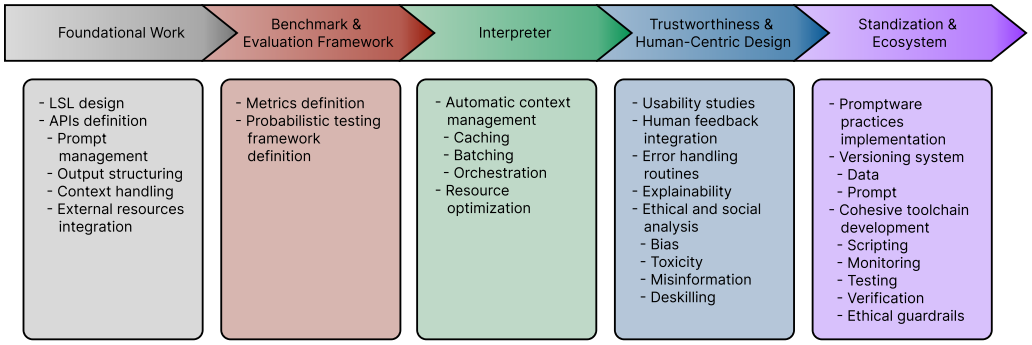


Fig. 10. Road map to the development of LSL.

Benchmarks and Evaluation Frameworks. Improving the reliability and robustness of GenAIware requires a shift from ad-hoc testing to systematic evaluation practices. Research is needed to define benchmarks and metrics that extend beyond model accuracy and capture safety, correctness, and alignment with user expectations. Evaluation should not be limited to LLMs in isolation but applied to the pipelines and applications in which they are embedded. Probabilistic testing, adversarial evaluation, and methods for runtime validation of generated outputs are especially promising directions. By establishing common benchmarks and frameworks, the community can enable more rigorous comparisons across tools and systems, foster reproducibility, and accelerate progress.

Interpreters. Interpreter infrastructures form the backbone of scripted LLM interactions and require dedicated research attention. Efficient interpreters should be capable of managing context automatically, exploiting opportunities for caching and batching, and orchestrating parallel execution where tasks allow. They must also balance usability and performance, providing developers with intuitive abstractions while optimizing resource utilization under the hood. This line of research naturally connects with the design of compiler and runtime systems yet introduces unique challenges related to the causal nature of LLMs and the integration of external resources. An interpreter that is both efficient and transparent will be decisive in making scripted LLM systems practical at scale.

Trustworthiness and Human-Centric Design. Usability studies, mechanisms for domain experts to shape system behavior, and frameworks for ensuring meaningful human control should become an integral part of GenAIware. Human feedback must not be treated as an afterthought but rather be integrated systematically to guide error handling and evaluation. Explainability tools should be embedded into workflows to allow developers and users to trace outputs, debug prompts, and build confidence in the system's behavior. At the same time, ethical and social concerns like bias, toxicity, misinformation, deskilling, and job displacement require continuous monitoring and proactive mitigation strategies. Addressing these aspects is essential to ensure trustworthiness and long-term societal acceptance.

Standardization and Ecosystems. Standardization represents another key frontier. Just as MLOps practices evolved to bring order to machine learning pipelines, the community now needs standards for promptware, prompt and data versioning, and lifecycle management of GenAIware. Without such common ground, development practices will remain fragmented and inconsistent. Beyond formal standards, an interoperable ecosystem of tools must emerge. Instead of today's patchwork of isolated frameworks, we need cohesive toolchains that integrate scripting, monitoring, testing, verification, and ethical safeguards into the everyday workflows of developers.

Beyond this road map, our long-term vision is to embed LLM scripting and interaction into the broader DevOps lifecycle. In such a future, prompts, data, and scripts will be treated as first-class citizens alongside code, benefiting from the same rigor in versioning, testing, and verification. Verification, monitoring, and human oversight will be embedded throughout development and deployment pipelines, ensuring that reliability and transparency are not afterthoughts but built-in guarantees. This vision requires the construction of sustainable ecosystems in which GenAIware can be designed, deployed, and maintained with the same degree of trust, robustness, and societal accountability as traditional software.

10 Conclusion

LLMs represent valuable assets to build AI-powered applications. Nevertheless, they come with several risks and limitations that prevent freely adopt them into fully automated pipelines and workflows. Throughout this article, we have argued that SE plays a decisive role in addressing these limitations.

The core idea is to develop a DSL that works as a scripting language, namely LSL, to automate the templating of the input prompt and the constraining of the generated completions of the LLM to enforce guarantees on the LLM behavior. An interpreter complements LSL in multiple ways, like (1) managing the LLM context, (2) managing the interfaces towards the model and the external resources, (3) managing error correction subroutines, (4) running consistency checks, and (5) producing explanations when needed.

Building on that, we outlined a road map that spans foundational directions in language and API design, the development of benchmarks and evaluation frameworks, the creation of efficient interpreter infrastructures, the integration of human-centric feedback and usability studies, and the establishment of standards and ecosystems that can bring order to today's fragmented practices. This agenda emphasizes that progress requires not only technical innovations but also attention to social, ethical, and organizational challenges so that trustworthiness, transparency, and societal accountability are built into LLM-powered systems (GenAIware) from the outset.

Our survey study, conducted with 28 practitioners and researchers experienced in GenAIware development, provides initial empirical support for the relevance of LSL. Respondents broadly confirmed the practical pain points targeted by LSL: 82% agreed that separating deterministic logic from LLM behavior would be beneficial (Q7f; $\bar{x} = 4.14$), and maintaining and debugging LLM interaction logic were rated as challenging by a clear majority. All five proposed LSL features were perceived as useful above the neutral midpoint, with built-in output validation and constraint enforcement receiving the highest ratings ($\bar{x} = 4.32$ and 4.14, respectively). Overall, 61% of respondents rated the DSL concept as useful or very useful (score ≥ 4 on a 1–5 scale; $\bar{x} = 3.68$), while integration with existing toolchains and language flexibility emerged as the primary adoption concerns to be addressed in future work.

Data Availability

The replication package of our experiments is available at https://github.com/vincenzo-scotti/tosem_roadmap_to_2030.

Acknowledgments

Some parts of our work were assisted by Generative AI. We used Anthropic's Claude Opus 4.7 and Microsoft's Copilot (based on OpenAI's ChatGPT 5 model family) to help us refine the manuscript and prepare the visualization of the evaluation outcome.

References

[1] OpenAI. 2020. OpenAI API Reference: Documentation for Accessing and Interacting with OpenAI Models. Retrieved from https://platform.openai.com/docs/api-reference/

[2] LangChain. 2022. LangChain Documentation: Introduction to Building Applications with Language Models. Retrieved from https://python.langchain.com/docs/introduction/

[3] GitHub. 2023. CrewAI: Framework for Building Autonomous Multi-Agent LLM Workflows. Retrieved from https://github.com/crewAIInc/crewAI

[4] GitHub. 2023. FlowiseAI: Low-Code Platform for Building LLM-Powered Applications Visually. Retrieved from https://github.com/FlowiseAI/Flowise

[5] Google AI Studio. 2023. Access Gemini Models and API Tools for AI Development. Retrieved from https://ai.google.dev/

[6] Guardrails AI. 2023. Framework for Validating and Controlling LLM Outputs. Retrieved from https://github.com/guardrails-ai/guardrails

[7] GitHub. 2023. Guidance: Programmable Control for Language Models Using Constraints and Templates. Retrieved from https://github.com/guidance-ai/guidance

[8] Haystack by Deepset. 2023. Framework for Developing End-to-End NLP Applications with LLMs and RAG. Retrieved from https://github.com/deepset-ai/haystack

[9] IBM watsonx. 2023. AI: Enterprise Studio for Training, Validating, and Deploying AI Models. Retrieved from https://www.ibm.com/products/watsonx-ai

[10] Llama-cpp-python. 2023. Python Bindings for Llama.cpp for Easy Integration. Retrieved from https://github.com/abetlen/llama-cpp-python

[11] Llama.cpp. 2023. Efficient LLM Inference in C/C++ for LLaMA Models on CPUs. Retrieved from https://github.com/ggerganov/llama.cpp

[12] GitHub. 2023. LlamaIndex: Data Framework for Connecting LLMs with External Data Sources. Retrieved from https://github.com/run-llama/llama_index

[13] GitHub. 2023. Semantic Kernel: Microsoft's Framework for Integrating LLMs with Traditional Programming Models. Retrieved from https://github.com/microsoft/semantic-kernel

[14] Altaf Allah Abbassi, Léuson M. P. da Silva, Amin Nikanjam, and Foutse Khomh. 2025. A taxonomy of inefficiencies in LLM-generated Python code. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME '25)*. IEEE, 393–404. DOI: https://doi.org/10.1109/ICSME64153.2025.00043

[15] Deepak Bhaskar Acharya, Karthigeyan Kuppan, and B. Divya. 2025. Agentic AI: Autonomous intelligence for complex goals—A comprehensive survey. *IEEE Access* 13 (2025), 18912–18936. DOI: https://doi.org/10.1109/ACCESS.2025.3532853

[16] NIST AI. 2023. Artificial Intelligence Risk Management Framework. (AI RMF 10). DOI: https://doi.org/10.6028/NIST.AI.100-1

[17] Domenico Amalfitano, Andreas Metzger, Marco Autili, Tommaso Fulcini, Tobias Hey, Jan Keim, Patrizio Pelliccione, Vincenzo Scotti, Anne Koziolek, Raffaela Mirandola, et al. 2026. A research roadmap for augmenting software engineering processes and software products with generative AI. *ACM Transactions on Software Engineering and Methodology* (January 2026). DOI: https://doi.org/10.1145/3788879

[18] Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: A family of highly capable multimodal models. arXiv:2312.11805. Retrieved from https://arxiv.org/abs/2312.11805

[19] Scott Barnett, Stefanus Kurniawan, Srikanth Thudumu, Zach Brannelly, and Mohamed Abdelrazek. 2024. Seven failure points when engineering a retrieval augmented generation system. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI (CAIN '24)*. Jane Cleland-Huang, Jan Bosch, Henry Muccini, and Grace A. Lewis (Eds.), ACM, 194–199. DOI: https://doi.org/10.1145/3644815.3644945

[20] Nora Belrose, Zach Furman, Logan Smith, Danny Halawi, Igor Ostrovsky, Lev McKinney, Stella Biderman, and Jacob Steinhardt. 2023. Eliciting latent predictions from transformers with the tuned lens. arXiv:2303.08112. Retrieved from https://arxiv.org/abs/2303.08112

[21] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. DeepSeek LLM: Scaling open-source language models with longtermism. arXiv:2401.02954. Retrieved from https://arxiv.org/abs/2401.02954

[22] Antonio Brogi, Ahmad Ibrahim, Jacopo Soldani, José Carrasco, Javier Cubo, Ernesto Pimentel, and Francesco D'Andria. 2016. SeaClouds: An Open Reference Architecture for Multi-cloud Governance. In *Proceedings of the 10th European Conference on Software Architecture (ECSA'16)*. Bedir Tekinerdogan, Uwe Zdun, and Muhammad Ali Babar (Eds.), Lecture Notes in Computer Science, Vol. 9839, Springer, 334–338. DOI: https://doi.org/10.1007/978-3-319-48992-6_25

[23] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Advances in*

Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020 (NeurIPS '20). Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). Retrieved from https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[24] Nicolò Brunello, Davide Rigamonti, Andrea Sassella, Vincenzo Scotti, and Mark James Carman. 2025. InTraVisTo: Inside transformer visualisation tool. arXiv:2507.13858. Retrieved from https://arxiv.org/abs/2507.13858

[25] Radu Calinescu, Raffaela Mirandola, Diego Perez-Palacin, and Danny Weyns. 2020. Understanding uncertainty in self-adaptive systems. In *Proceedings of the IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS '20)*. IEEE, 242–251. DOI: https://doi.org/10.1109/ACSOS49614.2020.00047

[26] Xiang Chen, Chaoyang Gao, Chunyang Chen, Guangbei Zhang, and Yong Liu. 2025. An empirical study on challenges for LLM application developers. *ACM Transactions on Software Engineering Methodology* 34, 7, Article 205 (2025), 1–37. DOI: https://doi.org/10.1145/3715007

[27] Zhenpeng Chen, Chong Wang, Weisong Sun, Xuanzhe Liu, Jie M. Zhang, and Yang Liu. 2025. Promptware engineering: Software engineering for LLM prompt development. arXiv:2503.02400. Retrieved from https://arxiv.org/abs/2503.02400

[28] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2022. Scaling instruction-finetuned language models. arXiv:2210.11416. Retrieved from https://arxiv.org/abs/2210.11416

[29] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tai, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2024. Scaling instruction-finetuned language models. *Journal of Machine Learning Research* 25, Article 70 (2024), 1–53. Retrieved from https://jmlr.org/papers/v25/23-0870.html

[30] Simone Corbo, Luca Bancale, Valeria De Gennaro, Livia Lestingi, Vincenzo Scotti, and Matteo Camilli. 2025. How toxic can you get? Search-based toxicity testing for large language models. *IEEE Transactions on Software Engineering* 51, 11 (2025), 3056–3071. DOI: https://doi.org/10.1109/TSE.2025.3607625

[31] Tri Dao. 2024. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *Proceedings of the 12th International Conference on Learning Representations (ICLR '24)*. OpenReview.net. Retrieved from https://openreview.net/forum?id=mZn2Xyh9Ec

[32] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022 (NeurIPS '22)*. Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh. (Eds.). Retrieved from http://papers.nips.cc/paper_files/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html

[33] DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, et al. 2024. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model. arXiv:2405.04434. Retrieved from https://arxiv.org/abs/2405.04434

[34] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, et al. 2024. DeepSeek-V3 technical report. arXiv:2412.19437. Retrieved from https://arxiv.org/abs/2412.19437

[35] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT '19) (Long and Short Papers)*. Jill Burstein, Christy Doran, and Thamar Solorio (Eds.), Vol. 1, Association for Computational Linguistics, 4171–4186. DOI: https://doi.org/10.18653/V1/N19-1423

[36] Felix Dobslaw, Robert Feldt, Juyeon Yoon, and Shin Yoo. 2025. Challenges in testing large language model based software: A faceted taxonomy. arXiv:2503.00481. Retrieved from https://arxiv.org/abs/2503.00481

[37] Felix Dobslaw, Robert Feldt, Juyeon Yoon, and Shin Yoo. 2026. Challenges in testing large language model based software: A faceted taxonomy. *ACM Transactions on Software Engineering Methodology* (April 2026). DOI: https://doi.org/10.1145/3806396

[38] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 herd of models. arXiv:2407.21783. Retrieved from https://arxiv.org/abs/2407.21783

[39] Beyza Eken, Samodha Pallewatta, Nguyen Tran, Ayse Tosun, and Muhammad Ali Babar. 2025. A multivocal review of MLOps practices, challenges and open issues. *ACM Computing Surveys* 58, 2, Article 39 (Sept. 2025), 35. DOI: https://doi.org/10.1145/3747346

[40] Timothy Elvira, Tyler Thomas Procko, and Omar Ochoa. 2024. Requirements elicitation for machine learning applications: A research preview. In *Proceedings of the 2024 Conference on AI, Science, Engineering, and Technology (AIxSET '24)*, 218–221. DOI: https://doi.org/10.1109/AIxSET62544.2024.00042

[41] Alexander J. Fiannaca, Chinmay Kulkarni, Carrie J. Cai, and Michael Terry. 2023. Programming without a programming language: Challenges and opportunities for designing developer tools for prompt programming. In *Proceedings of the*

Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23). ACM, New York, NY, Article 235, 7. DOI: https://doi.org/10.1145/3544549.3585737

[42] International Organization for Standardization and International Electrotechnical Commission. 2023. *Information Technology—Artificial Intelligence—AI System Life Cycle Processes*. Standard. International Organization for Standardization.

[43] Raffaello Fornasiere, Nicolò Brunello, Vincenzo Scotti, and Mark Carman. 2024. Medical information extraction with large language models. In *Proceedings of the 7th International Conference on Natural Language and Speech Processing (ICNLSP '24)*. Mourad Abbas and Abed Alhakim Freihat (Eds.), Association for Computational Linguistics, 456–466. Retrieved from https://aclanthology.org/2024.icnlsp-1.47/

[44] Isabel O. Gallegos, Ryan A. Rossi, Joe Barrow, Md Mehrab Tanjim, Sungchul Kim, Franck Dernoncourt, Tong Yu, Ruiyi Zhang, and Nesreen K. Ahmed. 2024. Bias and fairness in large language models: A survey. *Computational Linguistics* 50, 3 (Sept. 2024), 1097–1179. DOI: https://doi.org/10.1162/coli_a_00524. Retrieved from https://direct.mit.edu/coli/article-pdf/50/3/1097/2471010/coli_a_00524.pdf

[45] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. arXiv:2312.10997. Retrieved from https://arxiv.org/abs/2312.10997

[46] Saibo Geng, Martin Josifoski, Maxime Peyrard, and Robert West. 2023. Grammar-constrained decoding for structured NLP tasks without finetuning. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP '23)*. Houda Bouamor, Juan Pino, and Kalika Bali (Eds.), Association for Computational Linguistics, 10932–10952. DOI: https://doi.org/10.18653/V1/2023.EMNLP-MAIN.674

[47] GitHub. 2021. Introducing GitHub Copilot: Your AI pair programmer. Retrieved from https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/

[48] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. 2016. *Deep Learning*. MIT Press. Retrieved from http://www.deeplearningbook.org/

[49] James D. Hollan, Edwin L. Hutchins, and David Kirsh. 2000. Distributed cognition: Toward a new foundation for human-computer interaction research. *ACM Transactions on Computer–Human Interaction* 7, 2 (2000), 174–196. DOI: https://doi.org/10.1145/353485.353487

[50] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2023. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. arXiv:2311.05232. Retrieved from https://arxiv.org/abs/2311.05232

[51] Yue Huang, Lichao Sun, Haoran Wang, Siyuan Wu, Qihui Zhang, Yuan Li, Chujie Gao, Yixin Huang, Wenhan Lyu, Yixuan Zhang, et al. 2024. Position: TrustLLM: Trustworthiness in large language models. In *Proceedings of the 41st International Conference on Machine Learning (ICML '24)*. OpenReview.net. Retrieved from https://openreview.net/forum?id=bWUU0LwwMp

[52] Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. GPT-4o system card. arXiv:2410.21276. Retrieved from https://arxiv.org/abs/2410.21276

[53] Hakan Inan, Kartikeya Upasani, Jianfeng Chi, Rashi Rungta, Krithika Iyer, Yuning Mao, Michael Tontchev, Qing Hu, Brian Fuller, Davide Testuggine, et al. 2023. Llama guard: LLM-based input-output safeguard for human-AI conversations. arXiv:2312.06674. Retrieved from https://arxiv.org/abs/2312.06674

[54] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, Pascale Fung, et al. 2023. Survey of hallucination in natural language generation. *ACM Computing Surveys* 55, 12, Article 248 (2023), 1–38. DOI: https://doi.org/10.1145/3571730

[55] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. arXiv:2310.06825. Retrieved from https://arxiv.org/abs/2310.06825

[56] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. arXiv:2401.04088. Retrieved from https://arxiv.org/abs/2401.04088

[57] Barbara A. Kitchenham and Shari Lawrence Pfleeger. 2008. Personal opinion surveys. In *Guide to Advanced Empirical Software Engineering*. Forrest Shull, Janice Singer, and Dag I. K. Sjøberg (Eds.), Springer, 63–92. DOI: https://doi.org/10.1007/978-1-84800-044-5_3

[58] Donald E. Knuth. 1984. Literate programming. *The Computer Journal* 27, 2 (1984), 97–111. DOI: https://doi.org/10.1093/COMJNL/27.2.97

[59] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022 (NeurIPS '22)*. Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho,

et al. (Eds.). Curran Associates, Inc., 22199–22213. Retrieved from http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[60] Jonathan Laurent and André Platzer. 2025. Oracular programming: A modular foundation for building LLM-enabled software. arXiv:2502.05310. Retrieved from https://arxiv.org/abs/2502.05310

[61] Jenny T. Liang, Melissa Lin, Nikitha Rao, and Brad A. Myers. 2025. Prompts are programs too! Understanding how developers build software containing prompts. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 1591–1614. DOI: https://doi.org/10.1145/3729342

[62] LMSYS. 2023. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. Retrieved from https://lmsys.org/blog/2023-03-30-vicuna/

[63] Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, et al. 2025. A survey of context engineering for large language models. arXiv:2507.13334. Retrieved from https://arxiv.org/abs/2507.13334

[64] Marjan Mernik. 2017. Domain-specific languages: A systematic mapping study. In *SOFSEM 2017: Theory and Practice of Computer Science—43rd International Conference on Current Trends in Theory and Practice of Computer Science*. Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria (Eds.), Lecture Notes in Computer Science, Springer, 464–472. DOI: https://doi.org/10.1007/978-3-319-51963-0_36

[65] Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, et al. 2024. Gemma: Open models based on Gemini research and technology. arXiv:2403.08295. Retrieved from https://arxiv.org/abs/2403.08295

[66] Shervin Minaee, Tomás Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large language models: A survey. arXiv:2402.06196. Retrieved from https://arxiv.org/abs/2402.06196

[67] Nadia Nahar, Christian Kästner, Jenna L. Butler, Chris Parnin, Thomas Zimmermann, and Christian Bird. 2025. Beyond the comfort zone: Emerging solutions to overcome challenges in integrating LLMs into software products. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (SEIP@ICSE '25)*. IEEE, 516–527. DOI: https://doi.org/10.1109/ICSE-SEIP66354.2025.00051

[68] Arshaan Nazir, Thadaka Kalyan Chakravarthy, David Amore Cecchini, Rakshit Khajuria, Prikshit Sharma, Ali Tarik Mirik, Veysel Kocaman, and David Talby. 2024. LangTest: A comprehensive evaluation library for custom LLM and NLP models. *Software Impacts* 19 (2024), 100619. DOI: https://doi.org/10.1016/J.SIMPA.2024.100619

[69] Donald A. Norman. 1993. *Things That Make Us Smart: defending Human Attributes in the Age of the Machine*. Addison-Wesley Longman Publishing Co., Inc.

[70] Robert Nystrom. 2021. *Crafting Interpreters*. Genever Benning. Retrieved from https://craftinginterpreters.com/

[71] Hartmut Obendorf. 2009. *Minimalism, Industrial Design and HCI*. Springer London, London, 81–95. DOI: https://doi.org/10.1007/978-1-84882-371-6_4

[72] OpenAI. 2023. GPT-4 technical report. arXiv:2303.08774. Retrieved from https://arxiv.org/abs/2303.08774

[73] Andrei Paleyes, Radzim Sendyka, Diana Robinson, Christian Cabrera, and Neil D. Lawrence. 2064. Code roulette: How prompt variability affects LLM code generation. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '26)*. ACM, New York, NY. Retrieved from https://arxiv.org/abs/2506.10204

[74] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-aligned decoding. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024 (NeurIPS '24)*. Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.), ACM, 24547–24568. Retrieved from http://papers.nips.cc/paper_files/paper/2024/hash/2bdc2267c3d7d01523e2e17ac0a754f3-Abstract-Conference.html

[75] Emanuele Pucci, Ludovica Piro, Salvatore Andolina, and Maristella Matera. 2024. From conversational web to inclusive conversations with LLMs. In *Proceedings of the 2024 International Conference on Advanced Visual Interfaces (AVI '24)*. Cristina Conati, Gualtiero Volpe, and Ilaria Torre (Eds.), ACM, Article 87, 1–3. DOI: https://doi.org/10.1145/3656650.3656739

[76] Markus N. Rabe and Charles Staats. 2021. Self-attention does not need O(n^2) memory. arXiv:2112.05682. Retrieved from https://arxiv.org/abs/2112.05682

[77] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving Language Understanding by Generative Pre-Training. Retrieved from https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[78] Traian Rebedea, Razvan Dinu, Makesh Narsimhan Sreedhar, Christopher Parisien, and Jonathan Cohen. 2023. NeMo guardrails: A toolkit for controllable and safe LLM applications with programmable rails. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing—System Demonstrations (EMNLP '23)*. Yansong Feng and Els Lefever (Eds.), Association for Computational Linguistics, 431–445. DOI: https://doi.org/10.18653/V1/2023.EMNLP-DEMO.40

[79] Petko Georgiev, Ving Ian Lei, Ryan Burnell, Libin Bai, Anmol Gulati, Garrett Tanzer, Damien Vincent, Zhufeng Pan, Shibo Wang, Soroosh Mariooryad, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv:2403.05530. Retrieved from https://arxiv.org/abs/2403.05530

[80] Morgane Rivière, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, Johan Ferret, et al. 2024. Gemma 2: Improving open language models at a practical size. arXiv:2408.00118. Retrieved from https://arxiv.org/abs/2408.00118

[81] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code Llama: Open foundation models for code. arXiv:2308.12950. Retrieved from https://arxiv.org/abs/2308.12950

[82] Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, et al. 2022. Multitask prompted training enables zero-shot task generalization. In *Proceedings of the 10th International Conference on Learning Representations (ICLR '22)*. Retrieved from OpenReview.net. https://openreview.net/forum?id=9Vrb9D0WI4

[83] Nicolas Schuler. 2027. Architecting self-adaptive systems with learned and symbolic components. In *Proceedings of Software Architecture. ECSA 2026 Tracks and Workshops*. Springer Nature Switzerland, Cham, 1–8. DOI: https://doi.org/10.5445/IR/1000195259

[84] Vincenzo Scotti, Licia Sbattella, and Roberto Tedesco. 2024. A primer on Seq2Seq models for generative chatbots. *ACM Computing Surveys* 56, 3, Article 75 (2024), 1–58. DOI: https://doi.org/10.1145/3604281

[85] Yuchen Shao, Yuheng Huang, Jiawei Shen, Lei Ma, Ting Su, and Chengcheng Wan. 2025. Are LLMs correctly integrated into software systems? In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE '25)*. IEEE, 1178–1190. DOI: https://doi.org/10.1109/ICSE55347.2025.00204

[86] Kensen Shi, Deniz Altinbüken, Saswat Anand, Mihai Christodorescu, Katja Grünwedel, Alexa Koenings, Sai Naidu, Anurag Pathak, Marc Rasi, Fredde Ribeiro, et al.. 2025. Natural language outlines for code: Literate programming in the LLM era. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25)*. Leonardo Montecchi, Jingyue Li, Denys Poshyvanyk, and Dongmei Zhang (Eds.), ACM, 150–161. DOI: https://doi.org/10.1145/3696630.3728541

[87] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023 (NeurIPS '23)*. Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (Eds.), Neural Information Processing Systems Foundation, 8634–8652. Retrieved from http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html

[88] Alejo José G. Sison, Marco Tulio Daza, Roberto Gozalo-Brizuela, and Eduardo C. Garrido-Merchán. 2024. ChatGPT: More than a "weapon of mass deception" ethical challenges and responses from the human-centered artificial intelligence (HCAI) perspective. *International Journal of Human–Computer Interaction* 40, 17 (2024), 4853–4872. DOI: https://doi.org/10.1080/10447318.2023.2225931

[89] Lintang Sutawika, Hailey Schoelkopf, Leo Gao, Baber Abbasi, Stella Biderman, Jonathan Tow, Ben Fattori, Charles Lovering, Jason Phang, Anish Thite, et al. 2024. EleutherAI/lm-evaluation-harness: V 0.4.7. DOI: https://doi.org/10.5281/zenodo.14506035

[90] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. arXiv:2307.09288. Retrieved from https://arxiv.org/abs/2307.09288

[91] Igor Tufanov, Karen Hambardzumyan, Javier Ferrando, and Elena Voita. 2024. LM transparency tool: Interactive tool for analyzing transformer language models. arXiv:2404.07004. Retrieved from https://arxiv.org/abs/2404.07004

[92] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*. Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.), ACM, 5998–6008. Retrieved from https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[93] Irene Weber. 2024. Large language models as software components: A taxonomy for LLM-integrated applications. arXiv:2406.10300. Retrieved from https://arxiv.org/abs/2406.10300

[94] Laura Weidinger, John Mellor, Maribeth Rauh, Conor Griffin, Jonathan Uesato, Po-Sen Huang, Myra Cheng, Mia Glaese, Borja Balle, Atoosa Kasirzadeh, et al. 2021. Ethical and social risks of harm from language models. arXiv:2112.04359. Retrieved from https://arxiv.org/abs/2112.04359

[95] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP '20)*. Qun Liu and David Schlangen (Eds.), Association for Computational Linguistics, 38–45. DOI: https://doi.org/10.18653/V1/2020.EMNLP-DEMOS.6

[96] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2023. AutoGen: Enabling next-Gen LLM applications via multi-agent conversation framework. arXiv:2308.08155. Retrieved from https://arxiv.org/abs/2308.08155

[97] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, and Mengnan Du. 2024. Explainability for large language models: A survey. *ACM Transactions on Intelligent Systems and Technology* 15, 2, Article 20 (2024), 1–38. DOI: https://doi.org/10.1145/3639372

Received 1 October 2025; revised 26 May 2026; accepted 13 July 2026