

The Developer as Curator: A Canvas for Coding Agent Context

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

Michael Perscheid
michael.perscheid@sap.com
SAP SE
Potsdam, Germany

Tom Beckmann
tom.beckmann@kuas.ac.jp
Kyoto University of Advanced Science (KUAS)
Kyoto, Japan

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

Abstract

The quality of AI-generated code depends on the context that coding agents automatically receive as input. Today’s coding agents do not expose that context to the developer. When retrieval fails, the developer cannot distinguish whether a wrong output stems from flawed reasoning or from incorrect or missing input. We describe how coding agents retrieve context internally and derive three failure modes: wrong context, missing context and stale context. Existing mechanisms for managing context (compaction, session reset, and sub-agent delegation) are automated responses to token limits that give developers no way to correct these failures. We present Monet, a canvas-based development tool that positions the developer as curator of the context the agent receives. Monet is guided by two design goals: *context observability*, letting developers see every piece of gathered context and its current state, and *context curation*, letting developers add, remove, and refresh context through direct manipulation. On Monet’s canvas, each piece of context is a node that developers can remove, add, or refresh to correct the agent’s input before re-prompting. Case studies show that each mechanism catches retrieval errors that autonomous agents cannot self-correct. Monet opens a new design space for human-AI collaboration in agent-assisted programming, shifting the developer’s role from prompt author to context curator.

CCS Concepts: • Human-centered computing → Graphical user interfaces; Interaction design; • Software and

its engineering → Software development process management; **Software implementation planning**; • **Computing methodologies** → Natural language generation.

Keywords: AI-assisted programming, Coding agents, Context management, Human-AI interaction

ACM Reference Format:

Lukas Böhme, Tom Beckmann, Michael Perscheid, and Robert Hirschfeld. 2026. The Developer as Curator: A Canvas for Coding Agent Context. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! ’26)*, October 4–9, 2026, Oakland, CA, USA. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3840586.3843212>

1 Introduction

The quality of AI-generated code depends directly on the context the model receives, yet current tools make it difficult for developers to see or influence that context. Early AI-assisted programming tools kept developers in control: inline completions let them accept or reject one suggestion at a time [4, 24, 33]. IDE-integrated chat automated context assembly but still let developers select files and references to include. Today’s coding agents [45] explore codebases on their own, selecting which files to read, which tools to invoke, and when to stop gathering information, evaluated on benchmarks like SWE-bench [17]. Multi-agent systems push this further: sub-agents operate independently with a specific task and return summaries, leaving the developer with no observability into what was gathered or how [13, 15, 37].

While each step increased the number of tasks a developer can work on, it also made it harder to inspect or shape the context on which the agent operates. Autonomously gathered context can be wrong, incomplete, or outdated [23, 44], causing faulty output. The underlying cause for wrong, incomplete, or outdated context is that the required agentic context retrieval is unreliable. An agent may settle on a superficially relevant file while missing the module that actually



This work is licensed under a Creative Commons Attribution 4.0 International License.

Onward! ’26, Oakland, CA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2934-8/2026/10

<https://doi.org/10.1145/3840586.3843212>

matters [6, 44, 48]. It reads a file early in the conversation and continues reasoning over that version long after its own edits have invalidated it. When the gathered context is wrong, downstream generations are wrong too [7]. Instead of receiving a correct output, the developer spends time diagnosing and fixing errors in a context that is opaque to them.

Finding the errors and repairing them accounts for a large share of session time in AI-assisted coding [33]. Empirical studies further indicate that context-related failures cause the majority of errors in AI-generated code [48]. Liu et al. [26] analyze hallucinations in LLM-generated code and identify five common types. Three of these types trace directly to the input the model received rather than to its reasoning over that input. Generated code misses the developer’s actual intent when the prompt and the retrieved files do not convey what the task requires. Generated code contradicts the rest of the codebase when irrelevant or outdated information accumulates in the context window [11, 23]. Generated code misuses APIs and project-specific identifiers when the model relies on definitions it never retrieved or retrieved incorrectly [48]. Together these three types account for approximately 79% of observed hallucinations. This suggests the cause is not that the model reasons poorly but that it reasons over the wrong information.

Current tools only give developers limited agency and inspectability over context. Some coding agents let developers inject context explicitly via the prompt, for example through file mentions or project-level rule files [16]. These mechanisms give developers a way to supply additional information to the agent. However, there is no tooling to inspect what context the agent already gathered, whether it is still current, or how it differs from what the developer expects the agent to be working with. Even when the developer suspects a retrieval error, no context management mechanism allows removing the context that the agent gathered incorrectly. The existing context management mechanisms offer no fine control. Compaction, also known as LLM-based summarization, replaces earlier messages with a lossy summary produced by a separate LLM call, condensing reasoning, actions, and observations alike into a single compressed representation [25]. Session reset and rewind discard the conversation history, removing valid context alongside degraded context indiscriminately. Sub-agent delegation surfaces only a summary of the sub-agent’s findings rather than the full retrieved information [13]. All three are triggered by context window constraints rather than by an explicit developer decision to verify or correct the contents.

To address these challenges, we present Monet, a canvas-based development tool that makes context a visible, manipulable artifact. Monet is guided by two design goals: *context observability*, letting developers see every piece of gathered context and its current state, and *context curation*, letting developers add missing context, remove wrong context, and refresh stale reads. On Monet’s canvas, every piece of context

becomes an individually addressable node that developers can inspect, add, remove, or refresh through direct manipulation. Multiple conversations can run in parallel on the same canvas, and validated context can be reused across them. Together, these goals give the developer a new role: beyond writing prompts and reviewing output, the developer becomes a curator of the context the agent operates on.

The remainder of this paper traces the decline of developer agency across five generations of AI-assisted programming tools, analyzes how coding agents retrieve context and why this process might fail. We then present Monet, describe how the tool lets the developer curate context on a canvas, how observability features surface context state, and how direct manipulation lets developers curate the agent’s input. Case studies demonstrate how these mechanisms address each failure mode in scenarios drawn from real development sessions. We close with a discussion of when curation helps, why current tools hide context, and directions for future empirical validation and survey related work.

2 Coding Agents

This section examines coding agents, their context problems and the consequences. We first trace how each generation of AI coding tools traded developer agency for automation, leaving today’s agents with little developer oversight. We then explain how coding agents retrieve context through autonomous tool calls, and why this process is unreliable. Following, we describe three context failure modes: wrong, missing and stale context. Finally, we review context management mechanisms that current tools use and show that none of them let developers inspect or curate what the agent has gathered.

2.1 The Decline of Developer Agency

We define *developer agency* as the degree to which a developer can observe and manipulate the actions an AI tool takes on their behalf. This includes both observability into the tool’s inputs and outputs and the ability to influence what it reads, generates, and modifies. Since the adoption of AI assistance during programming, this agency has been declining. Developers report difficulty controlling AI coding tools and not knowing what input shaped the output as among the most frequent usability issues [24]. Each iteration of tooling traded developer agency for automation. In the following, we briefly describe five kinds of tools with decreasing degrees of agency, which are also illustrated in Fig. 1.

Inline auto-completion Inline completions are code suggestions an AI model generates during programming at the current cursor position, typically displayed as gray text. Inline completions could be accepted or rejected using a predefined key. While the context the model used to generate a suggestion was already

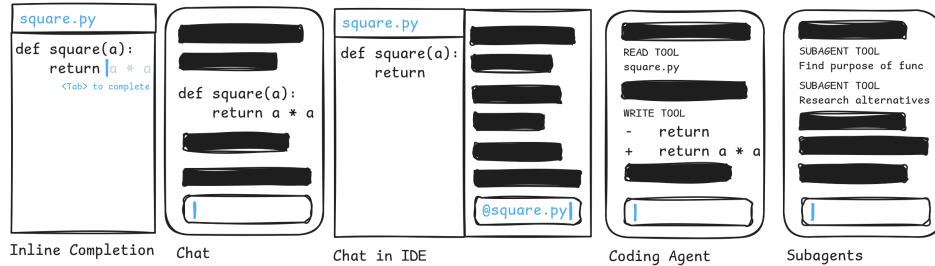


Figure 1. Five kinds of tools for interacting with AI models.

opaque to the developer, agency was preserved because developers could accept or reject each suggestion individually. Each completion was a small, local edit that the developer explicitly approved or dismissed before moving on [4, 33]. The developer remained close to the code, reasoning at the level of individual lines and functions.

Chat-based interfaces General-purpose chat models offered a different trade-off. Developers pasted snippets, asked for explanations, and drafted functions manually with the assistance of the LLM [21]. Because input and how the output was used were entirely under the developer’s control, agency was high. In practice, developers used chat primarily for guidance and learning rather than generating ready-to-use artifacts. They struggled to provide sufficient context when they did attempt artifact generation [21].

Chat-based IDE integration The subsequent integration of chat into IDEs attempted to combine both approaches, trading some of this control for tighter tool integration. IDE-integrated chat interfaces let developers ask questions about their code directly in the editor. These tools lacked autonomous tool use and were limited to heuristics that automatically assembled context such as the current selection or opened files [34]. Developers could steer answers by highlighting specific code, retaining some agency over what the model takes as input.

Coding agents Terminal-based coding agents extended the chat paradigm with autonomous tool use to gather context information. Rather than relying on heuristics to infer context, these agents explore the codebase on their own, mostly through bash commands [42, 45]. These agents are typically able to execute the program they are working on, thus forming a feedback loop that allows the agent to iteratively improve upon its results. This autonomy removes the need to manually supply relevant files or feedback, but also reduces the agency of the developer. The developer’s role shifts from selecting context and approving individual edits to writing a natural-language specification upfront and reviewing the result afterward [9]. While the used

context is essential for the outcome, the developer only sets a direction with the prompt. This shift effectively returns developers to compile-and-run cycles, where they specify intent, wait for the agent to produce a result, and inspect the outcome rather than the steps that led to it [30]. Details about coding agents are discussed in Section 2.2.

Multi-agent systems Multi-agent systems are a further iteration on this concept. An orchestrating agent coordinates multiple specialized agents that each operate on their own context without developer oversight [13, 15, 37]. For example, Claude Code spawns sub-agents that explore the codebase independently and return only a summary to the parent agent.¹ This further reduces developer agency: the developer approves a final result without knowing what context each agent gathered or how their intermediate findings shaped the outcome.

As AI-assisted programming tools grow more autonomous, developers lose observability into what the tool reads and control over what it generates. Yet the quality of the output depends directly on the gathered context. Autonomous agents frequently select insufficient or incorrect context [23, 44]. Restoring this agency without sacrificing automation is an open challenge.

2.2 Context Retrieval in Coding Agents

Monet builds on the context retrieval of existing coding agents. We therefore introduce how they work to get context from the codebase.

A coding agent is an LLM paired with a runtime which allows tool execution that retrieves context information through repeated tool calls in a loop. This agentic loop to retrieve context information is displayed in Figure 2. This allows the agent to locate relevant files autonomously, without requiring the developer to supply them. The agent framework provides the tools, a runtime that executes them, and a control loop that feeds results back to the model. Together, these exchanges form a conversation: an ordered sequence

¹<https://code.claude.com/docs/en/sub-agents>

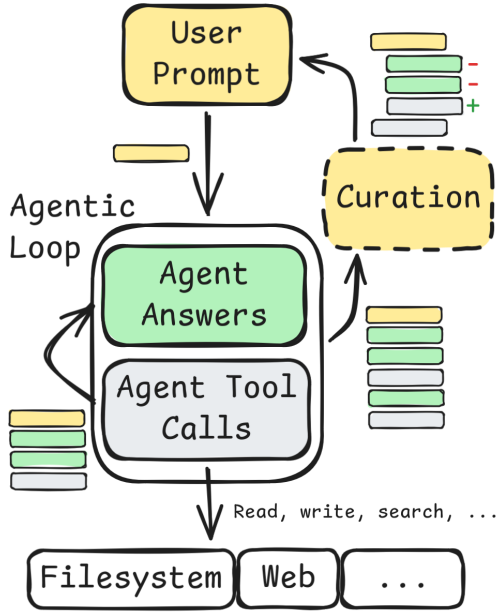


Figure 2. Agentic loop in coding agent context retrieval. The curation step (dashed) is not part of current tools and is introduced in Section 3.

of messages between developer, model, and tools. Each message appends to the conversation and becomes part of the context for the next model response. Throughout this paper, *context* refers to the complete input the model receives when generating its next response. This input is an ordered sequence of an initial prompt, tool calls and tool results, and prior model outputs from the current conversation.

While this automatic retrieval is a remarkable benefit, it offers little observability into what is retrieved or why. The developer expresses implementation intent in natural language as a prompt to the LLM. The prompt is the sole input for the coding agent. Important implementation details and relevant files need to be stated in the prompt to steer the coding agent to find specific information in the codebase. Based on the provided information in the prompt, the LLM executes an arbitrary number of tool calls [45].

The model has no prior knowledge of the repository’s state, but tool calls enable it to explore the codebase. Given the current context, the agent selects a tool from a set of textual tool descriptions and generates the corresponding parameters. Common tools include file search, text search, directory listing, a general-purpose shell, and reading and writing files [45]. The agent runtime executes each tool call on the host machine and feeds the output back to the model as input. Which tools are selected, in what order, and how often they are executed depends on the model and the given tool descriptions. Re-executing the same prompt most likely leads to different tool calls since the model is probabilistic [35]. The retrieved context therefore varies between runs, and

developers cannot anticipate which information the model will operate on.

The developer can steer this process only through natural language. A prompt or follow-up instruction may suggest where to look or what matters. However, the developer cannot determine which tools are invoked, which files are read, or what information the agent treats as relevant. The retrieval process is unreliable in several ways. The agent often stops searching and reading files once it finds related material, even if the retrieved context is incomplete for the task. Coding agents consistently prioritize gathering more material over gathering the right material [23]. If a search returns no results, the agent treats this as evidence that no relevant code exists. For example, the agent may search for the word ‘tag’ while the project uses ‘label’ instead, and an empty result can lead the model to conclude that no implementation of tags exists. Similarly, if you ask the coding agent an underspecified question such as ‘Give me the definition of a document’, it sometimes directly replies instead of searching for the class definition of the class ‘Document’. When asked ‘how input validation works’, the agent may find a validation function in a utility module and treat it as the complete answer. Empirical analysis of open-source projects shows that the majority of functions depend on entities defined elsewhere in the project, and agents frequently fabricate non-existent APIs, attributes, or variables when they fail to retrieve those dependencies [48]. It will miss that additional rules are enforced by middleware, database constraints, and frontend checks spread across several directories. The agent has no mechanism to verify whether all relevant information was found. All relevant information must either be specified in the prompt or happen to be found by the agent during its search, with no guarantee of the latter.

What the agent can find is further constrained by the tool set exposed to it, making the retrieval quality dependent on the surrounding system, not just the model. Prior work has shown that equipping LLMs with programmer-oriented tools changes which questions they can answer and how reliably they do so [5]. In multi-agent setups, the underlying codebase may change during retrieval if multiple agents operate concurrently, rendering previously gathered context invalid or out of date. None of these inconsistencies are surfaced to the developer. Even when retrieval succeeds, a separate problem emerges: the limited context size forces the agent to drop or summarize earlier findings as new information is added. What remains in the context is also not used evenly: information in the middle receives less attention than information at the start or end [27].

Agent frameworks differ in complexity, with more advanced ones adding components such as planning, specialized sub-agents, or structured retrieval on top of a basic tool-calling loop. Even these advanced frameworks offer only small gains in context retrieval over simpler ones, suggesting this is a fundamental limitation of coding agents rather

than an engineering problem [23]. The consequences of this are discussed in the following section.

2.3 Challenges of Unreliable Retrieval

Unreliable retrieval affects both the quality of the generated output and the ability to identify problems. For the purposes of our context curation design, we consider a classification of three different retrieval problems. For a more exhaustive classification, refer to Liu et al. [26].

Wrong context. Once the agent considers the retrieval finished, it generates code changes or a textual answer to the initial prompt. Agents tend to over-retrieve, gathering far more context than they ultimately use and introducing noise that appears to degrade downstream reasoning [23]. Errors such as irrelevant context during retrieval lead to errors in the generated output [22, 27, 39, 46]. The problem is not model capability but input quality: the model reasons over the wrong information [44]. Wrong context compounds a process that is already non-deterministic, since even identical inputs produce divergent outputs [11, 35, 39]. The generated code can be consistent with the gathered context but wrong for the entire repository. Only a developer familiar with the repository can recognize what was missed.

Missing context. Whatever was missed during retrieval is not available as reference during generation, leading to faulty output [6, 44, 48]. The agent stops as soon as its retrieval results seem *sufficient*, and what counts as sufficient depends on the model and its input and is not easily verifiable by the developer [23, 45]. Retrieval becomes the bottleneck once the agent has to discover the relevant structure. This bottleneck scales with repository size [48]. In small projects the agent can plausibly read everything relevant. In large codebases the relevant files are a small percentage of the entire repository. The probability that autonomous exploration finds them drops accordingly.

Stale context. Even context that was correct when it was gathered degrades as the conversation progresses. Every model has a maximum token budget. When the context window fills up, the agent is forced to summarize context, losing information that may still be relevant. Context rots: the agent’s own edits invalidate earlier reads, but the old versions remain in the context window [18]. Errors accumulate across the ongoing conversation, and there is no re-validation of earlier context after changes [43]. Inside the context window, the model cannot distinguish between the actual state of a file on disk and its own earlier output. As the context grows and accumulates stale information, model performance drops further because attention is spread across more tokens. Models reason better over small, accurate context than over large, polluted context [11, 27, 43].

The retrieval failures described above appear in the output as the hallucination types documented by Liu et al. [26].

Wrong and missing context lead to code that misinterprets the task or misuses APIs. Stale context leads to code that conflicts with the current state of the codebase. The same pattern holds in real-world projects, where context-related failures account for the largest share of generation errors [48].

The three failures above would be tractable if developers could inspect and edit the context window. Commercial tools such as Claude Code display which tool calls were made (for example, that the agent searched for five patterns) but hide the actual results returned, and because the interaction log is append-only, the developer cannot easily repair what the agent gathered or in what order. If the developer notices that a wrong file was retrieved, they cannot remove it from the context window. Asking the agent to ignore a file leaves the file in the context and adds the request on top of it [28]. The only fix is to restart the conversation with an additional hint in the prompt. The append-only model also reflects a broader regression in feedback granularity, where developers act on final outcomes rather than intermediate steps [30].

This opacity has direct consequences for trust and adoption. Developers frequently abandon generated code because it does not perform the intended action or fails to meet functional and non-functional requirements [24], yet current tools lack the affordances developers need to evaluate agent trustworthiness efficiently [41]. Code assistants present suggestions without exposing the specific codebase context influencing their generation [47]. When output quality is unpredictable and the causes are not inspectable, the developer falls back to manually verifying every result [24]. In the worst case, the agent no longer saves effort for the developer. The developer does the same work as before, plus the work of checking what the agent produced [24]. Without observability into the gathered context, the developer cannot distinguish successful retrieval from failed retrieval, and the append-only structure of the interaction log prevents them from curating it.

2.4 Context Management

A developer who wants to steer generation by improving the context, rather than reprompting or patching outputs, needs a way to inspect and curate what the agent uses as input. Because each model has a fixed context limit, how the space is used matters. Current agentic coding tools use three mechanisms that influence context size. None of them let the developer see, edit, or selectively keep what is inside.

Compaction Compaction, also known as summarization, replaces earlier context with a lossy summary to free up space once the context window fills. The summary itself consumes tokens, and because it is generated by a non-deterministic model, it can drift from the original source and introduce stale or incorrect information [35]. Compressing conversation history

into conversation summaries loses information and reduces downstream task performance [25, 43].

Session reset and rewind A session reset discards the entire conversation history. This removes faulty context but also throws out context that is still correct for the task. The developer and the coding agent start over with no carryover, forced to re-establish context from scratch. Session rewind offers finer granularity without added selectivity on what context should remain and which should be deleted. The developer rolls the conversation back to an earlier message, which discards every tool call and message after that point.

Sub-agent delegation Sub-agent delegation offloads a sub-task to a separate agent with its own context window [13]. The parent agent only receives a summary of what the sub-agent found, not the full retrieved context. The developer cannot check whether the sub-agent gathered the right information or from which information the summary was created.

Some coding agents compact and delegate tasks to sub-agents automatically without developer involvement. Session reset and rewind are developer-initiated but offer no selectivity over what context should remain. The developer can discard everything back to a chosen point, including the very start, or keep everything. The developer cannot see what is kept, remove a single wrong fact, re-validate a stale read, or inject knowledge that the agent missed. None of these mechanisms provide a channel for the developer to contribute codebase knowledge the agent failed to retrieve. The developer’s mental model of the codebase remains external to the agent, accessible only through the prompt.

3 Monet: A Curator’s Canvas

Monet is a canvas-based development tool that supports developers in curating context in collaboration with an AI coding agent throughout the development process. Figure 3 shows a screenshot of an open Monet project on the canvas. Developers become active curators of the context instead of being passive during context retrieval. State-of-the-art coding agents manage their context automatically, compacting or discarding information via compaction without developer agency [25, 45]. With the automatic context management, challenges such as context degradation and the approaching context limit negatively impact downstream code generation as described in Section 2.3. Monet gives developers an explicit view of the agent’s context and the ability to curate the gathered context actively.

The developer as curator. We use the term *curator* to describe a role that current AI-assisted programming tools do not support. A curator selects, organizes, and maintains a collection of artifacts. Applied to context, this means the developer decides which files, tool results, and prior outputs belong in the agent’s context window, removes what does

not belong, and keeps the remaining material current. The role is distinct from the prompt author and code reviewer. A prompt author specifies intent before generation. A reviewer evaluates output after generation. A curator acts on the input the agent reasons over before final output generation occurs or after to steer the model in the intended direction. The developer does not replace the agent’s retrieval but adds a curation step to it (see Fig. 2). They step in when it misses something, adding code the agent overlooked and cutting context that would hurt generation.

The remainder of this section first states the design goals that guided Monet’s construction, then describes how context can be curated on the canvas as observable, individually addressable objects, and finally presents the interactions that let developers manipulate those objects directly.

3.1 Design Goals

The challenges described in Section 2.3 fall into two categories. The first is an observation problem: the developer cannot see what context the agent gathered, whether it is complete, or whether it has gone stale. The second is an intervention problem: even when the developer suspects that context is wrong or missing, current tools offer no way to correct it beyond restarting or rewinding the conversation. Monet’s design addresses these two categories through two corresponding goals.

Context observability. The developer should be able to see every piece of context that the agent has gathered, its type of origin (file read, tool input and output, generated output), and its current state (up-to-date or stale). As discussed in Section 2.3, current tools only expose context through an append-only conversation log, leaving the developer without a structured way to see which files the agent read or whether earlier reads are still up to date. Without this observability, the developer has no basis to evaluate whether the agent has the information it needs or whether that information is still accurate.

Context curation. Seeing problems in the context is only useful if the developer can act on them. The developer should be able to add missing context, remove wrong context, and replace outdated context. As discussed in Section 2.3, wrong files remain in the context window indefinitely, missing files can only be hinted at through prompts, and stale reads cannot be refreshed. Current tools limit the developer to writing natural language descriptions, with no guarantee the agent retrieves the right information. Curation should replace this with direct manipulation of the agent’s input. Prior work supports this direction: reducing irrelevant context improves generation quality [11, 23], replacing stale information reduces errors in downstream output, and providing missing context increases the correctness of generated code [6, 44, 48]. Each interaction addresses a specific failure mode from the challenge analysis, letting developers express intent through

actions on context rather than through verbal descriptions alone.

Both goals augment rather than replace the agent’s autonomous retrieval: the developer can let the agent gather context on its own and intervene only when needed. This design shifts the developer’s role from writing prompts and reviewing results to actively participating in the generation process by curating the context.

3.2 Canvas Interface

Monet consists of two main views, as seen in Fig. 3: a file tree and a canvas. When a developer opens a project, this is represented as a workspace tab. In the newly opened workspace, they can then start writing prompts to interact with an agent.

Every part of an agent’s conversation context is represented as a separate node on the canvas: user prompts, tool calls, and agent responses each become an individual unit that the developer can inspect, move, or remove independently. We refer to these units as *context nodes* throughout the remainder of the paper. Each context node is freely movable on the canvas. Depending on the type of context node, different visualizations are shown. For example, a web search node shows the search results in a list with hover functionality to preview content, a read tool call node shows the file in a text editor with changes displayed in a diff view, and generated programming artifacts such as Mermaid diagrams are rendered directly on the canvas. Clicking a context node reveals a halo with navigation assistants and meta-information such as token usage. Nodes of specific types can be hidden or collapsed based on developer preferences. Hidden nodes remain on the canvas and can be restored at any time. Successive hidden nodes collapse into a single node that can be expanded again. Hiding is visual only: a hidden node remains part of the conversation context and is still sent to the model. This distinguishes hiding from context-altering operations described in Section 3.4: *removing* a node via drag-out excludes it from the conversation context while keeping it available on the canvas, whereas *deleting* a node discards it from the canvas.

Context nodes are organized into *chains*. A chain is a connected sequence of context nodes that represents a single conversation: the developer’s prompts, the files the agent retrieved, the tool calls it made, and the outputs it produced. Each context node of the chain is treated as part of the conversation context. Rearranging, deleting, or adding context nodes to the chain changes the context of the conversation. When the developer submits a prompt, only the context nodes within its chain are sent to the model for generating new context nodes, including the final outputs. We use the terms *conversation* and *chain* interchangeably.

A new conversation can be started anywhere on the canvas, in one of two ways. First, similar to state-of-the-art coding agents, the developer can create a new input context node on

the canvas via a double click. Submitting the prompt starts a conversation and generates new context nodes that appear on the canvas as a chain. Second, the developer can drag context nodes onto the canvas from a file tree (see Section 3.3) or reuse existing context nodes, arrange them as a chain, and ask questions about them using a new input context node. For example, a developer who wants to ask about the relationship between two modules can select both file context nodes and type the question, rather than explaining in the prompt what each module contains or where to find them.

Multiple chains can coexist on the canvas, allowing and encouraging exploration using multiple conversations at the same time. Prior context remains available, and there is no switching between sessions that cannot carry context. Rather than discarding an entire conversation to free up the context window, a developer can start a new chain while keeping the previous one visible and reusing context nodes. This is especially useful since context nodes persist after the conversation that produced them has ended and can be inspected, edited, or dragged into new chains as input.

The canvas persists across sessions. Prior conversations, context nodes, and spatial arrangement remain available when a developer returns. Earlier reasoning stays inspectable and gathered context can be reused. Because context nodes capture the state of files at the time they were added, persisted context may diverge from the current state of the codebase.

3.3 Observability

Changing context is only useful if the developer can understand what context exists and in what state it is. The observability affordances described below let developers see what needs changing; the interaction mechanisms described afterward let them act on those insights. Monet provides four affordances that communicate context information.

Color coding Every context node on the canvas carries a visual indicator of its state. Nodes fall into five categories: searches, reads, arbitrary calls, changes, and artifacts such as Mermaid diagrams. Each category has its own color. Nodes also indicate staleness via color. The system tracks which nodes reference which files. When a file changes after its last read, the node is marked stale.

File tree Alongside the spatial canvas layout, Monet provides a conventional file tree that lists the repository’s files. Developers can drag files directly from the tree onto the canvas to add them as context nodes. The file tree also serves as an overview of which files are currently part of the context. When the developer clicks a context node, the system traverses every context node in the conversation chain and color-codes each referenced file in the tree. The colors reflect how the agent interacted with each file.

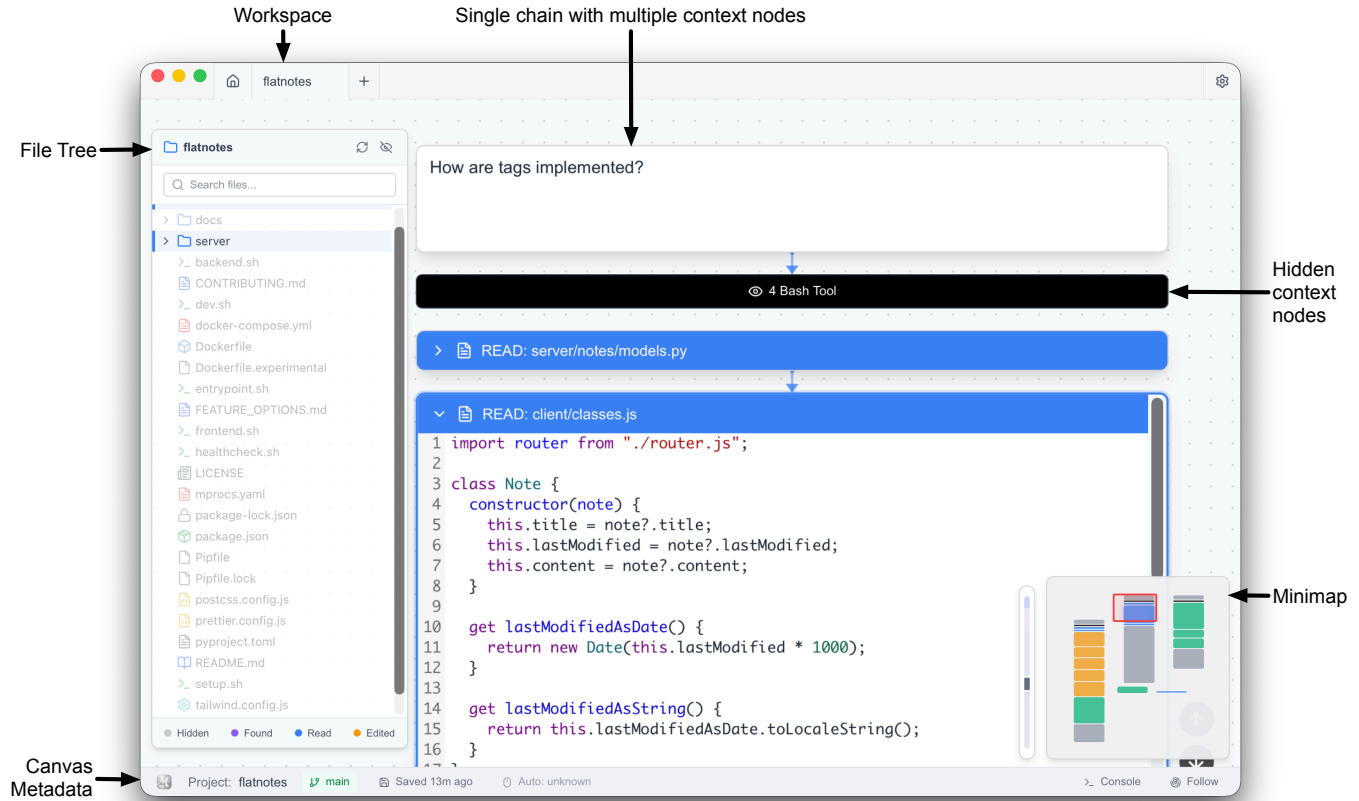


Figure 3. Project Canvas of Monet. An opened workspace within Monet. The canvas displays two parallel conversations. A minimap indicates the current viewport position. Zooming out expands the visible area, revealing more conversations and triggering semantic zoom.

Files pass through four states, ordered by increasing interaction. A file starts as *not found*. It becomes *discovered* once an exploration tool such as `ls`, `tree`, or `grep` has encountered it. It becomes *read* once its content has been retrieved. It becomes *written* once it has been modified. Only the highest applicable state is shown. A file that was both discovered and later read appears as read, and a file that was read and subsequently written appears as written. The color coding reveals how the agent interacted with each file. Gaps become visible where files were discovered but never read. Clicking an entry in the file tree highlights the corresponding node on the canvas and scrolls it into view. Figure 4 displays an instance of a file tree with color-coding.

Minimap When a canvas accumulates context nodes from multiple conversations, a reduced-scale overview of the full workspace becomes valuable. The minimap provides this overview, displaying all nodes and conversations on the canvas as an abstract layout. The color coding of context nodes is preserved in the minimap, allowing the developer to identify node categories at the reduced scale. It lets the developer orient

themselves within a large canvas, jump to specific regions, and see the extent of their workspace.

Semantic zoom Context nodes display the exact content that was used within a conversation. For large tasks or repositories, the number of context nodes can grow, making it impractical to inspect each node individually. Semantic zooming addresses this by replacing node content with summaries as the developer zooms out. At intermediate zoom levels, each context node is visually replaced by an LLM-generated summary describing the content of the context node. At further zoom levels, groups of summaries are themselves summarized, producing higher-level overviews. This layered representation lets developers navigate context histories by moving between different levels of abstraction, from broad summaries down to individual context nodes. Although these summaries are LLM-generated, they are only developer facing: semantic zoom changes what the developer sees, it does not change the input of the model since the underlying context nodes remain unaltered. A conceptual illustration is displayed in Figure 5.

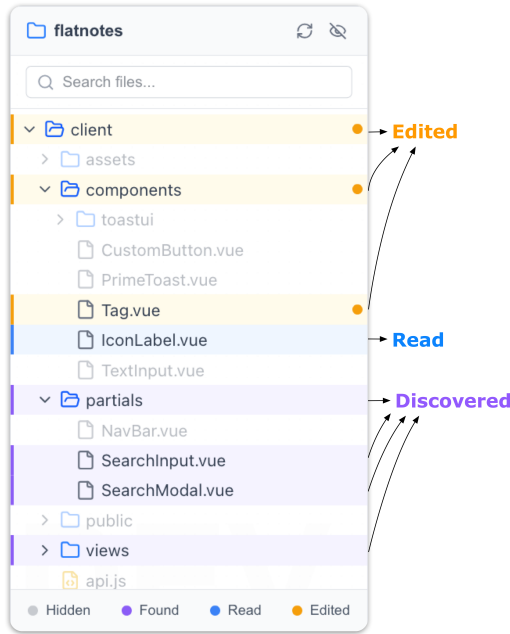


Figure 4. A project file tree with color-coded files indicates which files the agent discovered, read, or edited.



Figure 5. Conceptual figure of our implementation of semantic zoom at three levels. As the developer zooms out, individual context nodes are replaced by LLM-generated summaries, and groups of summaries are themselves summarized into higher-level overviews.

3.4 Interactions

The observability mechanisms described above let developers see the context and its current state. This section describes how developers can change it. Current tools require developers to describe their intent in natural language and hope the agent retrieves the right information. Monet lets developers steer generation by manipulating context nodes. This addresses both context degradation and the approaching context limit.

In a typical workflow in Monet, developers will start a conversation, have an agent generate context nodes, and then curate the generated context before continuing. Consequently, asking the agent to work more incrementally will give the developer more control, e.g., first asking it to build

up context with important code snippets, then asking it to formulate an implementation plan, and only then asking it to implement it. At each step, the developer can curate and steer the agent before it continues.

Drag-out: removing wrong context When the agent retrieves an irrelevant or misleading file, or when a tool call fails and returns an error, the result remains in the context window for the rest of the conversation in current coding agent systems. Similarly, exploration tools such as `ls` or `tree` can produce large outputs that consume large portions of the context window even when the information they contain is no longer needed. In the traditional terminal-based coding agent interface, the developer has little agency to remove any of this context. The only option is to start a new session, or to rewind the conversation to the point before the unwanted context was introduced, losing all subsequent context in the process. In Monet, the developer can drag a context node out of a conversation to remove it from that conversation’s context window. It is also reversible: the node remains on the canvas and can be dragged back in if the decision turns out to be wrong. If the developer is certain the node is not needed, they can also delete it from the canvas entirely. This interaction contrasts with compaction, where the system replaces context with a summary and the developer has no control over what information is kept or lost. Drag-out directly addresses the wrong-context failure mode (Section 2.3), allowing developers to correct retrieval mistakes as soon as they are noticed, rather than waiting for accumulated wrong context to cause generation errors.

Drag-in: injecting missing context Developers who are familiar with a codebase often know which files would be relevant if they were implementing the task themselves. However, they can only communicate this knowledge via a prompt. In current tools, the developer can include this information in the chat prompt or hope that mentioning it prompts the agent to search for it. In Monet, the developer drags a context node into the conversation. The node can be a file from the repository’s file tree or an artifact from a previous conversation. For example, if the agent fails to find a relevant configuration file, the developer can drag it from the file tree into the conversation chain. Similarly, context nodes from earlier conversations that are still visible on the canvas can be reused by dragging them into a chain. Drag-in addresses the missing-context failure mode (Section 2.3), where the agent cannot find information it does not know to look for. The developer bridges this gap by contributing their own knowledge of the codebase through a concrete action rather than through a verbal hint in the prompt.

Cross-conversation reuse As described in Section 3.2, context nodes persist on the canvas unless they are actively deleted, and can be moved or duplicated into new conversations. In current tools any context gathered in a previous session is lost, and the developer must either re-describe it or wait for the agent to re-derive it through another round of tool calls. This re-derivation is wasteful and unreliable: the agent may not find the same information a second time, or may find it in a different form [35]. Prompting the agent to persist context in scratchpad files offers a partial workaround. Cross-conversation reuse lets the developer carry forward the specific pieces of context they have already validated, rather than delegating the same retrieval task to the agent repeatedly. Because the context has already been retrieved and validated, the agent does not need to repeat tool calls to retrieve the same information, avoiding redundant token usage and computation.

Re-executing tools Tool results go stale as the conversation progresses. A file read at the start of a session may no longer reflect the current state of that file after the agent has made edits. In current tools, this stale information stays in the context window with no indication that it is outdated, and the agent continues to reason over it as if it were still accurate [43]. In Monet, each tool result node can be re-executed. Re-execution runs the same tool call again and replaces the node's content with the updated result. The node's identity and position on the canvas remain stable, so references from other nodes and conversations are preserved. Re-execution gives the developer a way to refresh context on demand, directly countering the stale-context failure mode (Section 2.3) without restarting the conversation.

3.5 From Canvas to Model Input

The canvas is the developer-facing representation of the entire context of multiple chains in a workspace. What the model receives as input is derived from it: a linear conversation history serialized from the canvas. The canvas itself is serialized as JSON. This representation retains the position of each node, the links between nodes, and the full content of every node. Each node on the canvas stores its original message. A prompt node stores the user message. A response node stores the model output. A tool node stores a call together with its result.

On every submission, Monet rebuilds the conversation history from the submitted chain. It reads the ordered nodes of a chain and passes them to the agent runtime as the full conversation history. As a result, the history contains no information about the canvas or the arrangement of nodes on it. An edited chain is submitted as a fresh conversation

rather than patched, which is what makes free reordering, removal, and injection possible.

Each curation interaction corresponds to a specific change in the rebuilt history. Removing a node drops its messages. A call and its result are serialized as one unit, so removing one removes both. The history, therefore, never holds a call without its result. A file dragged in from the file tree becomes a synthetic read tool call and its result. The history then looks the same as in a case where the agent reads the file itself, so the model treats the injected context the same way as the retrieved context. Re-executing a tool replaces the result content inside the same node. Because each edit produces a fresh conversation, it invalidates caches (Section 5).

4 Case Studies

To illustrate how developers can use Monet to identify and correct common context failures, we describe a series of scenarios drawn from real development sessions. Each scenario describes a concrete task, explains how the problem manifests in a conventional coding agent as described in Section 2.2, and shows how Monet's context observability and curation mechanisms change the developer's workflow. The scenarios cover the three categories of context failure: wrong context, missing context, and context limits and pollution due to accumulated stale information. The case studies assume that the developer has enough familiarity with the codebase to judge context quality. A developer who is new to a codebase can first run exploratory conversations on the canvas to map the structure before attempting curation.

Wrong and missing context. The following scenario emerged during the development of Monet itself, which was partly built using its own tooling. The application supports two runtime modes: a production mode that accesses the file system through the Tauri API (a framework for building desktop applications with web technologies), and a development mode that routes file operations through a dedicated file service to enable Chrome DevTools. A strategy adapter abstracts over both backends. When asked to implement a file-access feature, the agent read the project README, which describes the application as a Tauri project, and generated code that called the Tauri API directly, bypassing the adapter. This produced code that worked in production but broke the development environment. The scenario combines two failure modes: the agent retrieved *wrong context* (the README, which led it to use the Tauri API directly) and *missed context* (the strategy adapter, which it never read because the README already appeared to provide a sufficient answer).

In the **baseline**, the developer only discovered the wrong API usage by reviewing the generated code. The developer could not easily see which files the agent had consulted, making it impossible to tell whether the agent had found the adapter and ignored it, or never encountered it at all. After

identifying the problem, the developer reprompted the agent to use the adapter instead. The agent then re-read the adapter module and produced a new set of changes, which again required a full review. Each correction cycle was opaque and repetitive, with no guarantee that the agent would not reintroduce the same misleading context.

In **Monet**, the developer instructed the agent to gather context before generating code and then inspected the collected context on the canvas. The file tree color-codes files by interaction level (not found, discovered, read, written), making it apparent that the adapter had not been read. The developer navigated to the relevant context nodes in two ways. They either clicked on file tree entries or followed read operations via the color-coded canvas. Inline code previews let them verify what the agent had actually read. By dragging the adapter into the context chain, the developer could add the missing file. The canvas then visibly showed the corrected context chain, allowing the developer to confirm the correct abstraction was present before triggering code generation.

Missing context. In many web applications, a configuration is defined in one part of the codebase but consumed in another. In **Monet**, for instance, the settings for the AI backend are defined in a settings module, but a separate provider module in a different directory reads and applies them at runtime. When asked to modify how the backend configuration is handled, the agent found the settings definition and generated changes. It missed the provider module entirely, because it lives in a different directory and imports the configuration through a helper function rather than referencing the settings module directly.

In the **baseline**, the developer suspected the change was incomplete but could not confirm which files the agent had examined. The only option was to type follow-up prompts such as “did you check the provider?”, which triggered a new autonomous retrieval with no guarantee of finding the right file. The developer could not distinguish whether the agent had found the provider and dismissed it as irrelevant, or whether its search had never surfaced it at all.

In **Monet**, the file tree color-codes files by interaction level (not found, discovered, read, written). After the agent finished gathering context, the developer could see that the provider module was never read. The developer dragged the missing file from the file tree into the context chain, using the same interaction used for any context correction. The chain visibly grew with the new node, and resubmission produced changes that covered both the configuration definition and its consumption.

Context limit & pollution. In **Monet**, a service visually compacts context nodes based on grouping rules the user specifies in advance. During development of this feature, the developer asked the agent to fix a bug where grouped windows were not displayed correctly. The agent read the

service, tried an implementation approach, and generated code that did not resolve the issue. The developer pointed out the problem, and the agent tried again, reading additional files and producing new changes. After several failed attempts, the conversation contained multiple stale reads of the same file, failed diffs, and error output from earlier iterations. Eventually the context window filled up and the agent lost information through compaction. Outdated reads remained in the context alongside many code changes and exploration tool calls, leaving the agent to reason over conflicting contents.

In the **baseline**, the conversation is append-only. Besides rewinding the entire conversation to a specific point, the developer cannot remove failed attempts, refresh outdated reads, or selectively keep useful context. Each retry adds more noise, and the developer cannot tell whether a new error stems from bad reasoning or from the agent mixing up old and current file versions buried in the conversation history. When the context window eventually fills up, the system silently compacts earlier messages without notifying the developer. The developer continues prompting, unaware that relevant information was evicted, and errors caused by the missing context become undiagnosable. There is no way to see what was compacted, restore specific items, or choose what to evict. The only option is to start a new conversation, losing all accumulated context, including the parts that were still valid.

With **Monet**, the developer has two options that are not available in the baseline.

The first option is to free up context space without restarting the conversation or relying on compaction. The developer drags out stale reads of files that the agent has since modified. They also drag out large exploration nodes such as directory listings or grep results that consumed many tokens but are no longer relevant, as well as file reads which are not relevant to the problem. Only the relevant nodes remain, so the next prompt runs against a leaner context chain.

The second option is to reuse only the write nodes and start fresh. The developer duplicates the write tool calls from the failed attempts into a new branch and drags all other nodes out of the context chain. They add a prompt node stating that these writes represent approaches already tried, all of which produced the same bug. This effectively creates a new session that inherits the relevant history curated by the developer from the previous one, instructing the agent to explore a different direction rather than repeat the failed attempts.

5 Discussion and Future Work

Monet treats context as a first-class artifact of AI-assisted programming. Context is visible on the canvas, addressable as individual nodes, and editable through direct manipulation interactions. This positions the developer as a curator of

the agent’s context alongside the existing role of prompt author and reviewer. This framing raises questions about when curation applies, where its value lies, whether it stays relevant as models improve, and where its limitations are.

When curation applies. A developer can curate the context at two points in the AI-assisted programming workflow: before or after generation by the agent. Upfront curation keeps irrelevant or stale information out of the context window before generation begins. Irrelevant information in the context window has been shown to degrade generation quality [22, 27, 39, 46]. After generation, curation lets the developer act on the cause of a failure rather than describing the problem in a follow-up prompt. The developer can remove wrong context, add missing context, or refresh stale reads. Without curation, the developer retries blindly. They cannot tell which part of the context caused the failure, and they cannot learn how the agent retrieves from this repository.

Both forms of curation require the developer to judge what belongs in the context and what does not. That judgment presupposes familiarity with the codebase. A developer who is new to a repository may not know enough to judge whether the gathered context is correct or complete, while someone who knows the codebase well can spot gaps and fix them. Curation is likely most valuable for developers who know the codebase well enough to identify gaps in context retrieval but still benefit from the agent’s automation. Monet assumes that the developer has enough familiarity to judge context quality, which may not hold in onboarding scenarios. A possible mitigation is to let developers first run exploratory conversations on the canvas to build familiarity before curating context for implementation tasks.

Value of curation. Each curation action demands interaction effort, and that effort is a marginal cost paid per intervention. Whether curation is still valuable despite the additional effort depends on how value is defined. Notably, working in Monet does not exclude other workflows to manage context: manually listing files in a prompt, rewinding a linear conversation to an earlier point, or restarting with refined prompts are all still possible in Monet. Specifically, these workflows naturally map onto the chain concept rather than requiring special interactions. Consequently, for small tasks where curation adds little value as contexts remain naturally short, there is no inherent overhead to using traditional workflows.

Monet’s design suggests three dimensions in which the value of curation unfolds: cost, code quality, and developer understanding. In terms of cost, keeping a context trimmed close to the minimal amount of information that an agent requires may also reduce computational cost incurred. We discuss the trade-off to model caching below. Second, quality of the source code may be improved through active developer curation. Through tighter feedback loops in potentially smaller incremental steps, the developer can, for example, ensure that good reference examples are in the context for

the pending implementation, or that the agent makes adequate use of abstractions that are idiomatic for the code base. Further, as described in Section 2.3, a smaller context tends to produce better generation results as attention is focused on the relevant information. Third, developers may improve their understanding of the generated code through curation. In contrast to fully automated code generation, curating context forces the developer to interact and understand the agent’s reasoning as it is developing. Consequently, they understand how conclusions come to be, rather than having to later re-derive these from a code diff during a post-generation review.

Scaling does not eliminate curation. Two developments might suggest that manual context curation will soon be unnecessary: context windows are growing, and the underlying models are becoming more capable at retrieval. Some models already support context windows of one million tokens, reducing the need to drop information to stay within the limit. However, larger windows do not address the problems of wrong and missing context. More input tokens mean more information competes for the model’s attention as also discussed in the previous paragraph with regards to the quality of generated source code. Prior work has shown that model performance degrades as context grows, before the token limit is reached [11, 27]. Better retrieval capability does not eliminate the problem either. Recent benchmarking shows that even state-of-the-art models retrieve less than half of the code regions needed to resolve a task, and that elaborate retrieval frameworks do not reliably outperform simpler approaches [23]. Generation itself is non-deterministic and moving to newer models offers only marginal reductions in variance [35]. Even a perfect retrieval model still depends on the developer’s prompt to understand what to look for. An underspecified or ambiguous prompt leads to wrong or incomplete context, regardless of how capable the model is. The bottleneck is not only the model’s ability to find information but the developer’s ability to communicate intent. Curation addresses this from the other direction. The developer sees what the model retrieved from their articulated intent and corrects it directly, rather than refining the prompt and hoping for better retrieval.

Trade-off between curation and model caches. To optimize inference, model providers typically cache the conversation state with an agent [12]. Without the cache, a conversation with an agent has to be effectively replayed in its entirety to continue a conversation. As Monet allows for changing the conversation history, it has to invalidate the model cache as soon as a developer performs any change to past conversation history. In terms of performance, Monet thus presents a trade-off between the more optimized use of inference state through caching and a trimmed down, potentially shorter but hopefully better conversation history. Future work could investigate the effective cost in terms of

tokens that Monet vs. an immutable, cacheable conversation incurs. Our hope is that developers can and will trim down conversations to mitigate this issue to some extent but will notably also have to engage in shorter conversations with less re-work due to better conversation context.

Generalization beyond coding. The failure modes that motivate Monet are not specific to coding. Whenever an agent assembles information and produces an answer, the recipient might benefit from seeing the sources behind it. Writing assistants, research tools, and data analysis agents all gather context before generating output, and their users face the same opacity. In each case, the core problem is identical: the user cannot distinguish whether a wrong answer stems from bad reasoning or from bad input. Whether observability and curation should take the same form in those domains is open, but the underlying problem of opaque context during agentic retrieval is not unique to software development.

User study. Monet’s design has not yet been validated empirically with users, so our argument builds on the case studies, our own experience, and empirical evidence in prior work that shows that context degradation negatively affects the quality of generated code, providing initial evidence that tools for managing context are a worthwhile direction [22, 23, 26, 27, 39, 46]. A controlled user study could measure whether context observability and curation improve developers’ agency and affect task completion quality compared to chat-based coding agents. Further, a study could investigate if the context curation benefits novice and senior developers differently [4, 36]. Senior developers may curate more effectively because they know what context matters, while novices may use the canvas as an exploratory surface to build that understanding.

Observability and cognitive load in large repositories. Making context explicit introduces the risk of overwhelming the developer, especially for longer conversations where the number of context nodes grows. Full expressiveness is not the same as useful expressiveness: the developer needs to see enough to judge context quality without being forced to inspect every node. Monet’s semantic zoom and file tree address this by letting developers view context at different levels of detail, but the threshold between helpful observability and information overload is an open question. If developers spend more time managing context than writing code, the productivity gains that AI assistance provides would be undermined.

In very large repositories, where an agent will be exposed to a lot of information that is irrelevant to its task but which still has to be vetted by it to identify the useful information, the current facilities offered by Monet may no longer suffice: a conversation with too many nodes may not be handleable, even with semantic zoom.

We propose three directions of future work to target the challenges that arise with large repositories: automation, summary actions, and provenance visualization. To help with routine tasks, agent-based assistance could learn from developer behavior and suggest which context nodes to add, remove, or refresh based on observed curation patterns to help with a large volume of irrelevant reads in a large repository. Context templates could let developers save and reuse context setups for recurring tasks. Additionally, a second, cheaper model could inspect state on the user’s behalf to identify files that the main model pulled in for inspection but that yielded no valuable insights. The same model could act as a judge on context quality, checking whether the concepts a task touches are all covered and flagging gaps a developer cannot spot by reading a large repository unaided. At the same time, as automation for curation increases, there is a risk that users pay less attention and miss important retrieval failures that the automation did not catch either.

When the model did fetch helpful information but fetched a full file, where a single line of an API interface would have sufficed, future work could investigate summary actions. These would allow a developer to select a set of nodes and to ask a model to summarize these using preset prompts such as “reduce to a single line” or mechanical operations, such as “include only interface”, which can be performed using regular language tooling.

In terms of provenance, users could benefit from understanding how information that was fetched early during the generation history affect model decisions later on, for example, to allow removing a connected string of reasoning across non-adjacent context nodes. Such a graph view could thus show structural dependencies between nodes, making these implicit connections visible that remain hidden in the linear conversation timeline.

As an example, assume a larger feature addition to a web application. In this example scenario, the developer first asks the model to autonomously populate its context which files it deems necessary based on the task description. In this process, the model will likely fetch more information than is required by the task. As many of these reads will be tentative, just to discover what is even in the repository, the automation described above should be able to identify a number of opportunities to mechanically reduce the number of nodes. If the number of nodes is then still too large to make use of semantic zoom to gain an overview, switching to the structural graph view may, for example, reveal that a large compound of nodes was later discarded by the model as irrelevant. The developer can thus select the entire string of related nodes and delete them in one interaction. Once the model has completed a first version of the implementation, the model runs the implementation and encounters an error. After the model fixes the error, the developer makes use of the summary actions to select the consecutive nodes describing the program execution attempt and the problem

resolution and ask to summarize just the solution, as the changes and original error are not required for the model to proceed, replacing multiple hundreds lines of text including program output, errors, and multiple repair sites with just a single line of text. In particular, as the developer maintains short feedback loops in Monet, they maintain a good mental model of the agent’s decisions, which allows them to reduce the effort for reviewing code once the feature has been completed.

6 Related Work

Monet builds on prior work in alternative interaction surfaces for LLMs, developer agency in human-AI collaboration, task-specific views on programming artifacts, and human-in-the-loop machine learning.

Alternative interfaces for LLM interaction. Several systems replace the linear chat paradigm with spatial or structured surfaces. Node-and-wire interfaces let users compose LLM workflows visually, whether for prompt evaluation across models [3] or generative art [2]. Sensecape [40] targets open-ended information tasks such as trip planning and literature review: it places LLM responses as nodes on a zoomable canvas so users can organize findings hierarchically, seeing full text up close and summaries when zoomed out. Others let users click on a word or region and issue a prompt that transforms it in place, bypassing the chat input entirely [28]. One system moves closer to our concern regarding coding agents. BonsAIDE [31] visualizes AI-generated code alternatives as a branching graph, inspired by bonsai cultivation, that developers explore in parallel, compare, prune, and shape. The developer manages competing solutions simultaneously rather than committing to a single path. BonsAIDE operates on fixed input: the tool constructs each prompt automatically from the active file, task constraints, and the selected code state. In ReFiQ [29], developers receive multiple code proposals for each of their prompts to an AI and are shown the results of execution of each proposal first. This allows them to more quickly assess the quality of a proposal with regards to their domain, before engaging with the code. Autonomous context retrieval and its curation are outside the scope of these systems. They expand the interaction surface beyond the prompt, but do not address the context that a coding agent assembles autonomously before generation.

Developer agency in human-AI interaction. The need for such direct control is well documented in the broader human-AI interaction literature. Design guidelines for human-AI interaction identify transparency and controllability as central requirements [1]. A survey of 36 papers on in-IDE human-AI experience confirms that developer control and understanding of AI behavior remain open gaps [38]. Monet puts these guidelines into practice: context nodes make the

agent’s input visible, and drag-in and drag-out let the developer change it.

Task-specific views. The conversation history of an agent that the developer curates resembles a task-specific view on programming artifacts. The idea of task-specific views has been explored in various projects, including Mylar [19, 20], which establishes a degree-of-interest model for artifacts to reduce the burden of navigation on developers. In Code Bubbles [8], developers work on a canvas, similar to Monet, where source code is laid out based on relationships between artifacts, e.g., following references. The same can be done for debugging sessions, as seen for example in the Debugger Canvas [10] project. Patchworks [14] demonstrates a similar task-specific view on source code but constrains the layout to one dimension to reduce navigation effort.

Human-in-the-loop machine learning. The developer as curator model shares similarities with human-in-the-loop machine learning. In active learning, the system selects uncertain examples for human labeling. In interactive feature selection, users guide which signals the model attends to [32]. Context curation applies a similar principle at inference time: the developer corrects the input the model reasons over, not its learned parameters. Design lessons from interactive ML, such as surfacing uncertainty to guide human effort and thereby increase observability, may transfer to the curation setting. Whether they do is subject to future work.

7 Conclusion

The quality of code generated by coding agents depends on the context they receive, and the developer has little agency over what that context actually contains. Coding agents assemble this context themselves through autonomous tool calls, and this process is unreliable. They read the wrong files, miss the relevant ones, and keep reasoning over stale reads after their own edits have invalidated them. Roughly 79% of hallucinations in generated code trace back to these three failures rather than to flawed reasoning [26]. As the conversation grows, the context window fills up, and current management mechanisms such as compaction, session reset, and sub-agent delegation discard information without developer input. Current tools show tool calls and their results in an append-only log, which makes it hard for the developer to reconstruct what was gathered.

To address these failure modes, we argued that developers need observability into the context the agent works on and the agency to curate it. We presented Monet, a canvas-based development tool that implements both. On the canvas, each file read, tool call, and generated output becomes a node the developer can inspect, move, remove, or refresh. Context observability mechanisms make the gathered context visible and show its current state. Context curation interactions give the developer direct control over the gathered context. The

developer drags missing files in, drags wrong ones out, and re-executes stale reads without restarting the conversation.

This extends the developer's role beyond prompting and reviewing outputs. Whether this improves task outcomes requires empirical validation. If bad context retrieval is a major driver of generation failures, then the developer's most valuable contribution may be curating the input the agent reasons over, not just reviewing what it produces.

References

- [1] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. 2019. Guidelines for Human-AI Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3290605.3300233
- [2] Tyler Angert, Miroslav Suzara, Jenny Han, Christopher Pondoc, and Hariharan Subramonyam. 2023. Spellburst: A Node-based Interface for Exploratory Creative Coding with Natural Language Prompts. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*. Association for Computing Machinery, New York, NY, USA, 1–22. doi:10.1145/3586183.3606719
- [3] Ian Arawjo, Chelse Swoopes, Priyan Vaithilingam, Martin Wattenberg, and Elena L. Glassman. 2024. ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–18. doi:10.1145/3613904.3642016
- [4] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (April 2023), 85–111. doi:10.1145/3586030
- [5] 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)*. Association for Computing Machinery, New York, NY, USA, 15–24. doi:10.1145/3759534.3762682
- [6] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE '25)*. IEEE Press, Ottawa, Ontario, Canada, 2188–2200. doi:10.1109/ICSE55347.2025.00157
- [7] Islem Bouzenia and Michael Pradel. 2025. Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025, Seoul, Korea, Republic of, November 16–20, 2025*. IEEE, 2846–2857. doi:10.1109/ASE63991.2025.00234
- [8] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. Association for Computing Machinery, New York, NY, USA, 2503–2512. doi:10.1145/1753326.1753706
- [9] Shamse Tasnim Cynthia, Joy Krishan Das, and Banani Roy. 2026. Are We All Using Agents the Same Way? An Empirical Study of Core and Peripheral Developers Use of Coding Agents. arXiv:2601.20106 [cs.SE] doi:10.1145/3793302.3793377
- [10] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. 2012. Debugger Canvas: Industrial experience with the code bubbles paradigm. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Zurich, 1064–1073. doi:10.1109/ICSE.2012.6227113
- [11] 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. arXiv:2510.05381 [cs.CL] https://arxiv.org/abs/2510.05381
- [12] In Gim, Guojun Chen, Seung-Seob Lee, Nikhil Sarda, Anurag Khandelwal, and Lin Zhong. 2024. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13–16, 2024*, Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa (Eds.). mlsys.org. doi:10.48550/arXiv.2311.04934
- [13] Junda He, Christoph Treude, and David Lo. 2025. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead. arXiv:2404.04834 [cs] doi:10.48550/arXiv.2404.04834
- [14] Austin Z. Henley and Scott D. Fleming. 2014. The patchworks code editor: toward faster navigation with less code arranging and fewer navigation mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, Toronto Ontario Canada, 2511–2520. doi:10.1145/2556288.2557073
- [15] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiwu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7–11, 2024*. OpenReview.net.
- [16] 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
- [17] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7–11, 2024*. OpenReview.net.
- [18] Kelly Hong, Anton Troynikov, and Jeff Huber. 2025. Context Rot: How Increasing Input Tokens Impacts LLM Performance. Technical Report.
- [19] Mik Kersten and Gail C. Murphy. 2005. Mylar: a degree-of-interest model for IDEs. In *Proceedings of the 4th international conference on Aspect-oriented software development (AOSD '05)*. Association for Computing Machinery, New York, NY, USA, 159–168. doi:10.1145/1052898.1052912
- [20] Mik Kersten and Gail C. Murphy. 2006. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/1181775.1181777
- [21] 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 (July 2024), 81:1819–81:1840. doi:10.1145/3660788
- [22] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same Task, More Tokens: The Impact of Input Length on the Reasoning Performance of Large Language Models. 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, 15339–15353. doi:10.18653/V1/2024.ACL-LONG.818

- [23] Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. 2026. ContextBench: A Benchmark for Context Retrieval in Coding Agents. [doi:10.48550/ARXIV.2602.05892](https://doi.org/10.48550/ARXIV.2602.05892)
- [24] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14–20, 2024*. ACM, 52:1–52:13. [doi:10.1145/3597503.3608128](https://doi.org/10.1145/3597503.3608128)
- [25] Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. 2025. The Complexity Trap: Simple Observation Masking Is as Efficient as LLM Summarization for Agent Context Management. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*.
- [26] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and Evaluating Hallucinations in LLM-Powered Code Generation. [arXiv:2404.00971v1 \[cs\]](https://arxiv.org/abs/2404.00971v1) <https://arxiv.org/abs/2404.00971v2>
- [27] 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. *Trans. Assoc. Comput. Linguistics* 12 (2024), 157–173. [doi:10.1162/TACL_A_00638](https://doi.org/10.1162/TACL_A_00638)
- [28] Damien Masson, Sylvain Malacria, Géry Casiez, and Daniel Vogel. 2024. DirectGPT: A Direct Manipulation Interface to Interact with Large Language Models. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–16. [doi:10.1145/3613904.3642462](https://doi.org/10.1145/3613904.3642462)
- [29] Toni Mattis, Abdullatif Ghajar, Tom Beckmann, and Robert Hirschfeld. 2026. Live in the loop: rapid run-time feedback for prompts. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*. ACM, Barcelona Spain, 1–14. [doi:10.1145/3772318.3791731](https://doi.org/10.1145/3772318.3791731)
- [30] 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)*. Association for Computing Machinery, New York, NY, USA, 71–77. [doi:10.1145/3801119.3801131](https://doi.org/10.1145/3801119.3801131)
- [31] David Moreno-Lumbreras, Raula Gaikovina Kula, and Christoph Treude. 2026. BonsAIDE: An Extended Vision for Human-AI Interaction in IDEs. *ACM Trans. Softw. Eng. Methodol.* (Jan. 2026). [doi:10.1145/3793681](https://doi.org/10.1145/3793681)
- [32] Eduardo Mosqueira-Rey, Elena Hernández-Pereira, David Alonso-Ríos, José Bobes-Bascarán, and Ángel Fernández-Leal. 2022. Human-in-the-Loop Machine Learning: A State of the Art. *Artif. Intell. Rev.* 56, 4 (Aug. 2022), 3005–3054. [doi:10.1007/s10462-022-10246-w](https://doi.org/10.1007/s10462-022-10246-w)
- [33] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–16. [doi:10.1145/3613904.3641936](https://doi.org/10.1145/3613904.3641936)
- [34] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. [doi:10.1145/3597503.3639187](https://doi.org/10.1145/3597503.3639187)
- [35] Shuyin Ouyang, Jie M. Zhang, Mark Harman, and Meng Wang. 2025. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 42 (Jan. 2025), 28 pages. [doi:10.1145/3697010](https://doi.org/10.1145/3697010)
- [36] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A. Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The Widening Gap: The Benefits and Harms of Generative AI for Novice Programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research - Volume 1 (ICER '24, Vol. 1)*. Association for Computing Machinery, New York, NY, USA, 469–486. [doi:10.1145/3632620.3671116](https://doi.org/10.1145/3632620.3671116)
- [37] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative Agents for Software Development. 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, 15174–15186. [doi:10.18653/v1/2024.ACL-LONG.810](https://doi.org/10.18653/v1/2024.ACL-LONG.810)
- [38] Agnia Sergeyuk, Sergey Titov, and Maliheh Izadi. 2024. In-IDE Human-AI Experience in the Era of Large Language Models; A Literature Review. In *Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments, IDE 2024, Lisbon, Portugal, 20 April 2024*, Alexander Bezzubov, Danny Dig, Timofey Bryksin, and Yaroslav Golubev (Eds.). ACM, 95–100. [doi:10.1145/3643796.3648463](https://doi.org/10.1145/3643796.3648463)
- [39] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. 2023. Large Language Models Can Be Easily Distracted by Irrelevant Context. In *International Conference on Machine Learning, ICML 2023, 23–29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 31210–31227.
- [40] Sangho Suh, Bryan Min, Srishti Palani, and Haijun Xia. 2023. Sensecapse: Enabling Multilevel Exploration and Sensemaking with Large Language Models. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, UIST 2023, San Francisco, CA, USA, 29 October 2023–1 November 2023*, Sean Follmer, Jeff Han, Jürgen Steimle, and Nathalie Henry Riche (Eds.). ACM, 1:1–1:18. [doi:10.1145/3586183.3606756](https://doi.org/10.1145/3586183.3606756)
- [41] Ruotong Wang, Ruijia Cheng, Denae Ford, and Thomas Zimmermann. 2024. Investigating and Designing for Trust in AI-powered Code Generation Tools. In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency (FACCT '24)*. Association for Computing Machinery, New York, NY, USA, 1475–1493. [doi:10.1145/3630106.3658984](https://doi.org/10.1145/3630106.3658984)
- [42] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. [arXiv:2407.16741 \[cs.SE\]](https://arxiv.org/abs/2407.16741) [doi:10.48550/arXiv.2407.16741](https://doi.org/10.48550/arXiv.2407.16741)
- [43] Di Wu, Hongwei Wang, Wenhao Yu, Yuwei Zhang, Kai-Wei Chang, and Dong Yu. 2025. LongMemEval: Benchmarking Chat Assistants on Long-Term Interactive Memory. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24–28, 2025*. OpenReview.net.
- [44] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Demystifying LLM-Based Software Engineering Agents. *Proc. ACM Softw. Eng.* 2, FSE (June 2025), FSE037:801–FSE037:824. [doi:10.1145/3715754](https://doi.org/10.1145/3715754)
- [45] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10–15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). [doi:10.48550/arXiv.2405.15793](https://doi.org/10.48550/arXiv.2405.15793)

- [46] Minglai Yang, Ethan Huang, Liang Zhang, Mihai Surdeanu, William Yang Wang, and Liangming Pan. 2025. How Is LLM Reasoning Distracted by Irrelevant Context? An Analysis Using a Controlled Benchmark. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025, Suzhou, China, November 4-9, 2025*, Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (Eds.). Association for Computational Linguistics, 13329–13347. doi:10.18653/V1/2025.EMNLP-MAIN.674
- [47] Runlong Ye, Zeling Zhang, Boushra Almazroua, and Michael Liut. 2025. Beyond Autocomplete: Designing CopilotLens Towards Transparent and Explainable AI Coding Agents. (2025). doi:10.48550/ARXIV.2506.20062
- [48] Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA022 (June 2025), 23 pages. doi:10.1145/3728894

Received 2026-05-16; accepted 2026-06-23