Liveness in the Age of Agents: Are We Back to Compile-and-Run Cycles?

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

Marcel Taeumel
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
marcel.taeumel@hpi.uni-potsdam.de

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

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

Abstract

An experienced programmer who works happily motivated within a live programming environment can yield high-quality software within reasonable time. A novice can also perform well because live programming tools foster short feedback loops, which help translate thoughts into executable program code and thus the desired observable system behavior. In the age of agents, now, one might think that actual programming skills take a step back to let visions shine: express your goal in natural language and let the agents make even larger, coherent edits to the system.

However, agents will take their time to think, and programmers require skills to make informed decisions about an agent's generated results. How "live" is that? How "short" does this feedback loop feel?

In this paper, we take a look at Tanimoto's liveness levels 5 and 6 in combination with Norman's interaction "gulfs" to discuss the impact agents have on the overall programming experience. We realize that the agent support is far from *predictive* as prompts take noticeable effort to clarify (or derive) a programmer's current goals. We also conclude that while the interaction cycles might feel quite productive at times, the experienced "Gulf of Evaluation" can be too much for novices if the results are too sophisticated.

For the (near) future, we believe that the programming experience around agents can further be tailored to improve both prompting and reviewing so that this very promising technology does not take us back to the times where lengthy compile-and-run cycles disrupted our thoughts, flow, and motivation.

CCS Concepts

• **Software and its engineering** → **Software notations and tools**; Designing software; • **Human-centered computing** → **Natural language interfaces**; • **Computing methodologies** → Artificial intelligence.

‹Programming›Companion '26, Munich, Germany

ACM ISBN 979-8-4007-2555-5/26/03
https://doi.org/10.1145/3801119.3801131

Keywords

Live Programming, Coding Agents

ACM Reference Format:
Toni Mattis, Marcel Taeumel, Lukas Böhme, and Robert Hirschfeld. 2026. Liveness in the Age of Agents: Are We Back to Compile-and-Run Cycles?. In *Companion Proceedings of the 10th International Conference on the Art, Science, and Engineering of Programming (‹Programming›Companion '26), March 16–20, 2026, Munich, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3801119.3801131

1 Introduction

Until recently, programming meant to form an intent precise enough to express it in a formal notation, a programming language. Evaluating whether the resulting program matches the programmers' expectations relies on executing it[1]. Making it easier to connect abstract edits on the notation level to their consequences on concrete run-time behavior and vice versa is a core challenge in the design of programming environments and led, among others, to faster feedback cycles enabling smaller increments and powerful introspection tools that help programmers focus on the most relevant feedback. The experience created by immediate feedback, the impression of editing a running program, and tools capable of making run-time phenomena visible is loosely defined as *live programming*.

We currently see the widespread adoption of LLM-based agents capable of generating larger, coherent edits. They typically respond to a *natural-language specification* ("prompt"), interacting with programming and life-cycle tools such as search, version control, and dependency management, to implement and evaluate the requested change.

As live programming motivates smaller, incremental changes that take effect immediately, the current trend suggests that agents are taking on increasingly large tasks, spending several minutes on LLM generation and interactions with the environment, and producing complex changes that require substantial effort to evaluate. Instead of the compile-and-run cycle that live programming helps to avoid, we are now facing a "generate-and-run-cycle" that brings

[1]There are cases that demand statically verifying a property, but the scope of this discussion will be behaviors that emerge from concrete data and scenarios.

back similar problems, such as interruptions in flow and difficulties connecting a change to its run-time consequences.

At the same time, live programming provides novel opportunities in agent-centered workflows, helping programmers to better specify and evaluate agent-suggested changes.

In this work, we view this conflict under the lens of Tanimoto's liveness hierarchy, notably how the extension to liveness Levels 5 and 6 hold up to current agent-based workflows, and Norman's seven-stage model of action, where the meaning of the "Gulf of Execution" and "Gulf of Evaluation" in programming fundamentally shifts in the light of agent use.

We employ this perspective to propose and discuss future directions where live programming might be exactly the right paradigm to improve interactions with agents: When it helps programmers understand the effects of small changes, why would it not help understand the large changes produced by agents? Would live programming not make an agent faster, just as it makes programmers faster?

2 LLM-based Agents

LLM-based Agents are natural language interfaces designed for programming tasks that implement an intent of a programmer. Programmers conversationally formulate their intent in natural language. The intent formulation ranges from a brief command to a detailed specification of an implementation. Based on the intent, the LLM-based Agent searches the codebase using tools to gather context information. The context information serves as a basis to generate code changes. Code changes proposed by the agent depend on the programmer's intent, the derived context information, and the used model. They can range from minimal changes involving a single character to multiple thousands of lines of code. Examples of LLM-based Agents for programming tasks are Claude Code, Codex CLI, Google Jules and OpenCode.These agents share a similar architecture: all of them have tool integrations to allow for context retrieval and code changes.

Tools as interface to the environment. Besides the underlying model, tools are the central building block of an LLM-based agent. A tool consists of at least a natural language description of what the tool does, which parameters are available, and an implementation of the given tool. The model can invoke tools by generating the necessary input parameters. Using the input parameters, the tool is executed, and the results are fed back to the LLM as input information. This simple structure allows wrapping every task a programmer could perform as a separate tool. Common out-of-the-box supported tools are Unix-like command line tools such as *grep*, *glob*, *ls* to search folders using keywords to find relevant context information. Tools to interact with file content are *Read*, *Write*, and *Edit*. The fallback tool for arbitrary commands is *Bash* to execute system-installed command line tools.

Context management. The execution of tools is an act of context retrieval of the LLM. The agent continues the loop of tool executions until it is confident enough that it has gathered enough information for the given problem to continue. Continuing the task can be proposing an implementation plan, applying changes to the codebase using *Write* and *Edit* tools, or asking clarification questions. The amount of human oversight this loop involves depends on the agent's autonomy level and its current mode. Most LLM-based Agents support multiple modes, which prompt the model to either propose an implementation plan or directly apply changes without human oversight. A programmer can further tune model behavior using natural language configurations stored as configuration files.

Currently, no single workflow for LLM-based Agents has been established. Effective workflows for specific LLM-based Agent are learned through experience or by shared best practices from online communities. Common workflows range from meticulously approving each tool call to running multiple agent sessions concurrently in auto-approve mode to bridge the generation wait time.

Current limitations. Various workflows exist to circumvent the limitations of LLM-based models. LLM-based models inherit the limitations of general LLMs while introducing additional challenges specific to programming tasks. Context window limits constrain how much context can be considered simultaneously, forcing selective retrieval, without certainty that the used context is relevant to the intent [2, 6]. Incomplete context can lead to code duplication, as the agent generates functionality that already exists elsewhere. Outdated or incorrect context, such as referencing old API versions, leads to faulty code changes [5]. The context selection problem intensifies as the codebase grows, since the agent has to find relevant information in an expanding corpus while its context window remains fixed. Hallucinations proposes a plausible code that might be syntactically correct but fails at runtime [5]. Programmers also have limited transparency in what context the agent is using [4]. In current LLM-based Agents tools calls are often hidden or too numerous to follow, while the linear append-only chat interface makes it difficult to track included context.

Obstacles to immediate feedback. A major limitation is that tool calls and token generation take time, slowing the feedback loop for programmers. Although hardware development increases LLM processing speed *on average*, generating tokens that eventually make up tool calls, internal reasoning, and responses remains a bottleneck for multiple reasons.

Tokens are generated sequentially, so even if the economics of AI services constantly push hardware to better throughput by parallelization and batching, the speed of an individual answer depends on latency. This is only one factor alongside increasing model sizes, processing more requests at once, and allowing larger contexts. Thus, technical progress is only partially reflected in end-to-end latency. Growing model sizes also encounter memory bandwidth bottlenecks, as generating a single token can require accessing billions of parameters, which further incentivizes batching to reuse already loaded parameters across requests rather than speeding up a single request. Conceptually, almost all practically used LLMs rely on the transformer architecture, in which generating a token inherently requires considering every previous token, leading to quadratic complexity that slows down generation. Beyond the LLM itself, tool calls can add significant overhead as the LLM has to wait for files to be read (file system and storage latency), commands to execute (inter-process communication), and sometimes remote API calls to return (network latency). In cases of errors, the agent might have to read error messages and cycle through a series of tool calls

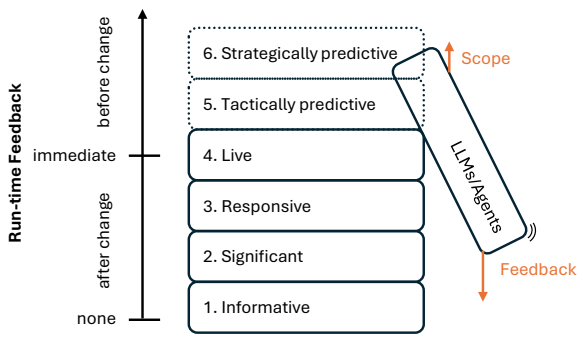

Figure 1: The liveness hierarchy according to Tanimoto, extended with the role LLM-based agents might play when used in a live programming context. Extensive generation time lengthens feedback cycles, so that interactions are no longer live; however, the scope of changes that such an agent can propose has grown to strategic levels.

again. As a result, programmers must wait a significant time before they can inspect generated code and validate correctness [12].

Current trends point toward agents that will work autonomously for longer stretches of time, possibly even days [2], setting off in a direction that eliminates human interaction rather than speeding up feedback.

3 The Liveness Hierarchy

The programming experience created by a specific programming environment or system can be classified according to its *liveness*. Tanimoto described a hierarchy of initially four liveness levels, later refined to six levels [10]:

(1) Informative: Computation is described using a notation.
(2) Executable: That notation can be executed, yielding a program with observable behavior.
(3) Edit-triggered updates: The running program adapts to a notation change on request.
(4) Live: The program continuously reflects the notation, feedback is immediate.
(5) Tactically predictive: Programmers can select from future running behaviors inferred by the programming environment.
(6) Strategically predictive: Large-scale functionality is inferred.

At the time of publication, levels 5 and 6 were positioned as the future of live programming, but in light of the rise of LLM-based agents, we must critically reflect on how close we are to this future or whether programming is taking a different direction.

3.1 Levels 5 and 6 in the Age of LLMs

Shift from programmer input to prediction. With the shift from reacting to user input to predicting the next running program, level-5 liveness aims to reduce the time to feedback "below zero" as new running behavior becomes visible before changes to the program have been made. This raises the question of how these adjacent programs can be inferred, as previous levels of liveness still relied on programmer input. Extrapolating from previous programming activity might work for small or recurring patterns (e.g., building an undo-operation into an editor is likely followed by implementing a redo-operation), but many changes are motivated by external requirements that a programming environment cannot know unless specified. Guessing them would raise the question of how many concurrent behaviors would be needed to cover a large enough space of meaningful future programs, while selecting from too many behaviors might get more tedious than specifying the desired behavior in a notation.

Inference through agents. Agent-assisted programming currently relies on natural-language cues in the form of comments or prompts. Rather than removing user input, its notation shifts from program code to unconstrained natural language. Technically, agents produce code, but in live programming environments, these suggestions can be easily executed and compared using programming environment extensions such as example-based programming as seen in Exploriants [1] or LEAP [3]. This means that the technical foundations to run future programs exist and are currently being researched; however, some form of specification, done by programmers, is still required, even if on a higher level than before.

Immediacy in agents. From the perspective of feedback, however, agents appear to be moving in the opposite direction. Tanimoto's liveness hierarchy accelerated feedback with each level, eventually overtaking programmer input from level 5 onward. Current systems, such as Claude Code, GitHub Copilot, or OpenAI Codex, spend a significant amount of time generating sequences of tokens representing internal reasoning and tool calls to interact with the environment before synthesizing a code change, effectively increasing the time between programmer input and a running program.

Thus, we observe two forces: The scope of changes a programming environment can propose at once has grown drastically, likely beyond the degree originally envisioned as "strategically predictive", while the speed and continuity of feedback has dropped back to compile-and-run levels, but instead of compile time we are confronted with *generation time* instead. Independent of the programming experience, the runtime environment may still be "live" in the sense that a change can be experienced in a running system immediately after it has been generated without losing context in between. Thus, a live programming environment operated through an agent remains live, but the programmer's experience does not.

4 The Seven-stage Model and its Gulfs

Norman's seven-stage model of action [7] describes how users interact with systems to achieve a goal. In terms of programming, our users are programmers, and the system consists of both its executable notation and run-time phenomena, connected via a runtime environment. Goals can be directed toward notation (e.g., refactoring), but in the cases of interest to us, they are requirements that change how the program behaves. We use a simple example of a jump-and-run game to illustrate the seven stages:

[2] https://www.anthropic.com/engineering/effective-harnesses-for-long-running-agents (last accessed 2026-02-06)

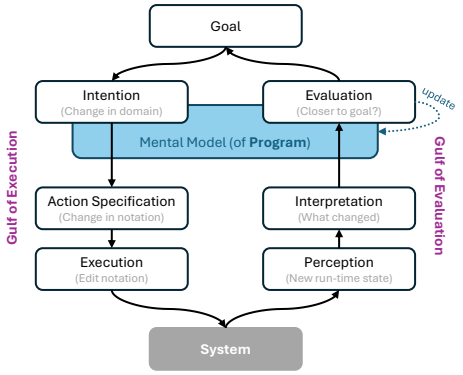

Figure 2: The seven-stage model after Norman, modified to match the (live) programming domain.

(1) *Goal*: The character should jump high enough to reach a platform.
(2) *Intention*: (Using knowledge about the implementation) "I must decrease gravity to allow for higher jumps."
(3) *Action specification*: "I will put half the current value in the gravity constant."
(4) *Execution*: Selecting the number in the code editor, typing to replace it.
(5) *Perception*: See the character follow a much slower, wider jump trajectory.
(6) *Interpretation*: "It jumps high enough, but also slower"
(7) *Evaluation*: Goal partially reached, but slowing down the fall was not a goal (Updating knowledge that gravity influences both height and fall speed, might need to reset gravity and adjust jump speed as next intent).

Stages 2 – 4 form the *gulf of execution*, where mental effort is focused on translating the goal into actions, while stages 5 – 7 form the *gulf of evaluation*, where the concern is judging how closely the actions brought the user to the goal. Repeated interactions with a system shape a *mental model* of the system, which allows users to perform mental simulation of each action to find the best action sequence that fulfils the intent. This model is updated every time the evaluation of the system state uncovers a discrepancy between the mental simulation and the observed system behavior. An overview is presented in Figure 2.

4.1 Live programming, Agents, and the Gulfs

Live programming has a direct impact on the Gulf of Evaluation and the formation of a mental model. The time between execution (a change on notation level) and perception is minimal. The running program facilitates easier interpretation and is usually augmented with tools that enhance the observability of relevant behavior, such as introspection facilities that monitor specific runtime values or automated re-execution of tests or examples. At the same time, it facilitates small increments as the cost of perceiving the system after each change is small. Combined, these mechanisms compress the time required to accumulate feedback, explore an unknown domain, build tacit knowledge, intuition, or fine-tune a user experience.

While immediacy and relevance of feedback primarily help bridge the gulf of evaluation, notations and program editing tools can also bridge the gulf of execution. Direct manipulation, domain-specific languages, and graphical abstractions that replace text editing actions (e.g., tables, projectional editors, or graph-based notations) can provide condensed affordances, making certain intents easier to express (usually at the expense of others) and thus synergizing with the live programming paradigm.

Agents collapse the Gulf of Execution. With LLM-based agents, this mapping shifts: The actions are no longer performed by programmers; instead, the goal can be directly communicated in natural language. Forming a concrete intent within the program's domain becomes optional, as long as the agent can guess from the natural language specification what needs to be done in the program. Actions are generated by an LLM and performed through agent tools provided by the environment, such as protocols like MCP or plain shell commands.

Agents widen the Gulf of Evaluation. However, while the agent abstracts away the gulf of execution, the gulf of evaluation widens: Natural language descriptions can describe much larger program transformations than individual syntactic changes, resulting in a considerably different system state. Perceiving large changes is challenging, as demonstrated by ongoing research in the field of code review, and their interpretation occurs at a higher level (e.g., rather than merely observing whether a concrete value changes as expected, a newly implemented feature must now be thoroughly tested). Moreover, the executable notation, as a medium where actions persist, is no longer part of the Gulf of Execution but becomes part of the Gulf of Evaluation if programmers need to assess the consequences of their agent interactions on, for example, the program's architecture or other notational properties they would intentionally edit themselves in traditional programming.

The Gulf of Envisioning. The "new action" to be performed now is expressing the goal or intent in natural language. Also, the mental model previously shaped by notational actions and observing their consequences is now vastly more complex, as the novel available actions interact with the combined system that constitutes the agent and the program. Thus, finding that the results of an agent interaction are not satisfactory can mean that the agent failed (e.g., because the user's mental model of the agent overestimated its capabilities) or that it did exactly as expected, still resulting in the wrong program (e.g., because the mental model of the program expected a behavior that was actually different).

Subramonyam et al. [9] analyzed how Norman's model holds up when interacting with LLMs. Their refined model incorporates the facts that users lack a mental model of how LLMs work, as well as a well-defined set of actions, as the only input is unconstrained natural language. Now that an LLM starts to work *without a well-formed intent* and will always yield some result, while natural language does not provide any affordances similar to a well-defined set of actions, they identified the importance of translating a goal into an unambiguous intent and established that phase as *Gulf of Envisioning*. We argue that bridging the Gulf of Envisioning should

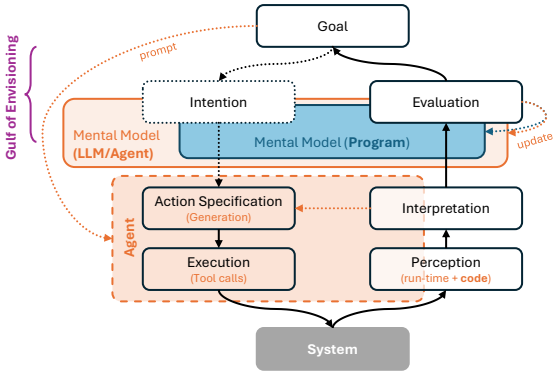

Figure 3: Seven-stage model after Norman and Subramonyam et al., modified to highlight the combined mental model of agent and program and the parts collapsed through the use of the agent. The Gulf of Envisioning spans the mental activities of forming an intent and communicating with the agent, and is harder to bridge as natural language no longer provides any affordances and constraints that editing a programming language might have. At the same time, evaluation now encompasses the full scope of what the agent did to the system, not just the changed behavior, and mismatches must update both the mental model of the program and the agent.

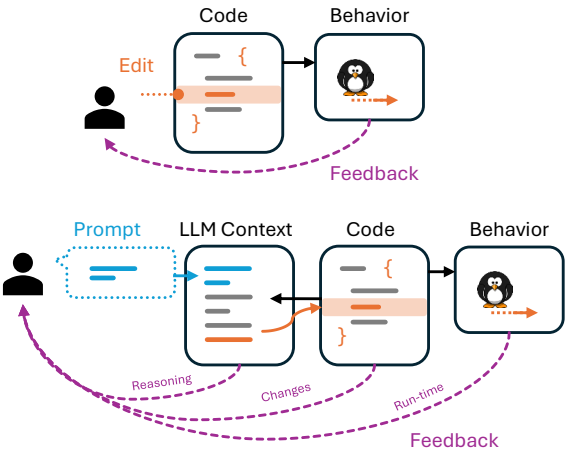

Figure 4: Compared to live programming (top) where edits translate directly to observable behavior, feedback loops with an agent (bottom) are much longer. The feedback now consists of three parts: (left) What the agent did and why, (middle) which effects it had on the code, and (right) the run-time consequences. Liveness can help communicate all three perspectives of feedback.

be a primary objective of designing programming environments that incorporate LLM-based agents.

5 Live Agentic Programming

Although programmer-agent interactions lose essential qualities of liveness, the ideas of immediate feedback and continuously available run-time information can provide substantial improvements to the programming experience when programmers pair up with agents. However, the feedback now takes a different shape, as not only does run-time behavior change in response to code, but code and behavior also change in response to a prompt in a complex way (see Figure 4).

Shaping concrete intents. As demonstrated by the Gulf of Envisioning, forming a precise intent is key to effectively conversing with agents and interpreting feedback. Thus, we propose that intent be clarified "live".

The presence of run-time data should allow programmers to reference live data and objects in their conversations, ideally leveraging direct manipulation to demonstrate goals using concrete examples. At the same time, the agent can ask clarifying questions grounded in run-time data. This should involve suggesting multiple ways to interpret a natural-language specification, similar to Tanimoto's level 5 liveness, in which programming is driven by selecting from running behaviors but with more programmer input.

Combining systems that provide means to explore parallel program versions and select from them, such as Exploriants [1], with approaches that converse with run-time data appears to be a start. The work of Thiede et al. [11], for example, integrates LLM-based

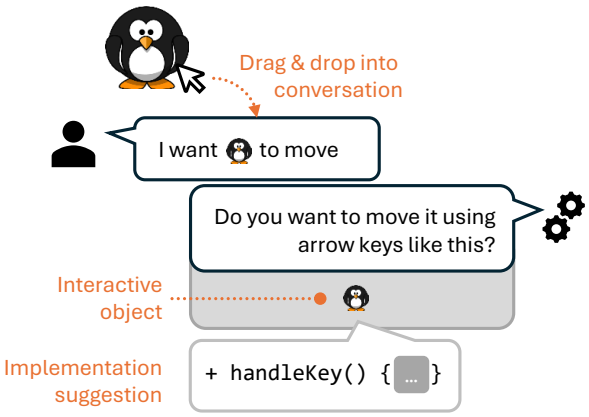

Figure 5: Grounding conversation in live objects. Programmers can reference run-time objects in their input, while the agent explains suggested changes using these objects as an example, which allows programmers to directly experience the consequences of the suggestion in terms of their chosen run-time objects.

chat into introspection tools to talk to the currently inspected object. Example-based live programming, such as Babylonian Programming [8], can provide a framework for managing conversation-level examples and persisting them across conversations.

An open challenge is the seamless integration of run-time artifacts into the conversation, e.g., providing interactions to "drag-and-drop" objects into a prompt and enabling agents to refer to objects in their response. Objects should present themselves using rich representations for the user and expressive tooling for the agent to process them.

This way, programmers can be more precise in their specification while agents can assist intent-forming using concrete data as the shared artifact between the programmer, the runtime, and the agent.

Evaluating suggestions. The second challenge we discussed is the widening Gulf of Evaluation, as increments are getting larger and changes to the program require evaluation from both code and run-time perspectives. A study by Ferdowsi et al.[3] demonstrated that reviewing a code change alone may be insufficient to judge its adequacy or select the most promising among several suggestions, and that live insights into how behavior and concrete state changes facilitate the evaluation of suggestions.

In the future, we expect to be able to see the consequences of agent-suggested changes in a running system, ideally demonstrated using well-scoped, representative examples that trigger exactly the changed behavior. As described before, these examples could include objects referenced by programmers, but must also consider novel edge cases or even invariants (e.g., example scenarios that intentionally did not change their outcome). We see open challenges in automating the construction of such a "frame of explanation", selecting effective example inputs, visualizing differences and invariants in behavior, and connecting them back to possibly substantial code and architecture changes.

Updating the mental model of the agent. Since the programmer's mental model of both agent and program will be involved in intent formation and evaluation, programmers would also need an opportunity to better understand what the agent knows and decides. Complementing run-time and code perspectives, we propose to include a third perspective in which the agent explains itself. This can include information that was pulled into its context and the feedback it received during experimentation, creating opportunities to spot misunderstandings (e.g., when the agent overlooked a file because it used the wrong search term). Ideally, this explanation is a first-class artifact itself that can be manipulated, and its consequences seen live. Programmers could experiment with whether the result improves if missing information is added or misleading information is hidden from the agent context.

Managing generation time. A third challenge is the potentially lengthy feedback time that occurs when agents perform a larger task autonomously. First, we suggest using this time more effectively, similar to how pair programming allows two programmers to work through a problem together, rather than one person delegating a task, waiting, and then needing to review the results all at once.

In such a setting, the agent could continuously explain what steps are taken. Technically, programmers can follow the console output of the agent; however, practically, this should involve live demonstrations rather than just abstract textual descriptions. Some mental load involved in evaluating the outcome could thus be front-loaded. This also connects to our previous suggestion that the agent should explain itself, as some explanation can also occur just in time rather than retroactively.

Second, live programming itself could shorten the time the agent needs. Agents often rely on running the code they produce to gather feedback, potentially building their own ad-hoc instrumentation (e.g., placing logging statements and subsequently consuming the log file) or generating tests. Since live programming enables humans to gather feedback more efficiently, it will likely also benefit agents. This opens up the design space for exposing live and run-time tools to agents, ranging from smaller, low-level tools like probes and inspectors to high-level tools such as profilers, call graph analyses, or omniscient debuggers.

6 Conclusion

LLM-based agents help programmers generate increasingly larger functionality at the cost of much slower feedback. Previously, live programming helped programmers form a better understanding of how their code changes affect a running program. Now that the activity of programming with the agent itself is no longer live, we see the need to use live programming as the paradigm that helps programmers better understand how the agent works with their program and which changes it implements.

Grounding conversations with the agent in live state and behavior, evaluating results continuously in a running system as the agent produces them, and having the agent explain itself are strategies that can bridge both the new Gulf of Envisioning and the traditional Gulf of Evaluation, now that agents appear to compress the Gulf of Execution. Moreover, we can make better use of the lengthy generation time by making generation observable "live" in a way that allows the programmer to learn alongside the agent. Providing live programming facilities (which helped programmers work faster) to the agent promises to shorten the overall feedback cycle.

Eventually, we expect the running program inside a live programming environment to become the shared artifact about which programmers and agents converse, not necessarily the underlying notation that programmers previously edited manually.

References

[1] Tom Beckmann, Joana Bergsiek, Eva Krebs, Toni Mattis, Stefan Ramson, Martin C. Rinard, and Robert Hirschfeld. 2025. Probing the Design Space: Parallel Versions for Exploratory Programming. *The Art, Science, and Engineering of Programming* 10, 1 (Feb. 2025), 5:1–5:33. doi:10.22152/programming-journal.org/2025/10/5

[2] Yufeng Du, Minyang Tian, Srikanth Ronanki, Subendhu Rongali, Sravan Bodapati, Aram Galstyan, Azton Wells, Roy Schwartz, Eliu A. Huerta, and Hao Peng. 2025. Context Length Alone Hurts LLM Performance Despite Perfect Retrieval. *CoRR* abs/2510.05381 (2025). arXiv:2510.05381 doi:10.48550/ARXIV.2510.05381

[3] Kasra Ferdowsi, Ruanqianqian (Lisa) Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. 2024. Validating AI-Generated Code with Live Programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3613904.3642495

[4] Shaokang Jiang and Daye Nam. 2025. *An Empirical Study of Developer-Provided Context for AI Coding Assistants in Open-Source Projects.* arXiv:2512.18925 [cs] doi:10.48550/arXiv.2512.18925

[5] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, and Li Zhang. 2024. Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. *CoRR* abs/2404.00971 (2024). arXiv:2404.00971 doi:10.48550/ARXIV.2404.00971

[6] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. 12 (2024), 157–173. doi:10.1162/TACL_A_00638

[7] Don Norman. 2013. *The design of everyday things: Revised and expanded edition.* Basic books.

[8] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-Style Programming. *The Art, Science, and Engineering of Programming* 3, 3 (Feb. 2019), 9:1–9:39. doi:10.22152/programming-journal.org/2019/3/9

[9] Hari Subramonyam, Roy Pea, Christopher Pondoc, Maneesh Agrawala, and Colleen Seifert. 2024. Bridging the Gulf of Envisioning: Cognitive Challenges in Prompt Based Interactions with LLMs. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–19. doi:10.1145/3613904.3642754

[10] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming (LIVE '13)*. IEEE Press, Piscataway, NJ, USA, 31–34.

[11] 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 the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '24)*. Association for Computing Machinery, New York, NY, USA, 68–84. doi:10.1145/3689492.3690049

[12] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022, Extended Abstracts*, Simone D. J. Barbosa, Cliff Lampe, Caroline Appert, and David A. Shamma (Eds.). ACM, 332:1–332:7. doi:10.1145/3491101.3519665