Taming Large Language Models via Scripted Interactions

VINCENZO SCOTTI, Karlsruhe Institute of Technology, Germany
JAN KEIM, Karlsruhe Institute of Technology, Germany
TOBIAS HEY, Karlsruhe Institute of Technology, Germany
ANNE KOZIOLEK, Karlsruhe Institute of Technology, Germany
RAFFAELA MIRANDOLA, Karlsruhe Institute of Technology, Germany

We are currently experiencing a bloom of Artificial Intelligence (AI)-powered software thanks to the development and advancements of Large Language Models (LLMs). Despite the applications of these LLMs being impressive and countless, these tools alone are unreliable. In fact, the possibility of LLMs generating faulty or hallucinated content makes them unsuitable for automating workflows and pipelines. In this context, Software Engineering (SE) comes in handy, offering a wide variety of formal tools to specify, verify and validate software behaviour. These SE tool can be used to describe constraints over an LLM output and, thus, provide better guarantees on the generated content. In this paper, we argue how the development of a Domain Specific Language (DSL) for scripting interactions LLMs, namely LLM Scripting Language (LSL), could be the key to improve AI-based applications. The idea in the long term is to make the interaction with LLMs programmable, abstracting from their implementation and training procedure. Nowadays, LLMs and LLM-based software are still lacking in terms of reliability, robustness, and trustworthiness. With LSL, we aim at tackling these deficiencies by looking into approaches to control the output of LLMs, to impose a structure to the interaction and to combine these ideas with verification, validation, and explainability.

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

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

ACM Reference Format:
Vincenzo Scotti, Jan Keim, Tobias Hey, Anne Koziolek, and Raffaela Mirandola. 2025. Taming Large Language Models via Scripted Interactions. In *Future Software Engineering Workshop 2025 (FSEws2025).* 12 pages. https://doi.org/10.5445/IR/1000181619

1 Introduction

Large Language Models (LLMs) [34] are enabling Artificial Intelligence (AI)-powered applications, reshaping software development, via coding assistance [21], and augmenting the software components, via natural language interfaces and supportive collaboration [51]. These LLMs are probabilistic generative models of text built on top of *Transformer* Deep Neural Networks (DNNs) [50] and owe their versatility to the expressive power of human (natural) language and the large amount of data they were trained on [11, 24, 31]. The combination of the two gave LLMs the *language understanding* and the *world knowledge* necessary to make them look so flexible and so easily usable in everyday tasks, leading to this "gold rush" of having the LLM or AI badge on software products [23]. Nevertheless, these tools are prone

Authors' Contact Information: Vincenzo Scotti, vincenzo.scotti@kit.edu, Karlsruhe Institute of Technology, Karlsruhe, Germany; Jan Keim, jan.keim@kit.edu, Karlsruhe Institute of Technology, Karlsruhe, Germany; Tobias Hey, hey@kit.edu, Karlsruhe Institute of Technology, Karlsruhe, Germany; Anne Koziolek, koziolek@kit.edu, Karlsruhe Institute of Technology, Karlsruhe, Germany; Raffaela Mirandola, raffaela.mirandola@kit.edu, Karlsruhe Institute of Technology, Karlsruhe, Germany.

to faulty behaviours, like that of *hallucinations* [26, 29], which harms the *reliability*, *robustness*, and *trustworthiness* of an application built on top of these models. In fact, despite their wide range of capabilities, LLMs cannot provide guarantees on the generated output. This includes cases where users are interested in guarantees that the model will stick to a certain structure (e.g., a JSON schema) or more complex guarantees on *factuality* (i.e., correctness with respect to world knowledge) [26], and *faithfulness* (i.e., correctness with respect to the current context, which contains the instructions) [26]. Using techniques like *few-shot learning*, Retrieval Augmented Generation (RAG), or Chain-of-Thought (CoT) help improve performance and reduce errors, but still they don't give any guarantee on the content generated by the LLM; the same applies when breaking down the problem into subtasks with the help of a domain expert [18, 45]. Even *fine-tuning* the LLM has the same issues and, at the same time, introduces also the need for labeled data and the possibility of running into *catastrophic forgetting* (i.e., the disruption of prior knowledge gained from the pre-training) [25, 33]. The lack of these guarantees makes these models unsuitable for software pipelines and workflows where we would actually like to employ autonomous AI components.

Instead of looking at the problem in an AI for Software Engineering (SE) perspective, where we want to integrate LLM-based components into software, we could turn it the other way around into a SE for AI challenge, where we can try to tackle issues concerning LLMs' reliability, robustness, and trustworthiness using SE [37]. In fact, SE offers formal tools to (i) specify, (ii) verify, and (iii) validate software behaviour, which we could use to provide guarantees on LLMs generated output and behaviour. With this paper, we take the chance to propose our view towards these challenges and we suggest resorting to a combination of templating and generative grammars as part of a Domain Specific Language (DSL) designed to script the interaction with the LLM –namely LLM Scripting Language (LSL)–, enforcing control on its input and output.

The problem of making these models more autonomous by enforcing their output or behaviour to respond to some requirements is already gaining attention. In fact, more advanced LLMs like *GPT* [28, 36] or *Gemini* [10, 42] already offer *structured output* functionalities via their APIs [7, 8] to generate JSON compatible with a given schema. There also exist more sophisticated approaches applying *generative grammars* to steer the generated content towards desired structures, implemented in frameworks like *Llama CPP* [3]. These formal grammar-based approaches are helpful when the LLM needs to adapt to an unseen or rare format whose description in the task instructions is not sufficient.

Ideally, LSL and its interpreter should be used to: (i) interact seamlessly with any underlying LLM; (ii) manage, automatically or based on developer's specifications, the context of the LLM depending on the ongoing task; (iii) impose desired format to the LLM input; (iv) impose target structure to the content generated by the LLM via constraints on the decoding process; (v) checking the consistency of the scripted process; (vi) offer automatic or semi-automatic error-handling procedures specific to the domain of the application. This idea may seem to point back towards frame and slot-filling architectures of old virtual assistants [30]. However, we also consider that (i) current technology allows for better language understanding and higher flexibility and (ii) introducing of formal structures allows to provide guarantees to the overall system.

2 Background

LLMs are DNNs trained on massive amounts of data to learn a probabilistic generative model of text. The underlying DNN of a LLM is a Transformer [50], a DNNs designed to process sequential data, like sequences of text tokens (the basic units processed by a LLM: words, sub-words, or characters), through the *self-attention mechanisms* [50]. This Transformer architecture enables LLMs to capture dependencies throughout the input text and memorise a large amount of information, allowing the generation of highly coherent and contextually relevant text.

Besides commercial, closed source models, there are ppen-access LLMs. They 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* [12, 44, 46] 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: (1) *system message* (main task instructions, which can be also chatbot directives); (2) *user message* (user's input to be processed, question, or request); (3) *agent response* (agent's output or response to user's question or request). Each sample in the data set 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 as coding assistants, like in the case of *Copilot* [21] or *Code Llama* [43], 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 specific task like classification (often modifying the architecture) [17, 40], nowadays prompting (i.e., instructing the model via natural language) is the preferred to solve most problems with LLM and fine-tuning is applied very rarely and preserving the generative nature of the LLM.

Capabilities. Most LLMs are auto-regressive (casual), i.e., they are designed to generate text one word after another. Users prompt the LLMs 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 are: (i) *few-shot learning* (sometimes called in-context learning) [11], (ii) *Retrieval Augmented Generation* (RAG) [20], and (iii) *Chain of Thought* (CoT) *reasoning* [32]. Few-shot learning 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 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 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.

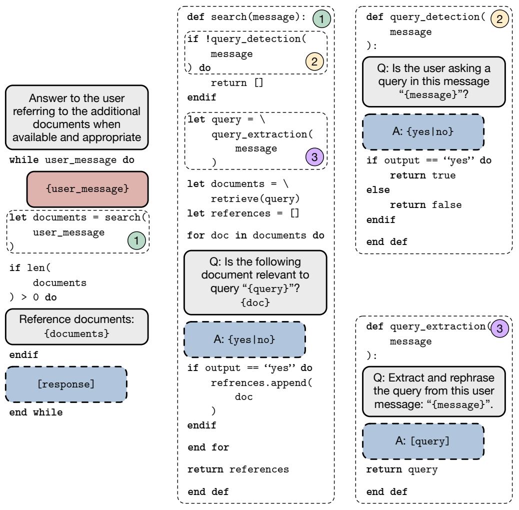
Limitations. LLMs come with very general capabilities and high flexibility, but they are prone to many known issues. These issues range from architectural limitations to problems with the generated content.

From the DNN architecture side, we identify two main issues: (1) context size (i.e., the maximum number of consecutive tokens a model can manage) and (2) computational complexity. Even though in the last few years we have seen an increase of 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 [15, 16, 39] 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 data.

These technical issues lead the models to rely on more recent context, ending up generating contradictions, struggling with multi-step reasoning, or 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 (factuality problems [29]) nor in the current context (faithfulness problems [29]).

The combination of these issues makes these models unsuitable for tasks that require high reliability, robustness, and trustworthiness, like automating some pipeline using the LLM. Consider the case where we are interested in interaction with a user and generate the 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 latest news) and CoT reasoning may help improving 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 LLM Scripting Language



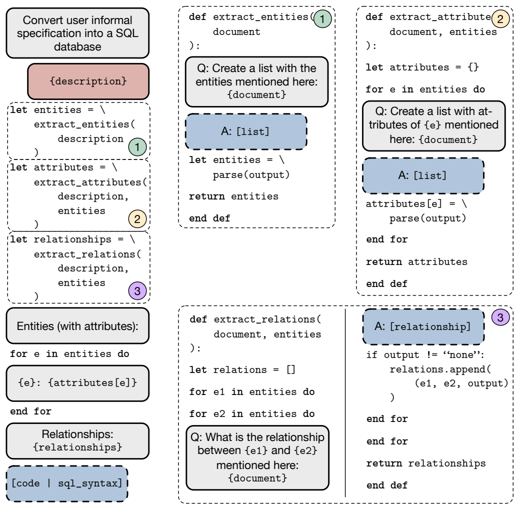
```
Answer to the user referring to the additional documents when available and appropriate
while user_message do
    {user_message}
let documents = search(
    user_message                                ①
)
if len(
    documents
) > 0 do
Reference documents: {documents}
endif
[response]
end while
```

```
def search(message):                            ①
if !query_detection(
    message                                     ②
) do
    return []
endif
let query = \
    query_extraction(
        message                                 ③
    )
let documents = \
    retrieve(query)
let references = []
for doc in documents do
Q: Is the following document relevant to query "{query}"? {doc}
A: {yes|no}
if output == "yes" do
    refrences.append(
        doc
    )
endif
end for
return references
end def
```

```
def query_detection(                            ②
    message
):
Q: Is the user asking a query in this message "{message}"?
A: {yes|no}
if output == "yes" do
    return true
else
    return false
endif
end def
```

```
def query_extraction(                           ③
    message
):
Q: Extract and rephrase the query from this user message: "{message}".
A: [query]
return query
end def
```

(a) Semi-structured use case: knowledge grounded chat.



```
Convert user informal specification into a SQL database
    {description}
let entities = \
    extract_entities(
        description                             ①
    )
let attributes = \
    extract_attributes(
        description,
        entities                                ②
    )
let relationships = \
    extract_relations(
        description,
        entities                                ③
    )
Entities (with attributes):
for e in entities do
{e}: {attributes[e]}
end for
Relationships: {relationships}
[code | sql_syntax]
```

```
def extract_entities(                           ①
    document
):
Q: Create a list with the entities mentioned here: {document}
A: [list]
let entities = \
    parse(output)
return entities
end def
```

```
def extract_attribute(                          ②
    document, entities
):
let attributes = {}
for e in entities do
Q: Create a list with attributes of {e} mentioned here: {document}
A: [list]
attributes[e] = \
    parse(output)
end for
return attributes
end def
```

```
def extract_relations(                          ③
    document, entities
):
let relations = []
for e1 in entities do
for e2 in entities do
Q: What is the relationship between {e1} and {e2} mentioned here: {document}
A: [relationship]
if output != "none":
    relations.append(
        (e1, e2, output)
    )
end for
end for
return relationships
end def
```

(b) Structured use case: database creation

Fig. 1. Example use cases (system messages, instructions and subroutine outputs).

We propose LSL: a DSL to format and constrain the input and output of an LLM. The scripting language and its interpreter should improve the guarantees on many aspects concerning the use of LLMs and its autonomy. The interpreter running the script should interact seamlessly with the LLM, locally or remotely, and to the user.

The intended uses of the scripting language are (i) defining interaction blocks, (ii) managing the context processed by the LLM, (iii) formatting input prompt to the LLM; (iv) constraining 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 and end user, or other services, can benefit from these AI-based applications using these pipelines. Given that LLMs can generate any sequence, as their vocabulary covers everything that can be written with *Unicode*, scripts and applications can be agnostic of them.

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 Figure 1. Thinking about the semi-structured use case of the knowledge-grounded chat represented in Figure 1a, we can 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 taking 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 creating from informal specifications of Figure 1b. This second use case shows 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.

When we mention "constraining the output", we mean limiting the output to a subset of possible strings whether they are enumerated or specified with some formalism. Instead, when we mention "controlling the interaction" we mean properly adding or removing pieces of the context whenever prompting the LLM, as we further outline in Section 3. In Figure 1, 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 both Figure 1a and Figure 1b, or to stick to some specific syntax, as displayed in Figure 1b. 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 1a, or when extracting entities and their relationships, like in Figure 1b.

Beyond prompting. Engineering the prompt to trigger the correct model completion is a non-trivial task [13, 44]. Besides phrasing correctly the request to the LLM, there are other issues. One is making sure to provide all necessary information (e.g., task description, target labels, output format). Another concerns using correct techniques like few-shot learning to increase the probability of triggering the desired model's behaviour, 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 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 1b or generating JSON strings that respond to a given JSON schema. As we discuss later in Sections 4 and 5, 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 maximises 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, 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 can be combined with that of prompt engineering techniques. Additionally, LSL and its interpreter can be designed to help the prompt engineering phase, 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.

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 one to the other. However, the stateless nature of LLMs makes them compatible with non-linear interaction, whether they are chats or other more structured processes [18, 38]. This means that we can integrate an LLM in a workflow characterised by (i) multiple, possibly independent, steps to solve the main task or its sub-tasks, (ii) isolated steps that do not require to process 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 1b shows how both a semi-structured task such as the knowledge-grounded chat of the example in Figure 1a and a structured task like the generation of SQL code to build a database of the example in Figure 1b 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 [41], but (i) they lack expressivity to deal with complex problems, (ii) they do not take into account possible optimisations, (iii) they are bound to one or a small set of domains, (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.

Some independent passages, like the relevant document selection in the search function in Figure 1a, or the attributes and relations extractions in Figure 1b, can be parallelised. A proper integration of this feature in LSL can allow to automatically exploit 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 utilisation of the computational resources. Furthermore, we also consider error management by integrating 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 [47].

4 Opportunities

Reliability. Concerning the reliability, which we define as the ability of the LLM to "perform as required, without failure" [9], 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 writing some code). Often, the ability to interact with the environment is implemented with *function calling*: the LLM generates a response matching a pattern that triggers the call according to information provided in the context.

Classification problems require that the LLM generates 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 that the LLM introduces errors stays, we can ensure that everything the LLM generates will comply with the required input to the following steps by constraining the generated completion to follow specific patterns. This can 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 a pipeline and makes it more reliable.

Robustness. LLMs are designed to be robust, where robustness is intended as the ability to avoid spurious correlations and patterns [11, 22]. Nevertheless, these models are error-prone: ambiguous or ill-formulated inputs can make them fail. The same applies to use cases with multi-step reasoning or those requiring very recent information.

A measure to improve robustness in these cases is to automate the use of the techniques we presented in Section 2. For example, given a known classification task for which there are available data sets, 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 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 [47]. 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.

Trustworthiness. Trustworthiness in AI-based applications come from two main sources: verification and validation, and explainability. 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. Explainability, instead, poses an important requirement to make AI-based application more *transparent* [27].

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 analysis via formal tools. Using tools for verification and tools for validation allows us to add guarantees on the constraints imposed to the LLM and on the outcome of the scripts realised with LSL [9]. For instance, one can verify that the interaction scripted with LSL always terminates or that there are no unreachable or dead nodes in the computations 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 insights on the model's accuracy on a specific task or sub-task and apply probabilistic model checking to measure the likelihood of certain behaviours or outcomes. 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.

Besides the transparency, which helps users understand the behaviour of the application, explainability can be also a debugging tool by highlighting which parts of prompt caused some faulty output and allowing to correct such parts, for example by rephrasing some 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 using existing tools and approaches [54]. The latter solution, more specific to LSL, is to use the structure in the interaction imposed by the script to trace back the outcome.

5 Challenges

Implementation. Developing a scripting language like LSL is complex, encompassing multiple SE areas including language and APIs design, concurrency and performance [35]. The syntax should allow the seamless integration of script code, input templating, and output constraining, easing the management of context and the interaction with the model. Particularly, output constraints, which are critical to reliability, should be analysed carefully to allow specifying both simple constraints, like alternative completions, to more complex constraints, like the syntax of a programming language.

About concurrency and performance, given the computational demand of running an LLM, being able to automatically parallelise the passages in the pipeline, packing together in batches those requests that can be run concurrently, is essential. The concurrency aspects span to concurrent processes characterised by different sessions and contexts, requiring to manage carefully which contexts are forwarded to the LLM and when to ensure optimal resource utilisation. Another aspect to take into consideration is the causal nature of LLMs, which makes it possible to exploit the caching mechanism.

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 run-time poses a non-trivial task, even if we want to resort to external tools.

Standardising the interfaces is crucial for several reasons, from *interoperability* of different modules to *consistency and predictability* of the behaviours and to *ease of integration*. In this context, we are interested in standardising two main interfaces: (1) the interface to the LLM, obviously, and (2) the interface to external resources, like data sets or documents and knowledge bases. The APIs should be designed to expose functionalities like likelihood prediction, generation, embedding and to access other resources. The most adopted interface to LLMs is the one in the *Transformers* library [53]. Yet, there are other valid alternatives to self-host models, like llama.cpp [4], or to access them remotely via commercial APIs, like *watsonx.ai* [5]. Moreover, larger commercial models have their own separate interfaces [1, 6]. Benchmark frameworks like *Big Bench* and *Language Model Evaluation Harness* [49] proposed wrappers to unify the interface to LLMs, yet not all APIs or inference servers support the necessary functionalities. Interfaces to external resources are essential to automate many tasks, like the retrieval of samples for few-shot learning. Having standard interfaces to document on knowledge bases can help with (i) making content generated by LLMs more factual and evidence-based and (ii) using LLMs for semantic ranking and re-ranking in search. Given standardised templates for the tasks that LLMs can be applied to, this can enable the automation of hyper-parameters tuning and the computation of performances for later verification and validation.

Inherent threats. Throughout this paper, we pointed out that LSL cannot remove all threats and risks related to the use of LLMs [9]. Nevertheless, some issues can be removed (e.g., enforcing an output template) and the likelihood of others occurring can be reduced (e.g., using few-shot learning). Identifying and quantifying all the threats that cannot be neutralised is a significant challenge as they depend on multiple factors, including the specific LLM being used and the specific (sub-)task in the pipelines. Undoubtedly, the problem of hallucinations, either in terms of factuality or in terms of faithfulness poses the biggest risk. The possibility of the LLM for 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 [26, 29].

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 [52].

Technologies aimed at automating processes using LLMs often introduce problems related to *job replacement* (i.e., when software takes care of human tasks) and *deskilling* (i.e., when humans lose their expertise on some tasks by stopping doing it) [48]. The intention of LSL is for sure to automate processes, but by promoting the collaboration between LLMs, and human experts. Moreover, human feedback and supervision are always crucial to ensure the proper functioning of the system.

Additionally, content generated by LLMs exposes to risks related to *bias* and *toxicity* [19], besides *misinformation* and *hallucination.* These issues pose a threat towards the user 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 [14] and measure quantitatively this predisposition and (ii) to recognise this kind of harmful content, preventing sharing it with the user. These tools should be integrated in the LSL ecosystem to quantify and mitigate these risks.

6 Conclusion

LLMs represent valuable assets to build AI-powered applications. Nevertheless, they come with several risks and limitations that prevent to freely adopt them into fully automated pipelines and workflows. With this paper, we presented our vision of what advances SE tools and frameworks can bring to AI in terms of (i) reliability, (ii) robustness, and (iii) trustworthiness of LLMs. The key idea is to develop a DSL working 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 behaviour. An interpreter complements LSL managing the LLM context, the interfaces towards the model and the external resources, and error correction subroutines as well as as running consistency checks and producing explanations.

Summarising, we plan to constrain LLM outputs to address reliability by automatically exploiting their capabilities (namely, few-shot learning, RAG, and CoT reasoning) and by introducing error handling loops. We address LLM robustness through verification and validation of script code, and through the generation of explanations, we address trustworthiness. In terms of challenges throughout the development, we expect to deal with the technical aspects immediately related to the design of the language and the implementation of the interpreter, with the lack of common standards and interfaces towards models and resources, with the difficulty in quantifying and identifying sources of risks commonly associated to the use of LLMs (errors and hallucinations) and with the ethical concerns related as well to the use of LLMs.

Acknowledgments

This work was supported by funding from the pilot program Core Informatics at KIT (KiKIT) of the Helmholtz Association (HGF) and supported by the German Research Foundation (DFG) - SFB 1608 - 501798263 and KASTEL Security Research Labs, Karlsruhe.

References

[1] 2020. https://platform.openai.com/docs/api-reference/

[2] 2022. https://python.langchain.com/docs/introduction/

[3] 2023. https://github.com/ggerganov/llama.cpp/blob/master/grammars

[4] 2023. https://github.com/ggerganov/llama.cpp

[5] 2023. https://www.ibm.com/products/watsonx-ai

[6] 2023. https://ai.google.dev/

[7] 2024. https://openai.com/index/introducing-structured-outputs-in-the-api

[8] 2024. https://ai.google.dev/gemini-api/docs/structured-output

[9] NIST AI. 2023. Artificial Intelligence Risk Management Framework (AI RMF 1.0). https://doi.org/10.6028/NIST.AI.100-1

[10] Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, et al. 2023. Gemini: A Family of Highly Capable Multimodal Models. *CoRR* abs/2312.11805 (2023). https://doi.org/10.48550/ARXIV.2312.11805 arXiv:2312.11805

[11] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, 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 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[12] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, et al. 2022. Scaling Instruction-Finetuned Language Models. *CoRR* abs/2210.11416 (2022). https://doi.org/10.48550/ARXIV.2210.11416 arXiv:2210.11416

[13] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, et al. 2024. Scaling Instruction-Finetuned Language Models. *J. Mach. Learn. Res.* 25 (2024), 70:1–70:53. https://jmlr.org/papers/v25/23-0870.html

[14] Simone Corbo, Luca Bancale, Valeria De Gennaro, Livia Lestingi, Vincenzo Scotti, et al. 2025. How Toxic Can You Get? Search-based Toxicity Testing for Large Language Models. *CoRR* abs/2501.01741 (2025). https://doi.org/10.48550/ARXIV.2501.01741 arXiv:2501.01741

[15] Tri Dao. 2024. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. https://openreview.net/forum?id=mZn2Xyh9Ec

[16] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, et al. (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/67d57c32e20fd0a7a302cb81d36e40d5-Abstract-Conference.html

[17] 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 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[18] 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 2024)*, Mourad Abbas and Abed Alhakim Freihat (Eds.). Association for Computational Linguistics, Trento, 456–466. https://aclanthology.org/2024.icnlsp-1.47/

[19] Isabel O. Gallegos, Ryan A. Rossi, Joe Barrow, Md Mehrab Tanjim, Sungchul Kim, et al. 2024. Bias and Fairness in Large Language Models: A Survey. *Computational Linguistics* 50, 3 (09 2024), 1097–1179. https://doi.org/10.1162/coli_a_00524 arXiv:https://direct.mit.edu/coli/article-pdf/50/3/1097/2471010/coli_a_00524.pdf

[20] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, et al. 2023. Retrieval-Augmented Generation for Large Language Models: A Survey. *CoRR* abs/2312.10997 (2023). https://doi.org/10.48550/ARXIV.2312.10997 arXiv:2312.10997

[21] GitHub. 2021. Introducing GitHub Copilot: your AI pair programmer. https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/

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

[23] Shane Greenstein. 2023. The AI Gold Rush. *IEEE Micro* 43, 6 (2023), 126–128. https://doi.org/10.1109/MM.2023.3322049

[24] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, et al. 2022. Training Compute-Optimal Large Language Models. *CoRR* abs/2203.15556 (2022). https://doi.org/10.48550/ARXIV.2203.15556 arXiv:2203.15556

[25] Jianheng Huang, Leyang Cui, Ante Wang, Chengyi Yang, Xinting Liao, et al. 2024. Mitigating Catastrophic Forgetting in Large Language Models with Self-Synthesized Rehearsal. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*,

ACL 2024, Bangkok, Thailand, August 11-16, 2024, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 1416–1428. https://doi.org/10.18653/V1/2024.ACL-LONG.77

[26] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, et al. 2023. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *CoRR* abs/2311.05232 (2023). https://doi.org/10.48550/ARXIV.2311.05232 arXiv:2311.05232

[27] Yue Huang, Lichao Sun, Haoran Wang, Siyuan Wu, Qihui Zhang, et al. 2024. Position: TrustLLM: Trustworthiness in Large Language Models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. https://openreview.net/forum?id=bWUU0LwwMp

[28] Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, et al. 2024. GPT-4o System Card. *CoRR* abs/2410.21276 (2024). https://doi.org/10.48550/ARXIV.2410.21276 arXiv:2410.21276

[29] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, et al. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12 (2023), 248:1–248:38. https://doi.org/10.1145/3571730

[30] Dan Jurafsky and James H. Martin. 2009. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition, 2nd Edition*. Prentice Hall, Pearson Education International. https://www.worldcat.org/oclc/315913020

[31] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, et al. 2020. Scaling Laws for Neural Language Models. *CoRR* abs/2001.08361 (2020). arXiv:2001.08361 https://arxiv.org/abs/2001.08361

[32] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and 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 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, et al. (Eds.). http://papers.nips.cc/paper_files/paper/2022/hash/8bb0d291acd4acf06ef112099c16f326-Abstract-Conference.html

[33] Dingcheng Li, Zheng Chen, Eunah Cho, Jie Hao, Xiaohu Liu, et al. 2022. Overcoming Catastrophic Forgetting During Domain Adaptation of Seq2seq Language Generation. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL 2022, Seattle, WA, United States, July 10-15, 2022*, Marine Carpuat, Marie-Catherine de Marneffe, and Iván Vladimir Meza Ruíz (Eds.). Association for Computational Linguistics, 5441–5454. https://doi.org/10.18653/V1/2022.NAACL-MAIN.398

[34] Shervin Minaee, Tomás Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, et al. 2024. Large Language Models: A Survey. *CoRR* abs/2402.06196 (2024). https://doi.org/10.48550/ARXIV.2402.06196 arXiv:2402.06196

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

[36] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/ARXIV.2303.08774 arXiv:2303.08774

[37] Mauro Pezzè, Matteo Ciniselli, Luca Di Grazia, Niccolò Puccinelli, and Ketai Qiu. 2024. The Trailer of the ACM 2030 Roadmap for Software Engineering. *ACM SIGSOFT Softw. Eng. Notes* 49, 4 (2024), 31–40. https://doi.org/10.1145/3696117.3696126

[38] 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 2024, Arenzano, Genoa, Italy, June 3-7, 2024*, Cristina Conati, Gualtiero Volpe, and Ilaria Torre (Eds.). ACM, 87:1–87:3. https://doi.org/10.1145/3656650.3656739

[39] Markus N. Rabe and Charles Staats. 2021. Self-attention Does Not Need O(n^2) Memory. *CoRR* abs/2112.05682 (2021). arXiv:2112.05682 https://arxiv.org/abs/2112.05682

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

[41] 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, EMNLP 2023 - System Demonstrations, Singapore, December 6-10, 2023*, Yansong Feng and Els Lefever (Eds.). Association for Computational Linguistics, 431–445. https://doi.org/10.18653/V1/2023.EMNLP-DEMO.40

[42] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *CoRR* abs/2403.05530 (2024). https://doi.org/10.48550/ARXIV.2403.05530 arXiv:2403.05530

[43] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, et al. 2023. Code Llama: Open Foundation Models for Code. *CoRR* abs/2308.12950 (2023). https://doi.org/10.48550/ARXIV.2308.12950 arXiv:2308.12950

[44] Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, et al. 2022. Multitask Prompted Training Enables Zero-Shot Task Generalization. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net. https://openreview.net/forum?id=9Vrb9D0WI4

[45] Vincenzo Scotti and Mark James Carman. 2024. LLM Support for Real-Time Technical Assistance. In *Machine Learning and Knowledge Discovery in Databases. Research Track and Demo Track - European Conference, ECML PKDD 2024, Vilnius, Lithuania, September 9-13, 2024, Proceedings, Part VIII (Lecture Notes in Computer Science)*, Albert Bifet, Povilas Daniusis, Jesse Davis, Tomas Krilavicius, Meelis Kull, et al. (Eds.), Vol. 14948. Springer, 388–393. https://doi.org/10.1007/978-3-031-70371-3_26

[46] Vincenzo Scotti, Licia Sbattella, and Roberto Tedesco. 2024. A Primer on Seq2Seq Models for Generative Chatbots. *ACM Comput. Surv.* 56, 3 (2024), 75:1–75:58. https://doi.org/10.1145/3604281

[47] 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 2023, New Orleans, LA, USA, December 10 - 16, 2023*, Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, et al. (Eds.).

http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html

[48] 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. *Int. J. Hum. Comput. Interact.* 40, 17 (2024), 4853–4872. https://doi.org/10.1080/10447318.2023.2225931

[49] Lintang Sutawika, Hailey Schoelkopf, Leo Gao, Baber Abbasi, Stella Biderman, et al. 2024. EleutherAI/lm-evaluation-harness: v0.4.7. https://doi.org/10.5281/zenodo.14506035

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, et al. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, et al. (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[51] Irene Weber. 2024. Large Language Models as Software Components: A Taxonomy for LLM-Integrated Applications. *CoRR* abs/2406.10300 (2024). https://doi.org/10.48550/ARXIV.2406.10300 arXiv:2406.10300

[52] Laura Weidinger, John Mellor, Maribeth Rauh, Conor Griffin, Jonathan Uesato, et al. 2021. Ethical and social risks of harm from Language Models. *CoRR* abs/2112.04359 (2021). arXiv:2112.04359 https://arxiv.org/abs/2112.04359

[53] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, 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 2020 - Demos, Online, November 16-20, 2020*, Qun Liu and David Schlangen (Eds.). Association for Computational Linguistics, 38–45. https://doi.org/10.18653/V1/2020.EMNLP-DEMOS.6

[54] Haiyan Zhao, Hanjie Chen, Fan Yang, Ninghao Liu, Huiqi Deng, et al. 2024. Explainability for Large Language Models: A Survey. *ACM Trans. Intell. Syst. Technol.* 15, 2 (2024), 20:1–20:38. https://doi.org/10.1145/3639372