Engineering Pitfalls in AI Coding Tools: An Empirical Study of Bugs in Claude Code, Codex, and Gemini CLI

Ruixin Zhang*
York University
Toronto, Canada
jason666@my.yorku.ca

Wuyang Dai*
York University
Toronto, Canada
ddai2002@my.yorku.ca

Hung Viet Pham
York University
Toronto, Canada
hvpham@yorku.ca

Gias Uddin
York University
Toronto, Canada
guddin@yorku.ca

Jinqiu Yang
Concordia University
Montreal, Canada
jinqiu.yang@concordia.ca

Song Wang
York University
Toronto, Canada
wangsong@yorku.ca

ABSTRACT

The rapid integration of Large Language Models (LLMs) into software development workflows has given rise to a new class of AI-assisted coding tools, such as Claude-Code, Codex, and Gemini CLIs. While promising significant productivity gains, the engineering process of building these tools, which sit at the complex intersection of traditional software engineering, AI system design, and human-computer interaction, is fraught with unique and poorly understood challenges.

This paper presents the first empirical study of engineering pitfalls in building such tools, on a systematic, manual analysis of over 3.8K publicly reported bugs in the open-source repositories of three AI coding tools (i.e., Claude-Code, Codex, and Gemini CLI) on GitHub. Specifically, we employ an open-coding methodology to manually examine the issue description, associated user discussions, and developer responses. Through this process, we categorize each bug in multiple dimensions, including bug type, bug location, root cause, and observed symptoms. This fine-grained annotation enables us to characterize common failure patterns and identify recurring engineering challenges.

Our results show that more than 67% of the bugs in these tools are related to functionality. In terms of root causes, 37.3% of the bugs stem from API, integration, or configuration errors. Consequently, the most commonly observed symptoms reported by users are API errors (18.3%), terminal problems (14%), and command failures (12.7%). These bugs predominantly affect the tool invocation (37.6%) and command execution (25%) stages of the system workflow. Collectively, our findings provide a critical roadmap for developers seeking to design the next generation of reliable and robust AI coding tools.

*Both authors contributed equally to this research.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

KEYWORDS

AI Coding, Engineering Pitfalls, Bug Characteristics

ACM Reference Format:
Ruixin Zhang, Wuyang Dai, Hung Viet Pham, Gias Uddin, Jinqiu Yang, and Song Wang. 2026. Engineering Pitfalls in AI Coding Tools: An Empirical Study of Bugs in Claude Code, Codex, and Gemini CLI. In *ACM International Conference on the Foundations of Software Engineering (FSE '26), June 5–9, 2026, Montreal, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The advent of powerful Large Language Models (LLMs) has fundamentally altered the software development landscape [4, 11, 23, 24]. Moving beyond simple code completion, tools such as Claude Code [1], Codex [2], and Gemini CLI [3] provide interactive, conversational interfaces capable of interpreting natural language instructions, performing task planning, invoking tools, maintaining session memory, modifying existing codebases, and executing system commands. These tools promise to minimize boilerplate code, expedite debugging workflows, and democratize programming by lowering technical barriers. However, transforming these systems from prototype demonstrations into production-ready, reliable developer tools requires substantial engineering effort.

Existing research has largely focused on evaluating the outputs of AI coding tools, such as the correctness, efficiency, and security of the code generated by these tools [7–9, 14, 17]. In contrast, the engineering of the tool layers that wrap, integrate, and operationalize these models has received far less attention. Building an effective AI coding tool is fundamentally a systems engineering challenge: it requires embedding unstable and non-deterministic AI components into otherwise deterministic software systems with stringent requirements for reliability, security, and user experience [18]. Despite the widespread enthusiasm for these tools, we still have limited empirical understanding of how well they are engineered in practice, how reliable they are, and what classes of bugs and failures they tend to exhibit.

To bridge this gap, in this work, we study three widely used AI coding tools, namely Claude Code [1], Codex [2], and Gemini CLI [3], as representative examples for investigating engineering defects that arise during the development and deployment of AI-assisted programming tools. These tools maintain publicly accessible issue trackers that contain rich, real-world bug reports submitted

by users while operating the tools in local development environments. The reported issues span a broad spectrum of engineering defects and reveal key challenges in building reliable AI coding systems, including functional errors, environment and dependency misconfigurations, and unexpected behaviors exhibited by the underlying LLMs. Collectively, these issues provide valuable empirical evidence for understanding the unique engineering challenges introduced by integrating LLMs into practical software development workflows. Specifically, we conduct the first systematic analysis of over 3.8K issues collected from the three AI coding tools that were closed by December 12, 2025, to answer the following research questions:

RQ1 (Bug Type): What types of bugs occur in these tools? This RQ categorizes the bugs found in the studied tools by their nature to characterize the overall defect landscape of AI coding tools.

RQ2 (Root Cause): What are the root causes of these bugs? This RQ explores the underlying causes of bugs, aiming to provide a deeper understanding of why they arise in AI coding tools.

RQ3 (Bug Symptom): What are the symptoms of these bugs? This RQ examines how bugs manifest to users at runtime, aiming to understand how underlying engineering defects translate into user-visible failures and affect developer workflows.

RQ4 (Bug Location): Which architectural layers do bugs occur in? This RQ examines how reported bugs are distributed across the six-layer architecture of a typical AI coding tool (Figure 1) to identify how different bug categories manifest throughout the system.

We found that most bugs in AI coding tools are related to core functionalities (i.e., functional bugs, 67%), while a subset of bugs reflects user concerns regarding ease of use (usability and UI bugs, 17.9%) and model compatibility (7.6%). The primary root causes of these issues are engineering-related, such as configuration and integration errors. These bugs manifest through multiple symptom types, with tool/API errors and command/terminal failures dominating the distribution. Consequently, they primarily impact the external tool orchestration and command execution layers of the system, highlighting critical points of vulnerability in AI coding workflows. By revealing the engineering pitfalls in existing AI coding tools, this study provides actionable insights for designing more reliable, usable, and robust AI coding tools. This work makes the following contributions:

- We conduct the first systematic empirical analysis of over 3.8K real-world issues collected from three widely used AI coding tools, i.e., Claude Code, Codex, and Gemini CLI, providing a comprehensive characterization of bugs encountered in practical AI-powered programming workflows.
- We propose a structured taxonomy that categorizes bugs according to their type, locations, observable symptoms, and root causes, offering a systematic framework for understanding defects in AI coding tools.
- We quantify the prevalence and impact of different bug categories across tools, highlighting systematic reliability weaknesses and tool-specific failure profiles.
- Based on our findings, we derive actionable implications for designing and testing AI coding tools.

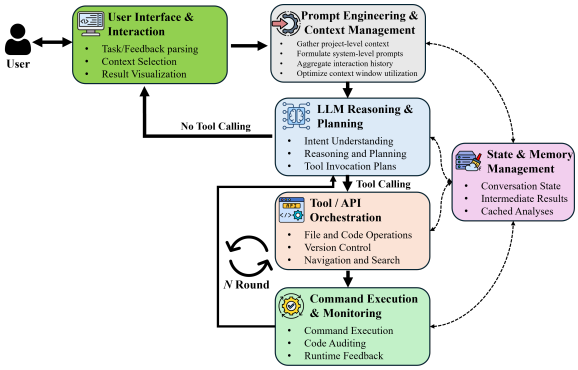

Figure 1: An Overview of AI Coding Tool

Data Availability: Our data are publicly available at https://zenodo.org/records/18342487.

The remainder of this paper is organized as follows. Section 2 provides the background for our study. Section 3 describes the design and setup of our empirical study. Section 4 presents the study results, while Section 5 discusses the key lessons learned. Section 6 outlines the potential threats to validity, and Section 7 concludes the paper.

2 BACKGROUND AND RELATED WORK

2.1 AI Coding Tools

AI coding tools, such as Claude Code [1], Codex [2], and Gemini CLI [3], represent a new paradigm in AI-assisted software development. These systems function as autonomous coding agents that integrate tightly with developers' workflows through command-line interfaces and IDE extensions. Despite differences in implementation, these systems share a common design paradigm: they leverage LLMs as their core reasoning engines to interpret natural-language commands, decompose high-level goals into executable steps, and coordinate tool usage to carry out multi-step programming tasks. By combining the reasoning capabilities of LLMs with program analysis, tool orchestration, and execution feedback, AI coding tools can perform tasks with limited human intervention while remaining responsive to developer input and oversight. We reviewed the online development documentation of the three tools and distilled their architectures into a common pattern, as illustrated in Figure 1. The system's architecture comprises six main distinct components that collectively enable sophisticated autonomous behavior, which are as follows.

2.1.1 User Interface & Interaction. This component serves as the primary interface between developers and the Claude Code system. It captures user commands, retrieves the current editor state (e.g., open files, cursor position, selected text), and displays results back to users through various UI components.

2.1.2 Prompt Engineering & Context Management. This component governs how the AI coding tool translates user intent into effective model instructions. It dynamically constructs prompts by combining user requests, relevant code snippets, project context, system directives, and tool metadata. Because LLMs are highly sensitive to context framing, this layer also handles context compression,

token budgeting, retrieval of relevant files, and conflict resolution between multiple sources of intent (e.g., user, system, agent).

2.1.3 LLM Reasoning & Planning. This component forms the foundation of AI coding tools and governs their interaction with the underlying LLM backends. It encompasses API invocation, streaming token handling, output parsing, rate limiting, and error handling, including retries and fallback strategies across different models or decoding configurations. Beyond basic communication, this component is responsible for orchestrating reasoning and planning behaviors, such as task decomposition, step ordering, and context management.

2.1.4 Tool / API Orchestration. AI coding assistants act as agents that rely on structured tool calls such as file editing, shell execution, code search, running tests, and interacting with version control to translate natural language directives into real system actions. The tool-invocation component defines the schema for these tools, interprets LLM-issued commands, validates parameters, and routes requests to the appropriate backend capabilities. It maintains a registry of all available tools and intelligently dispatches them based on the LLM's decisions.

2.1.5 Command Execution & Monitoring. This component provides the runtime environment in which code is compiled, executed, or tested. It includes interpreters, language runtimes, containerized sandboxes, build systems, dependency managers, and process-control mechanisms. The LLM relies on the correctness of this component to run commands, analyze outputs, and refine code.

2.1.6 State & Memory Management. This component maintains the state of ongoing conversations and tasks, implementing both short-term working memory and long-term persistence. It tracks what has been done, what needs to be done, which files have been modified, and what errors have occurred. It often also includes a prompt caching mechanism to reduce token usage by caching frequently accessed context like project structure and documentation.

2.2 IDE Design for AI-Powered Programming

Research on Integrated Development Environment (IDE) design has long examined how tooling, interaction paradigms, and information presentation affect developer productivity and experience. Early work by Wasserman et al. [25] introduced a graphical, extensible IDE architecture emphasizing modularity and customization, laying the groundwork for viewing IDEs as integrated platforms rather than simple editors. Subsequent studies explored IDE evaluation and adoption, highlighting context-dependent usability, developer expertise variability, and incremental evolution driven by workflow needs [10, 13].

Beyond feature-centric design, novel interaction paradigms such as Code Bubbles [6] demonstrated that spatially organized, task-focused interfaces can reduce cognitive load and improve navigation. With the rise of AI-assisted programming, IDE research has increasingly focused on human–AI interaction. Studies have identified challenges in trust, controllability, error handling, and understanding in natural-language-driven code generation [26], while recent surveys synthesized design patterns and evaluation gaps for LLM-integrated IDEs [19, 20, 22]. Parallel work links IDE design, developer experience, and productivity, emphasizing transparency, feedback, and cognitive support as key factors [15, 16], and exploring AI-driven automation in system architecture [5].

Table 1: The number of issues from each study tool

Tool	#Total Issues	#Closed Bugs	#Studied Bugs
Claude Code	14,804	3,536	2,343
Gemini CLI	7,988	618	329
Codex	3,883	1,897	1,192

3 EMPIRICAL STUDY SETUP

3.1 Studied AI Coding Tools

In this study, we examine Claude Code [1], Codex [2], and Gemini CLI [3] to investigate the engineering bugs that arise in modern agentic programming tools. These tools implement the common agentic workflow and are sufficiently complex to surface real-world engineering failures.

3.2 Data Collection

We collect the bugs reported by end users for each of the three tools from their publicly accessible GitHub repositories, ensuring transparency and reproducibility. The detailed steps are as follows.
Step 1: Automatic Issue Retrieval. For each repository, we used the GitHub REST API to retrieve the complete history of closed issues labeled as bugs. GitHub issues may represent different types of requests, such as new features, tests, debugging tasks, documentation, or bug reports; we therefore filtered issues by type to focus specifically on bugs. We considered only closed issues to ensure that each report included sufficient discussion, resolution context, or confirmation of a defect. For each issue, we collected and stored metadata including the title, description, labels, comments, timestamps, and resolution status for subsequent analysis.
Step 2: Manual Verification and Deduplication. Since labeling practices vary across repositories, we used label-based filtering only as an initial heuristic rather than a definitive criterion. We then manually inspected each remaining issue to ensure data quality. During this process, we removed: (1) issues marked as duplicates of earlier reports that described the same underlying problem; (2) deprecated issues associated with obsolete software versions, experimental features that were later abandoned, or components that had been officially discontinued; (3) test or mock issues intentionally created by developers for purposes such as debugging, regression testing, continuous integration validation, or demonstrating issue-tracking workflows; and (4) issues labeled as bugs that did not correspond to actual defects, including reports caused by user misuse or configuration errors. The remaining issues constitute our final bug dataset.

The first two authors were involved in the manual inspection process and conducted their assessments independently. In cases of disagreement, two additional authors participated in discussions to reach a consensus and finalize the decision. To quantify the reliability of the manual labeling process, we measured inter-rater agreement using Cohen's kappa, which yielded a value of 0.82,

indicating a high level of agreement. The number of bugs retained at each step of the filtering process is shown in Table 1.

3.3 Data Labeling

To construct a comprehensive taxonomy of issues in AI coding assistants, three of the authors work together employ an open-card sorting approach. This methodology allows categories to emerge inductively from the data rather than being imposed a priori.

We begin by randomly sampling issues from each of the three tools to construct initial categories. Based on standard sample size estimation [12], at a 95% confidence level with a 5% margin of error and an assumed population proportion of 20%, the representative sample sizes are 223 instances for Claude Code, 141 for Gemini, and 204 for Codex. We therefore select 200 issues per tool to (i) ensure an equal number of samples across tools for fair comparison and (ii) provide sufficient coverage for empirical characterization within each tool. This yields an initial corpus of 600 issues for subsequent analysis.

For each sampled issue, we manually examine the issue title, issue description, and associated comments to understand the reported problem, its manifestation, and the surrounding discussion. This manual analysis allows us to capture both surface-level symptoms and underlying engineering causes reflected in real-world usage. The three authors collaboratively developed the initial categories through thorough discussion and iterative refinement.

Based on this analysis, we iteratively constructed an initial taxonomy along four orthogonal dimensions: (1) Bug Type, describing the category of the defect; (2) Root Cause, capturing the underlying engineering or architectural reason for the issue; (3) Symptom, characterizing how the issue manifests from the user or system perspective; and (4) Bug Location, indicating the stage of the AI coding assistant (see Section 2.1) in which the issue occurs (e.g., user interaction, prompt engineering, LLM reasoning, tool orchestration, execution, or memory management). We then abstract recurring patterns across issues to derive high-level categories and, where appropriate, finer-grained subcategories within each dimension. This iterative process continues until the taxonomy stabilizes and no new major categories emerge within the sampled dataset.

After establishing the initial taxonomy, we apply it to label the remaining issues outside the initial sample. During this labeling phase, if the existing taxonomy cannot adequately capture an issue, we refine or extend the taxonomy by introducing new categories or subcategories.

4 RESULT ANALYSIS

In this section, we present and discuss our analysis results to address the four research questions we asked in Section 1.

4.1 RQ1: Bug Types

We categorize these bugs into five types, i.e., *Functionality Bug*, *Usability & UI Bug*, *Compatibility Bug*, *Performance Bug*, and *Security Bugs*. Table 2 shows their distributions across the three tools. Detailed definitions of each type, along with examples, are provided below.

Functionality Bug: is the bug that prevents one or more core capabilities from working as intended, leading to incorrect or incomplete task execution [21]. This category includes cases where a tool invocation, command, or model-assisted operation produces wrong outputs, fails to apply changes, or does not follow the expected workflow logic. Common examples include edit operations that fail to update files as requested, as well as file search or navigation features that return incorrect results or fail to execute. For example, in Codex issue #593, a user reported that Codex fails to execute the *apply_patch* command and instead merely displays the patch text. This is a functional bug since a core feature (i.e., automatic patch application) is not working as expected. The LLMs' inherent inaccuracies (generating wrong output or misunderstanding instructions) could not essentially be a bug, as it only shows the instability of the model's inner inference (prediction) logic, and it is rarely reported by users. However, if the AI coding tool consistently generates wrong outputs or misreads the instruction under some prompts with the same meaning, it could be a functional problem.

Usability & UI Bug: A Usability & UI bug impairs a user's ability to interact effectively with AI coding tools by introducing friction, ambiguity, or misleading feedback. This category includes interface-level issues such as flickering, misaligned layouts, missing elements, or incorrect UI states that prevent users from accurately interpreting or manipulating code suggestions. It also encompasses higher-level usability problems, including confusing workflows, unclear instructions, inconsistent prompts, or misleading documentation, which increase cognitive load and hinder task completion. Such bugs can lead to errors, reduce productivity, and influence navigation, ultimately reducing efficiency and overall user experience when interacting with these AI-assisted coding systems. For example, in Gemini issue #7016, users requested improved handling of content overflow in the terminal interface, replacing verbose truncated output (*... X lines truncated*) with robust visual scrollbars. The existing overflow behavior makes it difficult to read long outputs or navigate content efficiently, indicating a UI concern that affects readability and interaction.

Compatibility Bug: A compatibility bug arises from platform- or environment-specific constraints, where system behavior differs across operating systems, dependency versions, or terminal environments. Such bugs are characterized by features that function correctly in one environment but fail or behave differently in another. For example, a feature may fail on Windows due to version incompatibilities or exhibit incorrect input behavior under specific configurations. In Claude Code issue #300, a user experienced a "*node: bad option*" error when trying to run Claude Code on Ubuntu. This issue is caused by an unsupported Node.js version, which leads to a compatibility bug.

Performance Bug: A performance bug degrades efficiency in terms of latency, throughput, or resource usage, even if the tool

Table 2: Bug type distribution by model

Bug Type	Claude Code	Gemini	Codex CLI	Total	Rate (%)
Functional Bugs	1567	225	795	2587	67.0
Usability & UI Bugs	422	48	220	690	17.9
Compatibility Bugs	147	30	117	294	7.6
Performance Bugs	170	19	38	227	5.9
Security Bugs	37	7	22	66	1.7
Total	**2343**	**329**	**1192**	**3864**	**100.0**

still produces correct results. This category includes high computational costs, redundant operations, unintended recursion, and high memory consumption that significantly slow or crash workflows. In Claude Code issue #336, a user reported a crash due to an OutOfMemory error when running Claude Code in a specific codebase. This is a classic performance issue that crashes the system.

Security Vulnerability: A security vulnerability introduces vulnerabilities or unsafe behaviors that may enable unauthorized access, data exposure, or unintended destructive actions. This includes improper path or workspace validation (e.g., bypass via filesystem constructs), insecure handling of authentication state, and tool behaviors that execute sensitive operations without adequate safeguards. For example, a workspace restriction can be bypassed to access unintended files, or a destructive repository operation is performed without proper permission confirmation. Moreover, in Claude Code issue #263, users reported that running or updating the claude CLI as the superuser (*root*) poses a security risk. Specifically, the tool's installation and authentication instructions encourage operations requiring elevated privileges (e.g., *chmod -R* or *chown -R*), which can destabilize systems or expose critical files when executed with root permissions.

According to Table 2, bugs related to functionality account for the largest proportion (67%), indicating that many reported issues involve feature failures or incorrect/unexpected outputs during usage. Usability & UI and compatibility bugs represent the second and third largest categories (17.9% and 7.6%, respectively), suggesting that users often encounter interaction problems that require extra steps, cause confusion in workflows, or raise concerns about stability and consistent behavior. Together, these three categories make up approximately 85% of all bug reports. Besides those frequent bugs, security bugs appear the least (only 1.7%), but they might lead to serious problems such as private data leakage.

Answer to RQ1: Functional bugs are the most frequent and directly impede users from using the tools. While usability & UI, compatibility, and performance bugs occur less often, they still impact the user experience. Security vulnerabilities are the least common, but their potential consequences are severe.

4.2 RQ2: Root Cause

To systematically analyze the underlying reasons behind observed failures, we construct a root cause taxonomy for the issues studied. Each root cause was determined via evidence-based manual inspection: for each issue, we reviewed the title, description, and full discussion thread and extracted concrete evidence such as error logs/stack traces, command outputs and exit codes, and reproduction steps. As illustrated in Table. 3, the taxonomy organizes failures into high-level root cause categories and finer-grained subcategories. The taxonomy of root causes is categorized as follows:

API & Integration Errors: This root cause category covers failures that occur at integration points, where the system interacts with external APIs or third-party components. Subcategories include mismatches in API structure or validation logic, integration incompatibilities, network or proxy constraints, improper handling of authentication keys, and malformed or unexpected inputs exchanged between systems.

Configuration & Setup: Root causes in this category stem from incorrect, incomplete, or inconsistent configuration during system setup or maintenance. Subcategories include problems related to environment variables, directory structures and paths, installation or update processes, tool or configuration file settings, as well as documentation that leads users to configure the system incorrectly.

User Interaction & UI: Root causes in this category arise from incorrect assumptions about user interaction and interface behavior. Subcategories include improper handling of input and output, unsupported or conflicting keyboard shortcuts, incorrect UI rendering or display states, unsafe clipboard interactions, and keybinding conflicts that disrupt expected workflows.

Environment & Platform Compatibility: This category arises from mismatches between system and execution environment, including differences in shells, operating systems, runtime versions, terminal emulators, and file system semantics.

AI Logic & Behavior: This category captures root causes intrinsic to AI-driven components and model behavior. Subcategories include systematically incorrect outputs, hallucinated assumptions, unintended effects of content filtering, failure to follow instructions, and incorrect assumptions about available models or subscription-level capabilities.

Performance & Resource Limits: Root causes in this category arise when a workflow unexpectedly exceeds runtime-, configuration-, or platform-imposed resource limits, preventing tasks from completing. Subcategories include exceeding execution time limits (timeouts), exhausting memory or context budgets, and pathological computation patterns (e.g., non-terminating loops or unbounded recursion) that lead to resource exhaustion and prevent tasks from completing.

Incorrect Command Execution & Parsing: This category encompasses issues stemming from errors in command construction, parsing, or execution. Subcategories include incorrect handling of Bash or CLI syntax, command parsing failures, improper argument formatting, flawed process control, and logical conflicts between command arguments.

Improper State & Context Management: Root causes in this category arise from improper handling of session state, conversational context, or persistent memory. Subcategories include loss of session persistence, missing or truncated conversation history, errors during context compaction, incorrect instruction or memory handling, and unintended state resets or corruption.

Code & File Handling: Root causes in this category relate to how source code and files are processed, modified, or managed by the system. Subcategories include inconsistencies in line endings or character encoding, whitespace or tab handling errors, incorrect tracking of file state, and limitations or failures when processing large files.

Permissions & Security: This category arises from improper enforcement or usage of permission and security mechanisms. Subcategories include insufficient or overly restrictive access rights, misuse of elevated privileges, platform-imposed security restrictions, unsafe handling of sensitive data, and incorrectly scoped service roles or permission models.

Table 4 presents the aggregated distribution of root causes of the studied bugs grouped by high-level root cause categories, with counts shown separately for each tool. The primary root causes of

Table 3: Root cause taxonomy for observed failures (categories, sub-categories, and descriptions).

Category & Sub-categories	Description	No.	Rate
API & Integration Errors	Failures occur at system integration boundaries when interacting with external APIs or services.	**826**	**21.4%**
• API Structure / Validation	APIs reject requests due to schema mismatches or invalid parameters.	325	8.4%
• Key & Authentication Handling	Authentication tokens or API keys are missing, expired, or mis-scoped.	242	6.3%
• Network / Proxy / VPN Issues	Requests fail due to network routing, proxy, or VPN constraints.	103	2.7%
• Third-Party Integration	Third-party services behave inconsistently or are incompatible.	82	2.1%
• Malformed Input Handling	Malformed or unexpected inputs cause request rejection or undefined behavior.	74	1.9%
Configuration & Setup	Failures caused by incorrect, incomplete, or inconsistent system configuration.	**613**	**15.9%**
• Misconfigured Tools / Files	Tools or configuration files are misconfigured or incompatible.	186	4.8%
• Path & Directory Problems	Incorrect directory layouts or path resolution lead to runtime errors.	132	3.4%
• Environment Variable Issues	Required environment variables are missing or mis-set.	127	3.3%
• Installation & Update Problems	Installation, upgrade, or rollback processes fail.	104	2.7%
• Documentation Errors	Documentation inaccuracies cause incorrect user setup.	64	1.6%
User Interaction & UI	Failures arising from incorrect assumptions about user interaction or interface behavior.	**430**	**11.1%**
• Input / Output Handling	User inputs or outputs are mishandled or misinterpreted.	190	4.9%
• UI Rendering / Display	UI components render incorrectly or inconsistently.	145	3.8%
• Keyboard Shortcuts	Keyboard shortcuts do not function as expected.	63	1.6%
• Keybinding Conflicts	Conflicting keybindings disrupt workflows.	25	0.7%
• Clipboard & Data Leak	Clipboard usage leads to unintended data exposure.	7	0.2%
Environment & Platform Compatibility	Failures due to mismatches between system assumptions and execution environments.	**404**	**10.5%**
• OS / Platform Restrictions	Platform-specific constraints prevent correct execution.	154	4.0%
• Shell Incompatibility	Shell differences cause command failures.	118	3.1%
• Terminal Emulator Issues	Terminal emulators behave inconsistently.	64	1.7%
• Node Version Issues	Incompatible runtime versions cause errors.	34	0.9%
• File System Differences	File system semantics differ across environments.	34	0.9%
AI Logic & Behavior	Failures intrinsic to model reasoning or behavioral constraints.	**385**	**10.0%**
• Instruction Noncompliance	The model fails to follow explicit instructions.	145	3.8%
• Incorrect Output	Generated outputs are logically incorrect or misleading.	122	3.1%
• Model / Subscription Recognition	Model or subscription capabilities are misidentified.	80	2.1%
• Feature Hallucination	Hallucinated features or APIs are referenced.	22	0.6%
• Content Filtering	Content filtering interferes with task completion.	16	0.4%
Performance & Resource Limits	Failures caused by exceeding practical runtime or resource constraints.	**313**	**8.0%**
• Memory / Context Exceeded	Context windows or memory limits are exceeded.	163	4.1%
• Timeouts	Tasks terminate due to execution timeouts.	121	3.1%
• Infinite Loops / Recursion	Unintended infinite loops or recursion prevent completion.	29	0.8%
Command Execution & Parsing	Failures during command construction, parsing, or execution.	**262**	**6.8%**
• Process Control	Process lifecycle or control is mishandled.	95	2.5%
• Command Parsing	Commands are parsed incorrectly.	63	1.6%
• Argument Formatting	Arguments are malformed or incompatible.	45	1.2%
• Argument Conflict / Logic	Logical conflicts occur between command arguments.	36	0.9%
• Bash / CLI Syntax Handling	Shell syntax is not handled correctly.	23	0.6%
State & Context Management	Failures related to incorrect handling of session state or conversational context.	**253**	**6.6%**
• Session Persistence Issues	Sessions fail to persist across interactions.	99	2.6%
• State Reset / Corruption	Internal state resets or becomes corrupted.	72	1.9%
• Context Compaction Errors	Context compaction removes critical information.	38	1.0%
• Instruction / Memory Handling	Instruction or memory handling is flawed.	26	0.7%
• Conversation History Loss	Conversation history is lost unexpectedly.	18	0.5%
Code & File Handling	Failures caused by incorrect processing or manipulation of code and files.	**218**	**5.6%**
• File State Management	File state changes are tracked incorrectly.	137	3.5%
• Line Endings & Encoding	Encoding or line-ending mismatches occur.	51	1.3%
• Large File Issues	Large files exceed handling capabilities.	17	0.4%
• Whitespace / Tab Handling	Whitespace or tab inconsistencies cause errors.	13	0.3%
Permissions & Security	Failures related to improper enforcement or use of security mechanisms.	**160**	**4.1%**
• Insufficient Permissions	Operations fail due to insufficient privileges.	85	2.2%
• Security Restrictions	Platform security restrictions block actions.	48	1.2%
• Sensitive Data Handling	Sensitive data is mishandled.	12	0.3%
• Superuser Misuse	Elevated privileges are misused.	8	0.2%
• Service Role Permissions	Service role permissions are incorrectly scoped.	7	0.2%

Table 4: Root cause category distribution by model.

Root Cause Category	Claude Code	Gemini	Codex CLI	Total	Rate (%)
API & Integration Errors	495	49	282	826	21.4
Configuration & Setup	437	41	135	613	15.9
User Interaction & UI	232	61	137	430	11.1
Environment & Platform Compatibility	231	43	130	404	10.5
AI Logic & Behavior	159	43	183	385	10.0
Performance & Resource Limits	200	21	92	313	8.1
Command Execution & Parsing	180	18	64	262	6.8
State & Context Management	171	17	65	253	6.5
Code & File Handling	145	26	47	218	5.6
Permissions & Security	93	10	57	160	4.1
Total	**2343**	**329**	**1192**	**3864**	**100.0**

bugs are dominated by *API & Integration Errors* and *Configuration & Setup*, which account for 21.4% and 15.9% of all the bugs observed, respectively. These categories underscore persistent challenges at integration boundaries and in environment configuration across all three tools. Other notable causes include *User Interaction & UI* and *Environment & Platform Compatibility*, which account for 11.1% and 10.5% of all the bugs observed, respectively, reflecting issues stemming from assumptions about user workflows and execution contexts. Bugs attributed to *AI Logic & Behavior* account for only 10% of all observed bugs; this category of root causes indicates intrinsic limitations of model reasoning and constraint adherence, though they are not the predominant type. Less frequent, but still significant, are *Permissions & Security* and *Code & File Handling*, which exhibit model-dependent variation.

Answer to RQ2: Our root cause analysis shows that bugs primarily originate from systemic engineering issues. Such as configuration and setup errors, API and integration mismatches, and environment or platform incompatibilities, rather than from isolated coding mistakes or LLM reasoning errors.

4.3 RQ3: Bug Symptom

This question investigates how bugs manifest to users during runtime. By categorizing these observable symptoms, we aim to understand how underlying engineering defects translate into user-facing failures and affect developer workflows.

We identified 16 categories of bug symptoms from the observed bugs, each comprising several subcategories. Details of these categories are provided in Table 5. Below, we summarize the symptom taxonomy and describe the typical user-visible behaviors associated with each type of issue.

API & Server Communication. Symptoms in this category are primarily manifested as failures in communicating with APIs, where requests are rejected, delayed, or return unexpected errors. Subcategories include: 1) *API Errors & Status Codes*: explicit API error messages with status codes (e.g., 4xx/5xx) indicating request failures, 2) *Connection/Timeout Issues*: requests time out, stall, or the server appears non-responsive, 3) *Invalid/Blocked Requests*: requests are denied due to some reasons, 4) *Rate Limits & Quotas*: users hit usage limits and receive warnings, 5) *API Schema & Contract Errors*: mismatches between expected and actual request or response formats of APIs.

CLI/Terminal Experience. This category describes user-visible issues in terminal-based interactions, where the interface behaves unexpectedly or input/output is not rendered reliably. Subcategories include: 1) *UI Rendering Problems*: terminal UI is misaligned, flickers, or displays corrupted layout, 2) *Input & Keybinding Issues*: keystrokes (e.g., arrows, shortcuts) are ignored or trigger wrong behavior, 3) *Terminal Output Errors*: output is truncated or missing, 4) *Autocomplete & Reference Issues*: suggestions are incorrect, incomplete, or reference wrong files/commands, 5) *IME/Locale-Related Input Issues*: non-English input methods cause broken typing, composition, or command entry.

Command Execution. Symptoms in this category occur when the system runs commands on the user's behalf, and execution does not proceed as expected, often preventing task completion. Subcategories include: 1) *Timeouts & Hanging*: commands stall indefinitely or exceed expected runtime, 2) *Shell & Terminal Bugs*: shell-specific behaviors (path parsing, etc.) lead to failures, 3) *Unsupported Commands*: attempted commands are unavailable or not permitted, 4) *Interrupted Sessions*: execution is terminated mid-run, 5) *Autocomplete & Keybinding Failures*: command-line completion or shortcut-driven control fails during execution workflows.

Code & File Operations. This category captures symptoms where code editing, file manipulation, or repository operations behave incorrectly, leading to missing changes or inconsistent project state. It includes: 1) *File Editing & Replacement*: edits are not applied, partially applied, or overwrite the wrong content, 2) *File/Folder Structure*: incorrect paths, missing directories, or unexpected file placements, 3) *Context Management*: the system uses the wrong context or loses necessary context during editing, 4) *Search & Indexing*: file search is inaccurate, incomplete, or returns irrelevant results, 5) *Symlink & Symbolic Link Handling*: operations fail or behave unexpectedly when symlinks are present.

AI Model & Output Control. Symptoms here relate to the model's behavioral output rather than infrastructure failures, including unsafe, misleading, or non-actionable responses. Subcategories include: 1) *Harmful Content Generation*: outputs violate safety expectations or generate content that is not allowed, 2) *Overly Affirmative/Unintended Responses*: the model agrees incorrectly or produces unintended responses, 3) *Unauthorized Modifications*: the system performs changes beyond what the user requested, 4) *Misleading/Incorrect AI Output*: generated guidance or code is incorrect in a way that misleads the user, 5) *Simulated/Non-Functional Execution*: the model claims actions were performed but actually not, or produces hallucinated results.

Authentication & Access. This category reflects symptoms in identity, credential, and permission workflows that prevent users from accessing features or resources. Subcategories include: 1) *OAuth & Login Errors*: login redirects fail, tokens are not issued, or sign-in loops occur; 2) *API Key Issues*: keys are missing, rejected, invalid, or not recognized; 3) *Permissions/Authorization*: requests fail due to insufficient privileges or incorrect authorization scope.

User Feedback & Usability. Symptoms in this category reduce user confidence and efficiency by providing unclear, incomplete, or misleading guidance about system status and required actions. Subcategories are: 1) *Misleading Output*: messages imply success/failure incorrectly or provide confusing messages, 2) *Documentation Gaps*: missing, outdated, or ambiguous instructions that block progress, 3) *Logging & Notifications*: logs are absent, or do not show actionable error details, 4) *Keyboard Shortcuts & Keybindings Issues*: shortcuts are inconsistent or not functional, increasing interaction cost, 5) *Focus & Multitasking Problems*: context switching is fragile (e.g., losing focus, difficulty managing multiple tasks).

Session & State Management. This category describes failures to preserve or restore ongoing work, leading to lost progress or broken continuation. Subcategories include: 1) *Session Persistence*: sessions are not saved or restored, 2) *Unsaved Progress & Task Resumption*: unfinished work cannot be resumed from a prior session, 3) *Task & Agent State Loss*: internal agent state resets unexpectedly, 4) *Context Continuity & Restoration*: prior context is not carried over, causing inconsistent follow-up behavior, 5) *Session Timeouts & Expiry*: sessions expire too early that interrupt workflows.

Configuration & State. Symptoms here arise when configuration settings or runtime state are applied inconsistently, resulting in repeated failures or unstable behavior across runs. Include: 1) *Settings Persistence*: user settings do not persist, revert, or apply inconsistently, 2) *Env Variables & Paths*: environment variables or path resolution are incorrect or missing, 3) *Update/Restart Loops*: repeated update prompts or restart cycles, 4) *State Persistence Issues*: internal state is not correctly stored or recovered, 5) *Configuration File Bloat*: config artifacts grow unexpectedly that become corrupted or degrade usability.

Installation & Setup. This category captures symptoms during initial setup or environment preparation, where prerequisites fail and prevent the system from running correctly. Subcategories are: 1) *Platform Compatibility*: installation fails or behaves differently across operating system or architecture, 2) *Permissions & Privileges*: insufficient permissions block installs, tool access, or required operations, 3) *Package Management*: dependency installation fails, conflicts occur, or versions mismatch, 4) *Shell/Environment Issues*:

Table 5: Symptom categories and sub-categories

ID	Category-Subcategory	Description	No.	Rate
1	**API & Server Communication**	Remote requests fail or return unexpected server-side errors during interaction with backend	**708**	**18.3%**
	• API Errors & Status Codes	services.	403	10.4%
	• Connection/Timeout Issues	The system returns explicit API failures with status codes.	124	3.2%
	• Invalid/Blocked Requests	Requests stall or time out before the service responds.	37	0.9%
	• Rate Limits & Quotas	Requests are rejected due to validation errors, policy blocks, or forbidden endpoints.	90	2.3%
	• API Schema & Contract Errors	Operations are terminated after exceeding request quotas or usage limits.	54	1.4%
		Request or response fields do not match the expected API schema.		
2	**CLI/Terminal Experience**	Terminal-based interaction behaves unexpectedly, making input, output, or navigation unreliable.	**542**	**14.0%**
	• UI Rendering Problems	The terminal UI renders incorrectly, with layout glitches or flicker.	197	5.1%
	• Input & Keybinding Issues	Key presses and shortcuts are ignored or mapped to unintended actions.	188	4.9%
	• Terminal Output Errors	Terminal output is missing, truncated, or corrupted.	66	1.7%
	• Autocomplete & Reference Issues	Autocomplete suggestions or references point to incorrect or incomplete contents.	51	1.3%
	• IME/Locale-Related Input Issues	Locale or IME composition breaks command entry or text input.	40	1.0%
3	**Command Execution**	Commands executed on behalf of users do not run to completion or behave incorrectly.	**492**	**12.7%**
	• Timeouts & Hanging	Invoked commands hang or exceed expected execution time.	216	5.6%
	• Shell & Terminal Bugs	Shell or terminal issues that cause command failures.	106	2.7%
	• Unsupported Commands	Required commands are unavailable or not allowed in the current environment.	125	3.2%
	• Interrupted Sessions	Execution is terminated during running.	37	0.9%
	• Autocomplete & Keybinding Failures	Completion or shortcut-driven control fails within execution workflows.	8	0.2%
4	**Code & File Operations**	Editing project files produces missing, misplaced, or inconsistent changes.	**281**	**7.3%**
	• File Editing & Replacement	Edits are not applied correctly or modify the wrong files.	121	3.1%
	• File/Folder Structure	Files or directories are created, moved, or referenced at incorrect paths.	62	1.6%
	• Context Management	The system applies changes using the wrong file context or loses needed context.	52	1.4%
	• Search & Indexing	Search results are incomplete, irrelevant, or miss the intended files.	31	0.8%
	• Symlink & Symbolic Link Handling	Operations fail or misbehave when paths traverse symbolic links.	14	0.4%
5	**AI Model & Output Control**	Model issues that confuse users or derail task completion.	**271**	**7.0%**
	• Harmful Content Generation	Generated content violates safety expectations or includes disallowed contents.	7	0.2%
	• Overly Affirmative/Unintended Responses	The model produces inappropriate agreement or unintended behavior.	17	0.4%
	• Unauthorized Modifications	Actions are taken beyond the user's stated intent or scope.	38	1.0%
	• Misleading/Incorrect AI Output	Advice or generated code is incorrect in ways that take users toward failure.	164	4.2%
	• Simulated/Non-Functional Execution	The system says actions were executed but actually not.	45	1.2%
6	**Authentication & Access**	Login, credential, or permission flows fail, preventing access to required features or resources.	**258**	**6.7%**
	• OAuth & Login Errors	OAuth flows fail with redirects, loops, or missing tokens.	143	3.7%
	• API Key Issues	API keys are missing, invalid, or rejected by the system.	59	1.5%
	• Permissions/Authorization	Requests fail due to insufficient privileges or incorrect authorization scope.	56	1.5%
7	**User Feedback & Usability**	System messages and guidance are unclear or misleading, increasing user effort.	**243**	**6.2%**
	• Misleading Output	Status messages report is inconsistent with actual outcomes.	151	3.9%
	• Documentation Gaps	Help text or documentation omits information needed to proceed correctly.	38	1.0%
	• Logging & Notifications	Logs or notifications are absent or noisy.	27	0.7%
	• Keyboard Shortcuts & Keybindings Issues	Global shortcuts behave inconsistently, increasing interaction cost.	12	0.3%
	• Focus & Multitasking Problems	Focus management breaks when users switch tasks or contexts.	15	0.4%
8	**Session & State Management**	Sessions fail to persist, resume, or maintain consistent internal state across interactions.	**224**	**5.8%**
	• Session Persistence	Sessions are not saved or restored.	62	1.6%
	• Unsaved Progress & Task Resumption	Sessions that in progress can not be resumed from a previous checkpoint.	14	0.4%
	• Task & Agent State Loss	Agent state resets unexpectedly, losing progress or decisions.	71	1.9%
	• Context Continuity & Restoration	Prior context is not carried forward, causing inconsistent follow up behavior.	65	1.7%
	• Session Timeouts & Expiry	Sessions expire or timeout prematurely.	12	0.3%
9	**Configuration & State**	Configurations are applied inconsistently, causing repeated failures across runs.	**209**	**5.4%**
	• Settings Persistence	User settings revert or fail to take effect consistently.	74	1.9%
	• Env Variables & Paths	Environment variables or path resolution are missing or incorrect.	77	2.0%
	• Update/Restart Loops	Repeated updates or restarts prevent stable use of the system.	28	0.7%
	• State Persistence Issues	Internal state is not persisted correctly between runs.	11	0.3%
	• Configuration File Bloat	Configuration artifacts grow or corrupt, degrading stability or usability.	19	0.5%
10	**Installation & Setup**	Setup steps fail during environment preparation, blocking initial use or correct operation.	**166**	**4.3%**
	• Platform Compatibility	Behavior differs across OS/architecture, causing setup or runtime failures.	42	1.1%
	• Permissions & Privileges	Operations fail due to missing privileges or restricted access.	34	0.9%
	• Package Management	Package installation fails due to conflicts or version mismatches.	52	1.3%
	• Shell/Environment Issues	Shell or environment configuration prevents correct execution.	38	1.0%
11	**Security & Data Management**	Security handling is flawed, risking exposure, unauthorized actions, or incorrect access control.	**128**	**3.3%**
	• Credential Leaks	Secrets appear in outputs or are stored in insecure locations.	8	0.2%
	• Silent/Unauthorized Actions	Actions occur without authorization.	41	1.1%
	• File/Directory Access Control	Restricted files or directories can be accessed when they should not be.	33	0.9%
	• Keychain & Credential Store Issues	Secure credential stores fail to read, write, or unlock secrets.	46	1.2%
12	**Integration & Compatibility**	Interactions with external tools break due to integration-specific constraints.	**116**	**3.0%**
	• Third-party Tool Support	External tools cannot be invoked or coordinated correctly.	24	0.6%
	• Editor Integration	IDE/editor workflows break due to integration defects.	39	1.0%
	• Dev Container/WSL Issues	Containerized or WSL environments trigger path, permission, or runtime mismatches.	28	0.7%
	• Proxy/VPN/Networking	Network intermediaries disrupt connectivity or tool access.	25	0.7%
13	**Model Selection & Availability**	Model selection or access fails, preventing users from running with the model.	**97**	**2.5%**
	• Invalid Model Name Errors	Provided model identifiers are rejected or it is invalid.	39	1.0%
	• Model Overload & Availability	Requests fail because the selected model is overloaded or unavailable.	20	0.5%
	• Custom/Default Model Resolution Errors	The system resolves to the wrong model despite user configuration.	38	1.0%
14	**Hook & Automation Issues**	Automation triggers run at the wrong time or fail to execute in the expected context.	**53**	**1.4%**
	• Hook Configuration Errors	Hook definitions are malformed or misconfigured.	20	0.5%
	• Automation Execution Failures	Automations do not trigger or terminate unexpectedly.	23	0.6%
	• Session/Directory-Specific Execution Issues	Automations behave differently depending on directory or session scope.	10	0.3%
15	**Cost & Billing**	Billing or subscription-related issues that undermine trust in charging and accounting.	**39**	**1.0%**
	• Unexpected Charges	Users observe charges that do not align with their expected usage or plan.	22	0.6%
	• Cost Reporting Errors	Usage dashboards or cost summaries reports are incorrect.	17	0.4%
16	**Image & Media Handling**	Media upload, processing, or rendering fails, preventing the successful use of image-related features.	**37**	**0.9%**
	• Image Upload/Processing Errors	Uploads fail or processing stalls.	33	0.8%
	• Image Display Issues	Images fail to render correctly or do not appear in the UI.	4	0.1%

Table 6: Symptom category distribution by model.

Symptom Category	Claude Code	Gemini	Codex CLI	Total	Rate (%)
API & Server Communication	423	40	245	708	18.3
CLI/Terminal Experience	316	72	154	542	14.0
Command Execution	288	40	164	492	12.7
Code & File Operations	171	36	74	281	7.3
AI Model & Output Control	112	27	132	271	7.0
Authentication & Access	129	24	105	258	6.7
User Feedback & Usability	164	22	57	243	6.3
Session & State Management	140	16	68	224	5.8
Configuration & State	145	15	49	209	5.4
Installation & Setup	122	10	34	166	4.3
Security & Data Management	107	5	16	128	3.3
Integration & Compatibility	68	10	38	116	3.0
Model Selection & Availability	59	3	35	97	2.5
Hook & Automation Issues	48	5	0	53	1.4
Cost & Billing	28	3	8	39	1.0
Image & Media Handling	23	1	13	37	1.0
Total	**2343**	**329**	**1192**	**3864**	**100.0**

incorrect shell configuration, missing environment activation, or incompatible shell behavior.

Security & Data Management. Symptoms in this category involve security risks or incorrect handling of sensitive resources, potentially exposing data or performing actions without adequate safeguards. Subcategories include: 1) *Credential Leaks*: secrets appear in logs/output or are stored insecurely, 2) *Silent/Unauthorized Actions*: actions are performed without proper authorization, 3) *File/Directory Access Control*: improper access to restricted files or directories, 4) *Keychain & Credential Store Issues*: failures reading or writing credentials from secure stores.

Integration & Compatibility. This category shows symptoms that appear when the system interacts with external tools or runs under varied development environments, where integrations break or behave inconsistently. Subcategories include: 1) *Third-party Tool Support*: failures invoking or coordinating external tools, 2) *Editor Integration*: problems occur when integrating with IDE/editor workflows, 3) *Dev Container/WSL Issues*: failures in containerized or WSL environments, 4) *Proxy/VPN/Networking*: networking intermediaries disrupt connectivity or tool access. Common symptoms that we usually meet are related to errors in using models in IDEs such as VSCode and Cursor.

Model Selection & Availability. Symptoms here occur when users select models or when the platform resolves model choices incorrectly, leading to being unable to run to select models. This category include: 1) *Invalid Model Name Errors*: model identifiers (i.e., name) are rejected or unrecognized, 2) *Model Overload & Availability*: capacity limits or outages prevent model access, 3) *Custom/Default Model Resolution Errors*: the system chooses the wrong model (e.g., ignores user selection or chooses wrong defaults).

Hook & Automation Issues. Symptoms related to automation logic (hooks, scripted triggers, scheduled actions) that fail to run reliably or run in the wrong context. Subcategories are: 1) *Hook Configuration Errors*: hook definitions are invalid or misconfigured, 2) *Automation Execution Failures*: automation does not trigger, crashes, or ends early, 3) *Session/Directory-Specific Execution Issues*: automation behaves differently depending on working directory or session scope.

Cost & Billing. This category is reflected in user-observed billing issues. Such as users observing charges that do not match expectations, or usage dashboards and cost summaries are incorrect.

Table 7: Bug location distribution by model.

Architectural Layer	Claude Code	Gemini	Codex CLI	Total	Rate (%)
Tool/API Orchestration	949	98	408	1455	37.6
Command Execution & Monitoring	563	68	332	963	25.0
User Interface & Interaction	347	75	207	629	16.3
State & Memory Management	228	30	94	352	9.1
LLM Reasoning Planning	113	22	111	246	6.4
Prompt Engineering & Context Management	143	36	40	219	5.6
Total	**2343**	**329**	**1192**	**3864**	**100.0**

Image & Media Handling. This category describes symptoms arising when users upload, process, or view media, where media operations fail or render incorrectly. Subcategories include *Image Upload/Processing Errors*, where uploads fail, processing stalls, or formats are rejected unexpectedly, and *Image Display Issues*, which images render incorrectly or do not appear.

We further show the distribution of symptom categories across the three tools in Table 6. *API & Server Communication* is the most frequent category (708, 18.3%), followed by C*LI/Terminal Experience* (542, 14.0%) and *Command Execution* (492, 12.7%). Together, these three categories account for roughly 45% of all observed symptoms, indicating that user-visible failures often emerge when the system interacts with outside tools, terminal-based interaction, or executes commands.

Beyond these leading categories, several mid-frequency symptom groups, such as *Code & File Operations* (281, 7.3%), *AI Model & Output Control* (271, 7.0%), and *Authentication & Access* (258, 6.7%) remain non-trivial, reflecting that users also encounter failures in local project manipulation, model output, and access workflows. In contrast, categories such as *Cost & Billing* (39, around 1.0%) and *Image & Media Handling* (37, less than 1.0%) appear comparatively rare in the data, suggesting that user-visible issues are dominated by inner execution and interaction rather than peripheral platform features.

Answer to RQ3: Our symptom analysis indicates that failures are most visible during the system's execution steps, particularly when it relies on outside tools and command-line operations. In comparison, symptoms that occur outside the core execution pipeline occur less frequently.

4.4 RQ4: Bug Locations

This research question aims to identify where failures manifest within the architectural stack of a typical AI coding tool (as shown in Section 2.1), thereby uncovering structural weaknesses, cross-layer dependencies, and components that are particularly prone to failure. Due to limited resources, we did not reproduce the studied bugs; instead, we inferred the bug location of each bug through careful analysis of bug reports, issue discussions, and code snippets when available. Consequently, we do not analyze how these failures propagate across layers.

Table 7 shows the bug distribution across different architectural layers of AI Coding tools. Our analysis reveals that bugs in LLMs are not uniformly distributed across architectural layers, but instead concentrate heavily in a small number of core layers. For example, *Tool / API Orchestration* is the most affected layer, accounting for 37.6% of studied bugs. This indicates that interactions between the AI system and external tools constitute a primary source of system

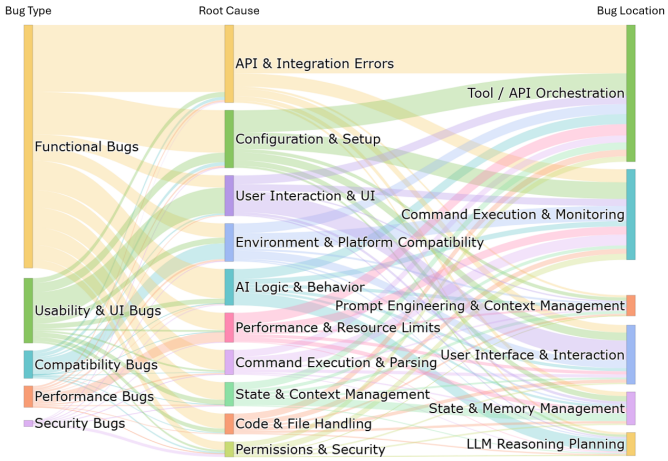

Figure 2: Mapping of bug types, root causes, and bug locations

fragility. A second major concentration of bugs occurs in *Command Execution & Monitoring*, reflecting the complexity of constructing, executing, and supervising system-level commands, which accounts for 25% of all bugs. In contrast, *User Interface & Interaction* exhibits a moderate but consistent number of bugs, which accounts for 16.3% of all bugs. These failures are typically localized and relate to mismatches between system assumptions and user behavior, such as unsupported inputs or incorrect rendering states, and rarely propagate deeply into other layers.

Figure 2 presents the mapping between bug types, their root causes, and the affected execution layers. Functional bugs dominate the distribution and exhibit a wide range of root causes; in particular, most bugs stemming from API misuse and configuration errors manifest as functional issues. In contrast, other bug types tend to have root causes closely aligned with their respective domains. For example, usability and UI bugs are primarily caused by user interaction or interface design issues and ultimately affect the UI layer. Moreover, bugs arising from diverse root causes frequently propagate to the execution stage of Tool/API orchestration, command execution, and UI. This observation reinforces our findings that core components of AI coding tools, such as rapid API invocation, command execution, and user interaction, remain key weaknesses, as these elements form the central logic of the tools and mediate the interaction between the system and its users.

Answer to RQ4: Bugs in AI coding tools occur across multiple architectural layers. Issues in the *Tool/API Orchestration* and *Command Execution & Monitoring* components are particularly prevalent, forming the dominant share of the observed failures.

5 LESSON LEARNED

Our study reveals several interesting findings that can serve as practical guidelines for both industry and academic communities to improve AI coding tool development in the future.

System-Level Failures Dominate: Our results suggest that many failures in AI coding tools arise from weak coordination among the language model, supporting infrastructure, and execution environment, rather than model reasoning alone. As shown in Section 4.2, *API & Integration Errors* (21.4%) and *Configuration & Setup* issues (15.9%) account for a large portion of failures, typically occurring at system boundaries such as API mismatches, authentication, network conditions, environment variables, path resolution, and version compatibility. Failures classified as *AI Logic & Behaviour*, which reflect intrinsic model issues like instruction noncompliance or hallucinations, represent a smaller share (10.0%), likely due in part to reporting bias: developers tend to report failures that block workflows, while suboptimal model outputs that do not halt execution are often tolerated or corrected. Consequently, system-level failures that impact usability and workflow continuity dominate issue reports.

Explicitly Define and Maintain Interactions: The interactions among LLMs, tools, and execution environments should be explicitly specified, validated, and continuously maintained. From a software engineering perspective, building AI coding tools introduces a substantially expanded set of artifacts, such as tool schemas, prompt templates, environment assumptions, and execution logs, all of which require systematic management and evolution. In addition, we also observe that many API and configuration failures stem from fragmentation issues, such as unexpected API formats, version mismatches, or incompatible runtime environments, which place significant stress on testing and validation. Proactively checking these conditions before execution can prevent cascading failures, avoid incorrect follow-up actions, and reduce user confusion. For example, AI coding tools could dynamically inspect users' environments and automatically adapt or select compatible tools and configurations accordingly.

Provide Clear and Reliable Execution Feedback: Execution feedback should be clear and reliable for both users and models. Many failures attributed to the tool were, in fact, caused by hidden errors or missing feedback from earlier command executions, which emphasizes the importance of proper logging and traceability. Providing clear and structured execution feedback helps both users and models understand what occurred, preventing incorrect attribution of failures to the model and supporting more effective debugging and workflow recovery.

Indicate Uncertain or Unreliable System States: Tools should clearly indicate when execution feedback or system state is unreliable. Without such signals, models may act on incorrect information and produce wrong outputs. Simple indicators of invalid or uncertain state can help models pause, retry, or ask for clarification, improving robustness and user trust.

6 THREATS TO VALIDITY

Our study is subject to several threats to validity. First, external validity may be limited by the scope of the studied tools. We analyze bugs from only three representative AI coding tools. Although these tools are widely used and cover different design choices, they may not fully represent the diversity of AI coding tools in terms of architectures, functionalities, and usage contexts. Consequently, our findings may not generalize to all AI-assisted coding systems.

Second, construct validity is affected by the manual labeling process. The categorization of bug types, root causes, symptoms, and architectural locations was performed manually based on bug reports and related artifacts. While we followed a consistent labeling protocol, manual analysis is inherently subjective and may introduce bias or inconsistencies.

7 CONCLUSION

In this paper, we presented the first large-scale empirical study of real-world bugs in widely used AI coding tools, namely Claude Code, Codex CLI, and Gemini CLI. Through an analysis of over 3.8K user-reported bugs, we characterized the defect landscape across bug types, root causes, bug symptoms, and bug locations. Our results show that most reliability issues arise from integration, configuration, and usability challenges, rather than solely from limitations of LLM reasoning. These findings demonstrate that building reliable AI coding tools is fundamentally a systems engineering problem and highlight the need for robust integration strategies, defensive LLM interaction, and improved user-facing error handling.

REFERENCES

[1] 2025. Claude Code. https://www.claude.com/product/claude-code. Accessed: 2025-12-12.

[2] 2025. Codex CLI. https://chatgpt.com/features/codex. Accessed: 2025-12-12.

[3] 2025. Gemini CLI. https://geminicli.com. Accessed: 2025-12-12.

[4] Mohammad Abdollahi, Ruixin Zhang, Nima Shiri Harzevili, Jiho Shin, Song Wang, and Hadi Hemmati. [n. d.]. Surveying the Benchmarking Landscape of Large Language Models in Code Intelligence. *ACM Transactions on Software Engineering and Methodology* ([n. d.]).

[5] Sunil Anasuri, Guru Pramod Rusum, and Kiran Kumar Pappula. 2023. AI-Driven Software Design Patterns: Automation in System Architecture. *International Journal of Artificial Intelligence, Data Science, and Machine Learning* 4, 1 (2023), 78–88.

[6] Andrew Bragdon, Steven P Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. 2010. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 455–464.

[7] Robert W Brennan and Jonathan Lesage. 2022. Exploring the implications of OpenAI codex on education for industry 4.0. In *International Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*. Springer, 254–266.

[8] Worawalan Chatlatanagulchai, Kundjanasith Thonglek, Brittany Reid, Yutaro Kashiwa, Pattara Leelaprute, Arnon Rungsawang, Bundit Manaskasemsak, and Hajimu Iida. 2025. On the Use of Agentic Coding Manifests: An Empirical Study of Claude Code. In *International Conference on Product-Focused Software Process Improvement*. Springer, 543–551.

[9] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian computing education conference*. 10–19.

[10] Daqing Hou and Yuejiao Wang. 2009. An empirical analysis of the evolution of user-visible features in an integrated development environment. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*. 122–135.

[11] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. 2024. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479* (2024).

[12] Steve R Jones, S Carley, and M Harrison. 2003. An introduction to power and sample size estimation. *Emergency Medicine Journal* 20, 5 (2003), 453–458.

[13] Rex Bryan Kline and Ahmed Seffah. 2005. Evaluation of integrated software development environments: Challenges and results from three empirical studies. *International journal of human-computer studies* 63, 6 (2005), 607–627.

[14] Seher Köksaldı, Mustafa Kayabaşı, Ceren Durmaz Engin, and Andrzej Grzybowski. 2025. Accuracy of ChatGPT, Gemini, Copilot, and Claude to Blepharoplasty-Related Questions. *Aesthetic Plastic Surgery* 49, 17 (2025), 4775–4785.

[15] Anand Kumar, Vishal Khare, Deepak Sharma, Satyam Kumar, Vijay Saini, Anshul Yadav, Sachendra Jain, Ankit Rana, Pratham Verma, Vaibhav Meena, et al. 2025. Intuition to Evidence: Measuring AI's True Impact on Developer Productivity. *arXiv preprint arXiv:2509.19708* (2025).

[16] Abdul Razzaq, Jim Buckley, Qin Lai, Tingting Yu, and Goetz Botterweck. 2024. A systematic literature review on the influence of enhanced developer experience on developers' productivity: Factors, practices, and recommendations. *Comput. Surveys* 57, 1 (2024), 1–46.

[17] Hélio Victor F Santos, Vitor Costa, João Eduardo Montandon, and Marco Tulio Valente. 2025. Decoding the Configuration of AI Coding Agents: Insights from Claude Code Projects. *arXiv preprint arXiv:2511.09268* (2025).

[18] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. 2015. Hidden technical debt in machine learning systems. *Advances in neural information processing systems* 28 (2015).

[19] 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*. 95–100.

[20] Agnia Sergeyuk, Ilya Zakharov, Ekaterina Koshchenko, and Maliheh Izadi. 2025. Human-AI Experience in Integrated Development Environments: A Systematic Literature Review. *arXiv preprint arXiv:2503.06195* (2025).

[21] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19, 6 (2014), 1665–1705.

[22] Christoph Treude and Marco A Gerosa. 2025. How developers interact with AI: A taxonomy of human-AI collaboration in software engineering. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 236–240.

[23] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936.

[24] Yanlin Wang, Wanjun Zhong, Yanxian Huang, Ensheng Shi, Min Yang, Jiachi Chen, Hui Li, Yuchi Ma, Qianxiang Wang, and Zibin Zheng. 2025. Agents in software engineering: Survey, landscape, and vision. *Automated Software Engineering* 32, 2 (2025), 70.

[25] Anthony I Wasserman and Peter A Pircher. 1987. A graphical, extensible integrated environment for software development. *ACM Sigplan Notices* 22, 1 (1987), 131–142.

[26] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.

Received 28 September 2023; revised 5 March 2024; accepted 16 April 2024