Toward Bridging the Tool Gap: Equipping Large Language Models with Tools to Answer Programmers' Questions

Lukas Böhme
lukas.boehme@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Christoph Thiede
christoph.thiede@student.hpi.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Toni Mattis
toni.mattis@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Tom Beckmann
tom.beckmann@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Jens Lincke
jens.linke@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Robert Hirschfeld
hirschfeld@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Abstract

Programmers ask complex questions in their search for solutions during software development. Along with traditional tools such as debuggers and profilers, state-of-the-art approaches like Babylonian Programming can help programmers answer those questions through interactive and visual feedback. Large language models (LLMs) and programming agents are part of programmers' toolboxes and are well-integrated into their development workflows. However, they are not yet helpful in considering questions involving run-time behavior.

In this paper, we first review the literature to identify concerns programmers face during development and highlight how humans usually address them. We then focus on questions about program behavior and propose integrating Babylonian-style programming techniques with LLMs to help answer related questions. Finally, we suggest four key properties that future LLM-based development tools should support: (1) LLM tool usage traceability for explainability, (2) resumability of development progress for handovers between human programmers and LLM-based programming agents, (3) context efficiency through selective data querying, and (4) multi-source synthesis for tool integration.

CCS Concepts: • **Software and its engineering → Development frameworks and environments**; *Software development techniques*; Integrated and visual development environments; • **Human-centered computing → Interactive systems and tools**; • **Computing methodologies →** *Natural language processing*.

Keywords: large language models, tool-augmented LLMs, programming assistants, developer questions

ACM Reference Format:
Lukas Böhme, Christoph Thiede, Toni Mattis, Tom Beckmann, Jens Lincke, and Robert Hirschfeld. 2025. Toward Bridging the Tool Gap: Equipping Large Language Models with Tools to Answer Programmers' Questions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3759534.3762682

PAINT '25, Singapore, Singapore

ACM ISBN 979-8-4007-2160-1/25/10
https://doi.org/10.1145/3759534.3762682

1 Introduction

During programming, programmers formulate and ask questions [6, 13–16, 23]. Questions during the development process span a wide range, from information-seeking questions, such as "Which class do I need to solve this task?" to detailed code comprehension questions about the run-time: is this line reachable under those constraints. Answering such questions in a timely manner increases programmer productivity [14].

Large language models (LLMs) are frequently used during programming to answer programmers' questions and they have become an integral part of their workflow [12]. LLMs show similar biases as humans regarding solution-planning steps since they are trained on a vast amount of text predominantly created by humans [18]. However, unlike human programmers, LLMs only have limited access to the tools and interfaces typically used to investigate and find answers to their questions. This limitation is particularly evident for questions related to run-time behavior, which often require dynamic analysis, and real-time feedback from code. Recent

studies emphasize this gap: state-of-the-art models without tool support perform poorly on basic code reasoning questions [2, 8]. This divergence raises a critical point: *if programmers use LLMs to answer questions about code, but LLMs can't explore or interact with code the way humans can, how well can they answer those questions?*

Humans have access to debugging tools (e.g., breakpoints, logs, watches) optimized for manual inspection and stepwise execution to develop an iterative understanding of program behavior [11]. Moreover, Live [25] and Babylonian Programming [20] in combination with example generation [17] allow direct inspection of run-time values to understand what the code actually does. Those interactive, visual tools are built to help humans understand code through direct interaction, not to automate understanding for machines. How LLMs can acquire the same information to be on par with human capabilities is still underexplored. By equipping LLMs with capabilities available to human programmers—such as debugging and run-time inspection—we envision a future in which LLMs can reliably answer programmers' most pressing questions, shortening the feedback loop during development.

In this paper, we formulate a vision and show an initial prototype for how we can bridge the gap between the tools human programmers use and LLMs. To that end, we first explain how LLMs can invoke external tooling in Section 2.1. Second, we list and categorize questions that human programmers ask and discuss tools they use to answer the questions in Section 2.2. We examine the tools that human programmers use to answer these questions, arguing that LLMs should likewise have access to such tools to effectively assist developers. While each concern represents an important gap in current LLM capabilities, we focus on run-time behavior and debugging questions for the remainder of this paper, as they exemplify the challenges of adapting interactive, visual tools for text-based LLMs. In Section 3, we take one tool that allows introspection into run-time information, Babylonian Programming [20], and demonstrate an approach to transfer its benefits to an LLM agent. In Section 4, we then envision a scenario where this tool's integration allows the LLM to answer questions that would have previously been difficult. Finally, after highlighting related work in Section 5, we discuss and formulate aspects of a vision on how further tools should be integrated and how challenges can be addressed in Section 6.

2 Questions in Human and LLM Workflows

In this section, we first describe how LLMs can ask questions and receive answers from an environment. We then give an overview of questions that programmers ask and derive challenges to integrate these with the mechanism LLMs use to ask questions.

2.1 Questions and Feedback Cycles for LLMs

Large Language Models are neural networks trained on a vast amount of publicly available textual data to predict the next most likely sequence of characters. Research has shown that LLMs excel in pattern-based tasks such as code generation [3]. Prompting strategies such as Chain Of Thought prompting instruct the LLM to build an internal reasoning chain through a task to improve answer quality [28]. Leveraging the same reasoning capabilities, models can be used as zero-shot planners, creating step-by-step instructions [10]. Yet, LLMs operate on text and cannot use interactive graphical tooling designed for humans. This limits their ability to answer questions requiring information that the tooling provides. In addition, because LLMs generate probabilistic output, they are not suited for deterministic answers, such as mathematical problems. Consequently, researchers equip LLMs with external tools that can return an exact deterministic result. Toolformer equips LLMs with an API to call external tools, such as Q&A systems, search engines, and calculators [21]. In the context of software engineering, ToolCoder uses a fine-tuned model to call search APIs to improve code generation tasks [31]. An external tool consists of a name, a short description of what the tool does, and optional parameters. Closed-source models such as GPT-4o incorporate similar capabilities through a technique called "function calling"[1]. The textual output of the tool is then fed as input to the LLM.

2.2 Questions Asked by Programmers

Several studies investigate questions programmers ask during programming activities to characterize their information-seeking behaviors, identify gaps in existing documentation and tooling, and inform the design of developer assistance systems. To synthesize insights from these studies, we conducted an exploratory literature search using variations of the query 'questions developers ask' across academic databases. From the results, we pragmatically selected papers that examine questions during programming activity, excluding studies focused solely on learning or documentation scenarios. We extracted representative questions from these papers and grouped them into underlying concerns. We define a concern as a group of questions with similar intent and sources of information required to answer these questions. For each concern, we outline the motivation behind the question, the information sources required to answer it, and the developer tools that LLMs should likewise be able to use to be on par with human programmers. LLMs must be able to answer questions from these concerns to match human programmer capabilities.

Code & usage exploration. Understanding and navigating an unfamiliar codebase can be challenging, as it requires

[1] https://platform.openai.com/docs/guides/function-calling

developing a mental model of the existing components and their purposes. When programmers work on a given task in an existing codebase, they want to reuse existing functionality to avoid code duplication or time required for implementation [4]. Because of that, they ask information-seeking questions and follow up with code comprehension questions to understand how to use the discovered API elements [6]. Questions such as *"Which packages or namespaces of an API provide types relevant to my task?"*, *"Is there a PieChart type?"*, *"What does the Validator class actually do?"* are asked to build up that mental model [6]. Programmers explore unfamiliar codebases using full-text or vector-based search and navigation features enabled by static analysis, and read documentation including changelogs [5, 6]. Beyond searching within the codebase, programmers search the web for possible answers, e.g., for frameworks or basic functionality of a programming language. They also interactively run code, evaluate examples by trial and error, and inspect run-time values to understand types and data usage with, for example, Babylonian Programming [20].

Software architecture. Enhancing the structural organization of a codebase and leveraging proven design patterns contributes to better maintainability and extensibility, especially when unexpected requirements arise [19]. By fitting new functionality into a coherent architecture, programmers ease possible extension and maintain flexibility for unforeseen requirements. In particular, design patterns are proven implementation guidelines that localize changes, ensuring extensibility and maintainability. Due to such lasting benefits, programmers raise concerns by asking: *What's the best design for implementing this? Is the existing design a good one? To move this feature into this code, what else needs to be moved?* [16, 23]. While design pattern recommendation systems exist, they are not widely adopted by programmers. As a result, identifying and selecting an appropriate design pattern for a given task still requires expert knowledge and experience [1]. Similar codebases and documentation on their architectural decisions can serve as inspiration, though finding them often requires an extensive search.

Code history & ownership. Most software projects are collaborative efforts that require tooling to track who made changes, why they were made, and how those changes affect the team. When a task requires understanding unfamiliar code snippets, questions such as *"Who owns this piece of code?"*, *"Who has made changes to a defect?"*, *"How have resources I depend on changed?"* emerge [7, 13]. Colleagues responsible for the given code might have additional context about why it was written in a certain way. Knowledge about design or implementation choices are often not externalized, making it inaccessible to others. Programmers might uncover this information by reviewing pull request discussions, mailing lists, and commit messages of version control systems. In other cases, it may be inferred through manual inspection of code history; however, this process is often tedious and time-consuming.

Performance & resource management. Programmers are also concerned with the performance characteristics of their code. Meeting quality standards or ensuring performance guarantees requires programmers to consider execution speed, memory usage, and thread safety. Programmers must identify and analyze bottlenecks when a function does not meet performance requirements. On embedded hardware, limited memory makes efficient memory management essential. In safety-critical systems, thread-safety is vital to avoid hard-to-debug race conditions. Thus, questions such as *"Which part of this code takes the most time?"*, *"How big is this in memory?"* and *"Is this method thread-safe?"* arise [16]. To track performance bottlenecks, programmers use time or memory profilers, which aggregate stack samples and record metrics such as execution time and memory usage. Results of the time profiling are often visualized as a flame graph, showing the total time spent in each function. Memory profiling shows allocations, lifetimes, and potential memory leaks.

Testing. Testing uncovers errors or unexpected behavior within the code, ensuring code quality and correctness. Programmers write tests to ensure quality, uncover potential issues and edge cases, and to decrease maintenance expenses [24]. Out of this motivation, questions such as *"How can I test this code or functionality?"* and *"Is the test or code responsible for this test failure?"* arise. Despite manual test creation, programmers employ test generation frameworks to create tests automatically. They utilize coverage tooling to identify non-tested execution paths. In addition to traditional unit and integration testing, fuzzing and mutation testing are techniques to uncover unexpected behavior in code. Both techniques can produce a large volume of output that programmers have to analyze and interpret.

Build environment. Getting the project to run is essential for debugging and introspection, but requires knowledge about the prerequisites, project dependencies, and the build chain. Studies show that programmers compile 7–10 times per day, making a functional build chain a key part of productive workflows [22]. Programmers are at least once concerned about the build process for their setup, resulting in questions such as *"What do I need to include to build this?"* and *"Why did the build break?"* [16]. In the best case, step-by-step documentation, an automatic script ensures a straightforward build or a knowledgeable and available colleague. However, as soon as errors arise, the programmer needs to find a solution by interactive trial-and-error execution, debugging and reviewing the build script, inspecting mistakes and progress, or searching the web.

Run-time behavior & debugging. Running and debugging code is an interactive task used to explore, understand,

and potentially fix code in case it does not show the desired behavior. It is useful for analyzing unfamiliar or buggy code and identifying its potential applications by formulating hypotheses and testing them through trial and error. When running and inspecting code, programmers mainly ask questions related to control flow and state: *Is this line reachable under these constraints? Why did it (not) happen? How did this run-time state occur? Where was this variable last changed?* [14–16]. Static code analysis is often not sufficient to reason about how a program actually behaves at run-time. Therefore, programmers rely on interactive and visual tools to expose and track control and data flow. Modern debugging tools including time-travel debuggers [27], dynamic slicing techniques, Live and Babylonian Programming environments, tracing frameworks and low level logging statements provide the programmer with the required information to validate their hypothesis on how the code works step by step. To test the code with different invocations, example generation techniques can be applied [17].

2.3 Discussion of Questions and Tools

The tools programmers use to answer their questions include both tools that are commonly used as command-line tools, as well as tools that programmers typically use interactively. Interactive tools present a challenge for text-based LLMs. For example, they present users with an interaction-feedback loop that often relies on frequent small steps. The programmer is presented with information, triggers an interaction, receives feedback in the form of new information, and derives the next interaction.

Another aspect that presents challenges is the visual nature of tools. For example, flame graphs give a detailed view of where an execution spends how much time. These tools use visual cues to structure information and guide a programmer's attention to relevant parts. Text-based LLMs necessarily require textual representations of information. Converting visual data to text reduces accessibility since visual cues like color and proximity help programmers interpret information faster [20].

The effect on LLMs varies depending on the tool and use case. However, it is known that the structure of the input can significantly impact LLM output generation [9]. Consequently, while the best practices for presentation to LLMs differ from those designed for humans, investigating and experimenting with turning visual structures into text is important.

Moreover, to ensure collaboration between programmers and LLMs, textual input should be optimized for model performance and developers' interpretability. Programmers should be able to seamlessly continue the work initiated by the LLM, whether that means resuming exploration or a debugging session. This requires intermediate artifacts, such as instrumented code, probe results, or potential edits.

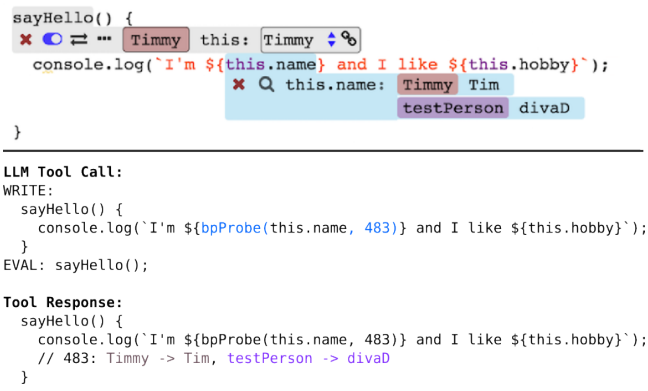

Figure 1. Top: A screenshot of a Babylonian Programming implementation [20]. An example is attached to the `sayHello` method. A probe is attached to the `this.name` expression and reports values observed during example execution. Bottom: an implementation of the mechanism from the LLM tool call perspective, where the LLM requests to write probes into the source and specifies the example it would like to invoke.

3 Babylonian Programming for LLMs

In this section, we describe a prototype of a visual tool made accessible to LLMs to serve as an example for similar tools. We choose for this example Babylonian Programming [20].

Babylonian Programming consists of examples and probes as shown in Figure 1. Unlike the program's main entry point, examples serve as alternative entry points that allow targeted execution of specific code sections. As such, they limit the scope of execution to those parts that the programmer deems relevant. Probes are used to sample information from the run-time state of a program. For example, a value probe reports the result of an expression's evaluation and displays it next to the expression's source code in the programmer's code editor.

Babylonian Programming thus allows programmers to quickly obtain information about program's run-time behavior without the often required overhead of manually interacting with a program until it reaches the relevant part in the code, interspersing `printf` statements, or inspecting the state in the debugger. The probes minimize the spatial distance between source and values to support programmers in reading the output, as opposed to, for example, `printf` statements that appear in a separate window and where output from multiple statements collects at the same time. Notably, to answer questions where the order of execution is important, a log akin to `printf` output could be a preferable presentation of information.

3.1 Mapping from Interaction Tool to Text

As described in Section 2.3, we need to identify a suitable mapping from the interactive, visual tool to a text format that is usable by the LLM. This relates to two parts: the input

that users would give the tool and the output that the tool presents to the user.

In the case of Babylonian Programming, we observe two main inputs the user provides: the placement of examples at the top of a method and the placement of value probes around expressions in methods. The output of the tool reports values by displaying them in the probe user interface element nested in the source code. Commonly, a Babylonian Programming implementation reports at least the last value captured as well as a count of values that this probe captured.

On the highest level, the communication is commonly facilitated by a simple tool calling interface, such as the model context protocol (MCP) [2]. The MCP is a standardized protocol that defines how applications and tools expose their functionality to LLMs. For concrete information, there are many ways in which communication between the tool and the LLM can occur for both input and output.

For input, an approach is to describe to the LLM how probes are inserted into source code as expressions and have it transform the source so that the probes are placed in the desired locations. For example, in Smalltalk, we define a method `bpTraceWithId:` that would be used in the following way:

```
c := (a + b) bpTraceWithId: 'id43123'.
```

Here, a generated ID that is inserted along with the instrumentation allows the tooling to identify where reported values originate from without changing the value of `c`. Alternatively, line number and textual expression or column indices must be provided to the LLM.

For output, the LLM could either receive a mapping from probe ID to observed values or the observed values could be inserted in the textual source, and the relevant excerpts could be passed to the LLM. Here, experiments could be worthwhile if the LLM, too, would benefit from the spatial proximity of source to values or if additional mapping effort makes no difference in response quality.

For cases where the LLM temporarily modifies the source code on disk for instrumentation purposes, a form of lifecycle would be desirable. For instance, the project could be copied into a container for isolation purposes. Alternatively, the system could keep track of changes and revert to the original state after the execution of the example has concluded.

3.2 Language-dependent Instrumentation

The mechanism proposed above relies on source code instrumentation as a lightweight form to obtain run-time values. In an environment such as Squeak/Smalltalk [11], such instrumentation is almost trivial as the development environment and application run in the same Smalltalk image and the instrumentation mechanism can thus simply write into a global variable.

[2] https://modelcontextprotocol.io/docs/

For languages that separate development from application environments, an inter-process communication mechanism needs to be used. For example, observed values could be appended to a temporary file that is read out when execution concludes, or they could be sent to a web server that the system keeps open while the example is executing.

The source code instrumentation mechanism, as described above, will require the LLM or the system to insert a small preamble that defines the probe function. Given a sufficiently simple probe mechanism, such as appending to a file, an LLM could generate such a preamble for the target language on the fly, without the system having to store corresponding stubs for the various languages.

Depending on the use case, the probe should extract different information. Likely use cases include:

value Report the (serialized) value of the observed expression.

stack Report the functions on the stack when the probe was hit.

thread Report the ID of the thread the probe was hit in.

conditional Only report if the conditional evaluates to true, e.g., for filtering for specific values.

time Report the time that the execution of the wrapped expression took.

count Report the total count of values observed.

In some programming languages, obtaining this information is simple, in others, it requires more advanced instrumentation, where it may also be unlikely that the LLM can generate correct code on the fly. Consequently, storing language-specific stubs along with the system may be helpful in those cases.

4 Case Study

In our case study, we illustrate how programmers could benefit from an LLM that has access to dynamic run-time tools to answer debugging-related questions. We describe our design and usage of a debugging agent that is connected to both static tools for retrieving the source code of an application and dynamic tools for Babylonian Programming.

Our agent is designed to take questions from a programmer that arise during an interactive debugging session and are asked at a high to medium abstraction level, such as "why"/"why not" questions or questions about the origin or evolution of certain objects (Figure 2). Internally, the agent performs its own autonomous research process: it (1) translates the initial question of the programmer into one or multiple lower-level questions (e.g., when or in which context a certain code branch is reached), (2) formulates hypotheses for these questions, (3) interacts with static and dynamic programming tools to verify or falsify these hypotheses, and (4) iteratively continues to ask further questions until it has gathered all required information to answer the original question of the programmer. The programmer can interact

with the agent through a dedicated tool window or access it from an open symbolic debugger, where it can also use the call stack of the halted application as additional context. Thus, programmers can remain at a more conceptual level in their workflow and delegate specific questions to the agent.

We implement a prototype[3] of this agent in the Squeak/Smalltalk programming system using the conversational agent framework SemanticText[4] and GPT-4o via OpenAI's chat completions API with function calling. We set up the agent with a chain-of-thought prompt (approximately 600 words) detailing the context, intended workflow, and available tools for the task. We provide the agent with two classes of functions. First, functions for static code browsing enable the LLM to list the members of a class, retrieve the source code of a method, and navigate between callers and implementors of methods. Second, functions for Babylonian Programming allow it to instrument existing methods with probes and run examples to trace values. For instrumentation, the LLM can call `recompile(className, methodName, source)` to decorate the original source code of a method with `bpTraceWithId:` sends. For running examples, it can pass a Smalltalk code expression or script to an `eval()` function. To avoid dangerous side effects of evaluating AI-generated samples, we can execute all `eval()` expressions in a lightweight sandbox such as SimulationStudio's[5]; however, we have not yet implemented this in our prototype as we have seen few incidents during our experiments.

Example. Figure 3 shows an interaction with our prototypical agent.[6] In this example, a programmer is exploring the Smalltalk compiler of the Squeak system. While debugging a concrete compilation task, they are wondering about the origin of a specific bytecode in a method they are compiling. Instead of manually interacting with the debugger or system browsers to search the possibly related subsystems of the compiler for this information, they invoke our agent and ask it: "why does this method have the byte 92 at position 30?" Subsequently, the agent performs its own reasoning process by breaking down this question into smaller questions regarding the parsing of the method source, the contents of the AST, and the encoding of the method.

To answer these questions, the agent starts by browsing the definition of the `Compiler` class and the methods from the open debugger session. Next, it annotates those methods most relevant to the compilation process with Babylonian probes and reruns the example from the debugging session to gather additional run-time information about the variable types and executed branches. By this, it discovers domain classes such as `Parser`, `Encoder`, and `CompiledMethod` as well as concrete run-time data such as the structure of the parsed AST and the generated bytecodes of the returned method. In the same way, the agent gradually navigates further through the implementation of the compiler until it eventually discovers a method in an encoder that emits the specific bytecode. Finally, it summarizes its findings on the data flow and involved methods in an answer to the programmer, who can use this information to proceed with their overarching exploration in the relevant places.

[3] https://github.com/hpi-swa-lab/squeak-semantic-babylonian
[4] https://github.com/hpi-swa-lab/Squeak-SemanticText
[5] https://github.com/LinqLover/SimulationStudio
[6] We provide the full conversation log here: https://hpi-swa-lab.github.io/squeak-semantic-babylonian/assets/Compiler.conversation.html

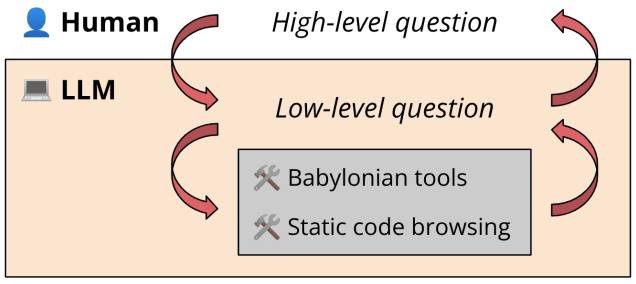

Figure 2. The design of our prototypical debugging agent. Humans ask conceptual questions to the agent, which develops low-level questions and hypotheses and tests them by interacting with tools for Babylonian Programming using run-time values and source code reading.

Current limitations. In our experiments, we noticed several weaknesses of our current prototype. First, the agent tended to terminate prematurely before conducting an exhaustive research of the program, which we mitigated by repeatedly typing "continue" into the chat. Second, the agent showed a low affinity for Babylonian interfaces but tended to mainly use static code browsing tools (presumably due to the low prevalence of relevant tool usage in the training data of the model). To mitigate this, we added structural coverage to its workflow by limiting the number of allowed browsing calls before instrumenting and running the next method. However, we believe that these problems can be addressed by systematically optimizing our prompts or using a fine-tuned model.

5 Related Work

To give a brief overview of related work, we first focus on research on LLMs' capabilities to reason about run-time information with and without tooling, and secondly discuss literature on how LLMs access external tooling.

Two peer-reviewed papers exist on the evaluation of the code reasoning capabilities of LLMs. CRUXEval was the first benchmark that analyzes input and output prediction of state-of-the-art language models [8]. They investigated how well a model can predict the output of small LLM-generated Python functions, based on an invocation and the source code. They discovered that models underperform, and Chain

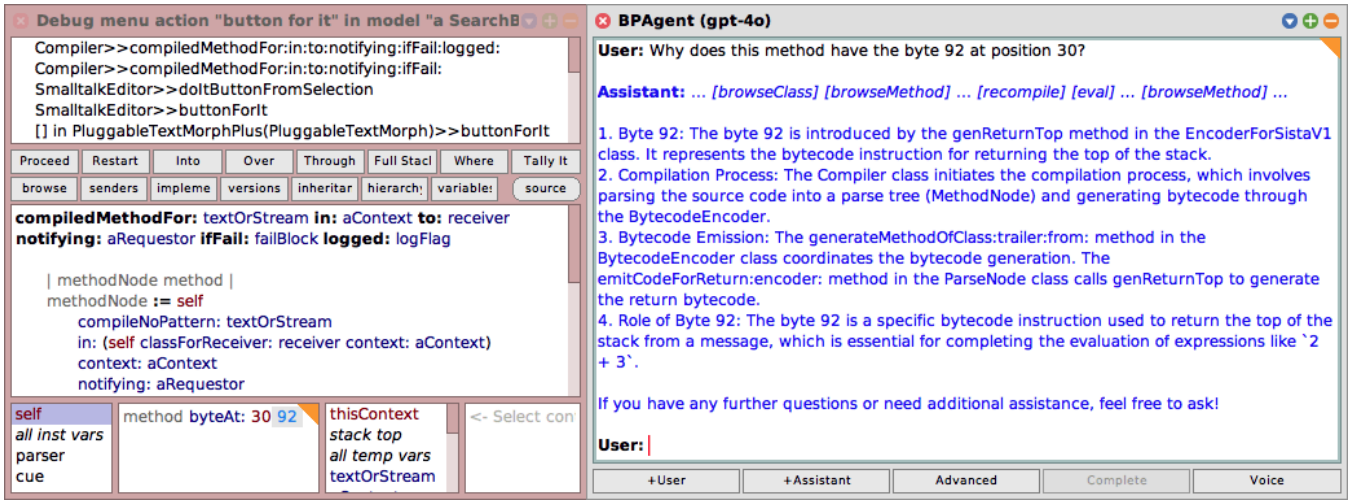

Figure 3. Invoking our prototypical debugging agent from a traditional debugger in Squeak. The agent browses and traces several methods to answer a question about the behavior of Squeak's compiler. (Conversation truncated for screenshot.)

of Thought prompting increases the accuracy. REval extends CRUXEval by adding intermediate predictions such as next-line and intermediate state prediction [2]. In addition, they introduced a metric measuring the consistency of the model, making predictions. They also concluded that the models show unsatisfactory performance. Both benchmarks focus on model prediction without tool calling.

Interesting studies regarding adding debugger capabilities to an LLM exist. Debug-gym is an interactive text-based environment that mimics a software engineer debugging a faulty codebase [30]. As one of its tools, it wraps the Python debugger (pdb) and provides the LLM with capabilities to add breakpoints, run tests, and rewrite lines to emulate how humans would debug code. In our approach, we rely on source code rewriting for Babylonian Programming instead of using an existing debugger, giving us more flexibility in information gathering. In addition, we aim to enhance the programmer's ability to debug instead of letting the LLM fix the bug itself. The Large Language Model Debugger [32] utilizes a control flow graph of generated code, their execution traces, and intermediate states. While they achieve better performance for code generation, they do not aim to explain and answer questions related to debugging.

Another related research is on how LLMs can be equipped with agentic capabilities and invoke external tools, such as developer tooling, to retrieve answers. Exploratory programming agents answer programmers' questions about objects from a running system by executing arbitrary code at run-time to access their state and behavior and conducting an internal research process [26]. SWE-Agent is a framework that allows LLM models to use external tools [29]. In their paper, they investigate how the interface design affects the performance of LLM agents and introduce a system that can modify, test, and execute a codebase. They report that the agent usage improved performance on two code generation benchmarks.

6 Discussion & Future Work

We argue that equipping LLMs with user-facing programming tools can benefit LLMs for programmers in two ways: First, developers can reuse existing tools, including the knowledge, mental models, and methods documented in them. Second, we believe that LLMs with user-facing tools can outperform traditional LLMs with only limited tool access.

As an initial evaluation of this claim, we have shown a prototype that uses Babylonian Programming to answer different questions about program behavior. In the future, we want to prove our hypothesis by comparing these equipped LLMs with traditional LLMs in a quantitative evaluation of accuracy, resource consumption, and user experience. To this end, we want to identify possible classes of use cases that benefit from run-time information and integrate further user-facing tools such as program tracers or profilers into our prototype.

We envision that future LLMs will enable for tighter interaction with existing tools and programmers. To achieve this vision, future research should focus on four aspects: explainability, resumability, context efficiency, and combining information sources.

Explainability. When LLMs solve problems using a set of tools that is analogous to those used by programmers, their reasoning has the potential to become more transparent and relatable. In the case of a Babylonian LLM interface, the LLM could internally reason in terms of examples and probes, allowing programmers to comprehend which examples led to which run-time values at which probes. These insights into how the LLM came to an answer do not need to be presented

as plain text. As Babylonian Programming has mechanisms to visualize run-time values, so can the reasoning done by the LLM be augmented by replacing serialized tool calls and values with domain-specific visualizations when programmers need to understand them retroactively.

Resumability. One particular advantage of LLMs using programmer-facing tooling facilities is that instrumentation artifacts, placed inside the code, can become reusable for programmers even after the LLM responds. A live example and corresponding probe the LLM came up with, or a particular debugging state, could be persisted and reused by programmers and future LLM calls, allowing programmers to "resume" from any reasoning state the LLM has left. An open question is how to design the user interface so that programmers can select instrumentation parts they want to keep, while discarding ephemeral examples and probes.

Context efficiency. Tools for programmers are often designed to reduce information overload. For example, a complex object tree can be displayed with expandable nodes, only showing those children that a programmer is currently interested in, while long strings and lists might appear truncated until explicitly selected for inspection.

Offering facilities that effectively select or focus relevant information to an LLM and "collapsing" information that is out of focus can ultimately reduce context size, thus minimizing the risk of polluting the context with distracting data and even saving computational resources. These savings would translate to faster round-trip times in conversations and less energy consumption.

Part of this yet unexplored design space will be how much information to provide eagerly in advance and how much the LLM needs to fetch lazily after being presented with an initial result from its tool call. A *query language* that specifies which information the LLM is interested in could be a useful tradeoff to explore. For example, instead of receiving all inputs and outputs of a function it investigates, the LLM could specify ahead of time that it is only interested in those where the function raised an exception, or only certain fields of a collection of objects. Extending the query language with abstractions to sample, aggregate, or group data could enable the LLM to further optimize yet-unseen data before it enters its context, ultimately serving as a natural-language interface to a query-based run-time inspection system.

Combining information sources. Run-time inspection tools provide stack traces and performance data such as timing, memory usage, page faults, and garbage collector statistics. Making such information available could enable an LLM agent to reason about the performance characteristics of concrete data, answering questions like, "Is there a specific shape or access pattern of my data structure that causes performance to degrade?"

Combining such information is not an easy task for programmers and requires the juxtaposition of code, concrete data, and profiler output. How we could combine such information in a serialized and efficient form for the LLM is yet to be explored. A hypothesis might be to interleave code with run-time data (including profiling information) as shown in our prototype, keeping related data in close proximity.

7 Conclusion

In this paper, we presented a first attempt at bridging the gap between LLMs used during development and human-centered dynamic tooling, enabling LLMs to gather and reason about run-time information better.

Our literature review showed a wide range of questions programmers ask and the tooling they use to solve them. LLMs should also leverage programmers' tooling to answer programmers' questions, particularly tooling designed to extract insights from concrete run-time data. Using Babylonian Programming as an example, we presented an integration of a tool initially designed for programmers into an LLM-based agent and demonstrated its practicality in a case study. However, we also noticed that current LLMs are still biased toward traditional ways of inspecting programs and might benefit from fine-tuning.

Looking forward, continued research in this direction can result in several opportunities for LLM agents: LLMs can make their reasoning process more transparent by "acting" during their reasoning in *in terms of human-facing tools*. Their investigation can become *resumable*, enabling programmers to reuse and build on artifacts and instrumentation previously used or produced by the LLM. Investigating more expressive ways for the LLM to specify the information it needs, e.g., through a *query language*, promises to reduce context size and improve overall efficiency.

Ultimately, combining and synthesizing multiple sources of static and dynamic data inspired by state-of-the-art programmer tools can improve the correctness, coherence, and depth of LLM answers to programmers' questions.

Acknowledgments

We sincerely thank the anonymous reviewers for their detailed and valuable feedback. This work was supported by SAP and the HPI–MIT "Designing for Sustainability" research program[7].

References

[1] Muhammad Zeeshan Asghar, Khubaib Amjad Alam, and Shahzeb Javed. 2019. Software Design Patterns Recommendation : A Systematic Literature Review. In *International Conference on Frontiers of Information Technology, FIT 2019, Islamabad, Pakistan, December 16-18, 2019*. IEEE, 167–172. doi:10.1109/FIT47737.2019.00040

[7] https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html

[2] Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. 2025. Reasoning Runtime Behavior of a Program with LLM: How Far are We?. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 1869–1881. doi:10.1109/ICSE55347.2025.00012

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[4] Luca Di Grazia and Michael Pradel. 2023. Code Search: A Survey of Techniques for Finding Code. *ACM Comput. Surv.* 55, 11, Article 220 (Feb. 2023), 31 pages. doi:10.1145/3565971

[5] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *J. Softw. Evol. Process.* 25, 1 (2013), 53–95. doi:10.1002/SMR.567

[6] Ekwa Duala-Ekoko and Martin P. Robillard. 2012. Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE, 266–276. doi:10.1109/ICSE.2012.6227187

[7] Thomas Fritz and Gail C. Murphy. 2010. Using Information Fragments to Answer the Questions Developers Ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 175–184. doi:10.1145/1806799.1806828

[8] Alex Gu, Baptiste Rozière, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. 2024. CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. https://openreview.net/forum?id=Ffpg52swvg

[9] Jia He, Mukund Rungta, David Koleczek, Arshdeep Sekhon, Franklin X. Wang, and Sadid Hasan. 2024. Does Prompt Formatting Have Any Impact on LLM Performance? *CoRR* abs/2411.10541 (2024). arXiv:2411.10541 doi:10.48550/ARXIV.2411.10541

[10] Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. 2022. Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA (Proceedings of Machine Learning Research, Vol. 162)*, Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato (Eds.). PMLR, 9118–9147. https://proceedings.mlr.press/v162/huang22a.html

[11] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '97)*. Association for Computing Machinery, New York, NY, USA, 318–326. doi:10.1145/263698.263754

[12] Ranim Khojah, Mazen Mohamad, Philipp Leitner, and Francisco Gomes de Oliveira Neto. 2024. Beyond Code Generation: An Observational Study of ChatGPT Usage in Software Engineering Practice. *Proc. ACM Softw. Eng.* 1, FSE (2024), 1819–1840. doi:10.1145/3660788

[13] Amy J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE, 344–353. doi:10.1109/ICSE.2007.45

[14] Amy J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 301–310. doi:10.1145/1368088.1368130

[15] Thomas D. LaToza and Brad A. Myers. 2010. Developers Ask Reachability Questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 185–194. doi:10.1145/1806799.1806829

[16] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-Answer Questions about Code. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU 2011, Reno, NV, USA, October 17-21, 2010*, Emerson R. Murphy-Hill, Shane Markstrum, and Craig Anslow (Eds.). ACM, 8:1–8:6. doi:10.1145/1937117.1937125

[17] Toni Mattis, Eva Krebs, Martin C. Rinard, and Robert Hirschfeld. 2024. Examples out of Thin Air: AI-Generated Dynamic Context to Assist Program Comprehension by Example. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming, Programming Companion 2024, Lund, Sweden, March 11-15, 2024*, Emma Söderberg and Luke Church (Eds.). ACM. doi:10.1145/3660829.3660845

[18] Andreas Opedal, Alessandro Stolfo, Haruki Shirakami, Ying Jiao, Ryan Cotterell, Bernhard Schölkopf, Abulhair Saparov, and Mrinmaya Sachan. 2024. Do Language Models Exhibit the Same Cognitive Biases in Problem Solving as Human Learners?. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net. https://openreview.net/forum?id=k1JXxbpIY6

[19] L. Prechelt, B. Unger, W.F. Tichy, P. Brossler, and L.G. Votta. 2001. A controlled experiment in maintenance: comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering* 27, 12 (2001), 1134–1144. doi:10.1109/32.988711

[20] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming - Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *Art Sci. Eng. Program.* 3, 3 (2019), 9. doi:10.22152/PROGRAMMING-JOURNAL.ORG/2019/3/9

[21] Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: language models can teach themselves to use tools. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2997, 13 pages.

[22] Hyunmin Seo, Caitlin Sadowski, Sebastian G. Elbaum, Edward Aftandilian, and Robert W. Bowdidge. 2014. Programmers' build errors: a case study (at google). In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 724–734. doi:10.1145/2568225.2568255

[23] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions Programmers Ask during Software Evolution Tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, Michal Young and Premkumar T. Devanbu (Eds.). ACM, 23–34. doi:10.1145/1181775.1181779

[24] Philipp Straubinger and Gordon Fraser. 2023. A Survey on What Developers Think About Testing. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Los Alamitos, CA, USA, 80–90. doi:10.1109/ISSRE59848.2023.00075

[25] Steven L. Tanimoto. 1990. VIVA: A visual language for image processing. *J. Vis. Lang. Comput.* 1, 2 (1990), 127–139. doi:10.1016/S1045-926X(05)80012-6

[26] Christoph Thiede, Marcel Taeumel, Lukas Böhme, and Robert Hirschfeld. 2024. Talking to Objects in Natural Language: Toward Semantic Tools for Exploratory Programming. In *Proceedings of SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Pascadena, California, USA) *(Onward! '24)*. Association for Computing Machinery, New York, NY, USA, 17 pages. doi:10.1145/3689492.3690049

[27] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. Object-Centric Time-Travel Debugging: Exploring Traces of Objects. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming* (Tokyo, Japan) *(Programming '23)*. Association for Computing Machinery, New York, NY, USA, 54–60. doi:10.1145/3594671.3594678

[28] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.

[29] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://arxiv.org/abs/2405.15793

[30] Xingdi Yuan, Morgane M. Moss, Charbel El Feghali, Chinmay Singh, Darya Moldavskaya, Drew MacPhee, Lucas Caccia, Matheus Pereira, Minseon Kim, Alessandro Sordoni, and Marc-Alexandre Côté. 2025. debug-gym: A Text-Based Environment for Interactive Debugging. *CoRR* abs/2503.21557 (2025). arXiv:2503.21557 https://doi.org/10.48550/arXiv.2503.21557

[31] Kechi Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin. 2023. ToolCoder: Teach Code Generation Models to use API search tools. *CoRR* abs/2305.04032 (2023). arXiv:2305.04032 doi:10.48550/ARXIV.2305.04032

[32] Li Zhong, Zilong Wang, and Jingbo Shang. 2024. Debug like a Human: A Large Language Model Debugger via Verifying Runtime Execution Step by Step. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, 851–870. doi:10.18653/V1/2024.FINDINGS-ACL.49

Received 2025-07-09; accepted 2025-07-28