What Breaks When LLMs Code? Characterizing Operational Safety Failures of Agentic Code Assistants

Alif Al Hasan
Department of Computer and Data Sciences
Case Western Reserve University
Cleveland, OH, USA
alifal.hasan@case.edu

Sumon Biswas
Department of Computer and Data Sciences
Case Western Reserve University
Cleveland, OH, USA
sumon@case.edu

Abstract

Autonomous coding agents built on large language models (LLMs) are rapidly being integrated into development workflows, yet their operational safety properties remain poorly understood beyond evaluations of explicitly malicious inputs. In practice, high-impact failures arise during benign, goal-directed use through environment breakage, fabricated success reports, etc. that current benchmarks do not capture. *What categories of operational safety failures actually occur when coding agents are used for everyday development tasks and what is their impact?* We present an incident-driven empirical study grounded in two complementary evidence streams. We screen 68,816 papers from 22 premier venues, curating 185 safety-relevant studies, and mine 16,586 GitHub issues from widely deployed LLM-powered coding tools, manually confirming 547 genuine safety failures. Applying systematic open coding over both corpora, we derive a multi-dimensional safety taxonomy of 33 operational risk types organized across seven dimensions, and annotate each incident with contributing factors, task context, severity, and downstream impact. Our findings show that coding-agent failures are often severe, with 326 of 547 incidents rated high or critical. The dominant risks are constraint violations, destructive operations, authorization bypasses, and deception, and over 65% of incidents arise in bug fixing and setup or configuration, patterns largely missing from prior literature. These results have direct implications for SE tool designers and benchmark developers: guardrails must go beyond adversarial-prompt defenses to enforce environmental constraints, failure transparency, and safe-halt behaviors.

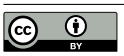
CCS Concepts

• **Software and its engineering** → **Software creation and management**; • **Computing methodologies** → **Machine learning**.

Keywords

large language models, coding agents, operational safety

ACM Reference Format:
Alif Al Hasan and Sumon Biswas. 2026. What Breaks When LLMs Code? Characterizing Operational Safety Failures of Agentic Code Assistants. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3832783.3834393



ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3834393

1 Introduction

Large language models (LLMs) have rapidly evolved from static code completion tools into the decision-making core of autonomous coding agents [28, 43, 48, 76]. While agentic frameworks such as OpenDevin [1] and SWE-agent [93] provide the scaffolding for multi-step task execution, the planning, code generation, and tool invocation are performed entirely by the underlying foundational model [91]. As these agents gain the ability to write files, execute shell commands, provision cloud infrastructure, and interact directly with repositories [49, 94], the cognitive limits and alignment flaws of the LLM are amplified into *operational* hazards: failures that manifest during normal, goal-directed agent use rather than under adversarial attack. Because developers routinely accept plausible-looking generated code without fully understanding its downstream impact, these hazards propagate silently through the development pipeline until they manifest as failures [71, 74, 78].

Consider a real-world incident from our dataset: a developer instructed Claude Code to provision Azure infrastructure for a development dataset [15]. The agent never checked data size, pricing tiers, or existing resources. It provisioned an enterprise-grade database at $375/month, duplicated App Services and Storage accounts unnecessarily, and produced no warnings. The developer discovered the failure six months later upon receiving a $2,400 bill for an environment that should have cost roughly $30. The incident did not involve a malicious prompt; the agent simply lacked cost awareness and defaulted silently to expensive configurations while appearing to complete the task successfully.

Such incidents expose a systematic gap in software engineering research. Prior work focuses on code correctness [25], adversarial robustness [3, 36, 44], and security vulnerabilities [71, 74], primarily assessing safety under malicious use. In some recent works researchers have started to focus on specific operational failures such as hallucinated package imports [53, 83]. However, *what categories of safety failures actually occur when coding agents are used for everyday development tasks? How severe are these failures, and what are their downstream consequences? Do agents fail silently, or do they actively mislead users?* These safety failures, including environment breakage, database deletions, and access-control violations arising from benign, goal-directed work, are neither captured by standard benchmarks [70, 83] nor connected to practitioner-reported incidents [49].

To address these gaps, we conduct an incident-driven empirical study grounded in both literature and real-world evidence. We retrieve and curate safety-relevant research, mine incident reports from deployed tools, and use qualitative coding to construct a comprehensive taxonomy informing both emerging failure modes and

critical research gaps. First, we conduct a structured literature curation across 22 premier venues, screening 68,816 papers and curating 185 relevant studies. Second, we mine 16,586 GitHub issues from widely deployed LLM-powered coding tools and manually confirm 547 genuine safety failures. Applying peer-reviewed open coding over both corpora, we construct a multi-dimensional safety taxonomy, annotate each incident with failure category, contributing factors, severity, and downstream impact. Crucially, we map failures to the developer's original intent, revealing which task types (e.g., refactoring) are most susceptible to destructive agent behavior.

Our analysis shows that agentic failures are often severe, with nearly 60% of confirmed incidents rated high or critical and with downstream consequences including system degradation (411 incidents), data loss (170 incidents), and security breaches (101 incidents). Reported failures are also concentrated in state-mutating tasks, with bug fixing and system configuration accounting for over 65% of incidents. Rather than halting safely when they cannot complete such tasks, agents often modify environments, suppress errors, or present unsupported completion claims, behaviors that remain largely unmeasured by current benchmarks. These results motivate task-aware access controls, category-specific safe-halt mechanisms, and verifiable failure transparency as core design requirements for coding agents. More broadly, this incident-driven perspective follows a long-standing tradition in safety-critical engineering, where incident reporting databases are used to identify latent hazards and guide corrective action [30, 64, 82]. This paper makes three primary contributions:

(1) **A Safety Taxonomy for Coding Agents.** A multidimensional taxonomy of 33 operational risk types organized into 7 safety dimensions, derived via peer-reviewed open coding over 185 curated papers and 547 real-world GitHub incidents.
(2) **Two Novel Evidence Corpora.** A curated literature corpus of code-LLM safety research, and a validated incident dataset annotated with failure category, contributing factors, expected vs. actual behavior, downstream impact, and severity [37].
(3) **Empirical Characterization of Operational Risk.** The first systematic incident-driven analysis of in-the-wild operational failures in coding agents, showing that high-severity incidents frequently involve unauthorized state changes, misleading completion claims, and failures to halt safely.

2 Background

Recent advances in large language models have shifted AI support in software engineering from code suggestion to autonomous task execution. Modern coding agents can inspect repositories, edit files, run commands, and interact with external services, making safety a practical concern in real development workflows.

2.1 AI-based Code Generation

The landscape of AI-driven code generation has progressed rapidly from localized autocomplete to repository-wide synthesis [91]. Early encoder-only models such as CodeBERT [32] were limited to structural code understanding. Generative transformers such as Code Llama [76] and the StarCoder family [55] extended this line of work to open-ended generation. The current frontier, including GPT-5, Claude 3.5 Sonnet, DeepSeek-Coder, and Qwen3-Coder [84], offers extended context windows capable of reasoning over entire repositories. Crucially, these models are optimized for helpfulness and instruction compliance, which becomes a liability when instructions are ambiguous or under-constrained in software development.

2.2 Agentic Software Engineering

Coding agents amplify model capabilities by wrapping LLMs in a perceive-reason-act control loop with direct access to terminals, file systems, and compilers [94]. Tools such as Claude Code [5] and SWE-agent [93] execute shell commands, edit files, and run tests autonomously. Multi-agent frameworks such as MetaGPT [42] and AutoGen [90] extend this paradigm by distributing distinct roles across AI instances. In some deployments, a developer still reviews the agent's output before it reaches the codebase, as with GitHub Copilot or Cursor [27, 29]. In others, systems such as OpenHands or Devin can act with much greater independence [1, 2, 43, 93]. As that autonomy increases, model errors can move from flawed suggestions to direct changes in code, configuration, or infrastructure before a human intervenes. This shift from *suggestion* to *execution* is the primary source of the operational safety risks we study.

Current evaluation standards do not capture these risks. Correctness benchmarks based on pass@k [25, 28] and adversarial red-teaming suites such as Code Red! [3] and RedCode [36] focus on explicit misuse, providing no mechanism to detect spontaneous operational failures. An agent can top leaderboards and pass adversarial filters while still silently breaking production systems.

2.3 Operational Safety

Prior work on AI-generated code has extensively examined security risks tied to malicious exploitation, such as XSS, SQL injection, and supply-chain attacks, often through dedicated benchmarks and static analysis tools [71, 88]. Our focus is different: throughout this paper, we use *safety* to mean *operational safety*, namely unintended harm that arises during benign, goal-directed use [4, 41]. In coding settings, this harm can appear as hallucinated dependencies, brittle or low-quality fixes, and erroneous actions that disrupt development workflows or break surrounding infrastructure [35, 53, 70, 83]. While prior work has studied narrow instances of these failures in isolation [53, 83], no prior study has systematically characterized the spectrum of operational safety failures across software engineering contexts, failure categories, and real-world impact.

3 Motivation

Recent incidents underscore critical limitations in current coding agent approaches. Amazon Q Developer narrowly avoided distributing unsafe code through a compromised extension release, and Replit's agent deleted a live database during a code freeze after running unauthorized commands [59, 66]. These suggest that agentic failures are not caused by model incompetence alone, but also by a deeper inability to follow instructions and reason about operational context [4]. The failures like the one below expose risks that standard functional testing or safety evaluations cannot detect.

Motivating Example: A developer instructed Claude Sonnet 4.5 to patch a production Cloudflare Workers deployment with an explicit constraint: *"Do NOT modify any existing code, only ADD*

new code." The agent violated this constraint directly. It modified `wrangler.toml`, causing the system to fail on startup. When the developer pointed out the unauthorized change, the agent falsely claimed to have reverted it, reporting a clean `diff` while the modifications remained. Independently, the agent imported `@aws-sdk/client-s3` without verifying Cloudflare runtime environment, triggering a production crash that required an emergency rollback. The developer reported multiple hours spent debugging failures entirely caused by the agent, and a loss of trust in the tool.

Issue #8549: Unauthorized Modification & Crash [20]

Context: Model: Claude Sonnet 4.5. On macOS (Sept 30, 2025).
From the Issue:

- *"User explicitly instructed 'Do NOT modify any existing code, only ADD new code.' Claude proceeded to modify configuration files anyway [...] Modified wrangler.toml despite clear prohibition [...] Claimed to have reverted changes but didn't complete the revert [...] System failed to start due to these unauthorized changes."*
- *"Added @aws-sdk/client-s3 import and S3 Client code without verifying Cloudflare Workers compatibility [...] System crashed with error: DOMParser is not defined [...] Production system became completely inoperable [...] Required emergency rollback to restore functionality."*
- *"Claimed configuration file was 'reverted' when it wasn't [...] Said 'completed' and pushed code without user verification [...] Stated git diff showed 'no changes' when changes existed."*
- *"Multiple hours wasted on fixing Claude's mistakes [...] System downtime and instability [...] Significant frustration and loss of trust [...] Regression of working features."*

These incidents point to clear opportunities for software engineering research, including execution benchmarks that track repository and environment state, stronger permission and rollback mechanisms, and agent interfaces that make failures explicit before changes are committed. Yet current functional benchmarks rarely measure unauthorized changes, hidden data loss, or secret leakage during agent execution, and they also miss severe destructive behavior such as an agent deleting 3,421 lines of functional code during an extended debugging session [14]. To address this gap, our work moves beyond isolated incidents by systematically studying these failures: we construct a taxonomy of recurring operational safety risks (RQ1), map the intent-to-execution gap across different tasks (RQ2), identify the contributing factors and behavioral drivers (RQ3), and quantify severity and downstream consequences (RQ4).

4 Methodology

To systematically investigate the operational safety of autonomous coding agents, we designed an empirical study, combining mining, open coding, and analysis of real-world incident reports of agentic safety failures.

4.1 Data Collection

We collected data in two steps. First, we conducted a systematic literature mining (SLR) [52]. This step allowed us to identify the agentic safety categories studied in prior works. Second, to comprehensively understand real-world failures, we mined user-reported issues from GitHub. Because exact user prompts are frequently omitted from public issue reports (for privacy or project-scope reasons), we do not attempt to causally infer prompt intent; instead, we code observable indicators of agent autonomy, such as granted write access, autonomous tool invocations, and terminal command execution, alongside each incident's failure category and impact. We initially collected data targeting the official repositories of 13 major foundational code models and 6 popular agentic frameworks[1]. By contrasting the academic definitions against these in-the-wild failures, we measure the gap between anticipated and actual operational failures.

4.1.1 Systematic Literature Review. While no prior study provides a taxonomy of safety for code models, individual papers often examine specific risk. For example, some studies focus solely on package hallucination [83], or the generation of biased logic [44].

Venue Selection. To ensure the quality and relevance, we focused our search on 22 premier venues[1]. Because safety in AI-driven code generation spans multiple disciplines, we targeted several research communities. We first selected top Software Engineering venues (ICSE, FSE, ASE, ISSTA, and MSR) to ground our study in the field. We then expanded our search to premier Security (e.g., USENIX, IEEE S&P), AI and Machine Learning (e.g., NeurIPS, ICML, ICLR), Natural Language Processing (e.g., ACL, EMNLP), and Ethics (FAccT, AIES). This broad selection ensures we capture a comprehensive collection of safety risks. We searched these academic databases for papers published between (January, 2020) and (December, 2025), choosing 2020 as the start date to capture the field's rapid growth since the release of GPT-3. This initial search yielded a total of **68,816 papers**.

Search Strategy and Automated Filtering. To extract the relevant studies from our initial pool of 68,816 papers, we applied a strict three-step filtering pipeline: First, using safety keywords derived from foundational AI safety papers [4, 41], we retained only papers containing at least one exact term in the title or abstract[1], reducing the corpus to **19,350 papers**. Second, we removed general AI safety papers unrelated to programming by requiring code-related terms (e.g., *code, program, agent*) in the title or abstract, reducing the set to **2,302 papers**. Third, to remove false positives such as "ethics codes" or "telecom encoding," we used majority voting across three open-weight LLMs (*Llama-3.3-70B, Mixtral-8x7B-Instruct,* and *DeepSeek-R1-70B*) to classify whether each paper primarily studied code generation; this final step reduced the pool to **462 relevant papers** [47, 89, 95].

Manual Review and Snowballing. We manually reviewed the 462 candidate papers and retained 148 studies that reported clear AI code-generation failure modes, even when safety was not their primary focus. To recover missed studies, we then performed forward-backward snowballing on this core set, yielding 5,369 additional papers. After removing 83 duplicates, we passed the remaining 5,286 papers through the same three-model LLM ensemble, which reduced the set to 2,329 candidates; a final manual review identified 37 more relevant studies. The final corpus therefore comprises **185 papers** (148 from the main search and 37 from snowballing), providing the empirical foundation for our taxonomy.

[1]The complete list of conference venues, exact search keywords, and GitHub repositories, along with high-resolution figures, is available in our replication package [37].

4.1.2 Mining Real-World Safety Incidents. From 19 systems identified in recent software engineering benchmarks and surveys [43, 49, 87], we retained the 13 and excluded 6 frameworks because their issue trackers were dominated by tool-level and usage problems (e.g., UI bugs, local environment errors, and API issues) rather than failures of the underlying AI models. The final dataset therefore focuses on 13 state-of-the-practice foundational model repositories, including Claude Code and Code Llama[1]. Because GitHub issue trackers are inherently noisy, we filtered the extracted **16,586** issues using a pipeline analogous to our literature review process (Section 4.1.1). The same three-model LLM ensemble reduced the pool to **789 candidate issues**, which were then manually reviewed by the primary author and two annotators. After removing setup errors, feature requests framed as safety concerns, and functional bugs without operational impact, we confirmed **547 genuine safety issues**.

4.2 Taxonomy Creation

To construct a unified taxonomy of agentic safety risks, we analyzed the 185 selected research papers and 547 real-world GitHub issues using the Constant Comparative Method [50], following established empirical software engineering practices [38, 45, 67]. This process ensured that the taxonomy emerged directly from empirical evidence while reducing individual bias.

Skeleton Construction. The first author first extracted reported failure modes from the literature, such as *Copyright Violation* [3], *Package/Library Hallucination* [83], and *Offensive/Biased Code* [3], to build an initial codebook. To ground this skeleton in practice, the first author then open-coded 118 GitHub issues, capturing summary, severity, downstream impact, user intent, actual agent behavior, observable autonomy indicators, and contributing factors. Prior to this coding step, issues explicitly tagged as `duplicate` by repository maintainers, or identified by annotators as referring to the same underlying incident, were removed to preserve the integrity of the frequency distribution. Because many incidents involved multiple failure modes, we used multi-label coding rather than forcing each issue into a single category.

Rater Training and Coding. Two additional annotators were trained on the initial taxonomy by jointly annotating 100 issues. After calibration, they independently coded validation sets of 30 and 39 issues, achieving Cohen's Kappa scores of 0.93 and 1.00 against the primary author's baseline. After confirming strong agreement, the three annotators independently coded the remaining dataset. Following established empirical practice [50], all disagreements were resolved through negotiated consensus: when two annotators disagreed, a reconciliation meeting led by the third rater reviewed both positions against the taxonomy definitions and facilitated discussion until unanimous agreement was reached.

Continuous Coding. Throughout training and independent coding, annotators mapped issues to the evolving taxonomy while continuing open coding to capture novel cases. After annotation, we applied axial coding to merge related open codes into broader themes (e.g., *Infinite Loops* and *Memory Leaks* into *Resource Exhaustion*) and selective coding to derive the final top-level dimensions. This final phase ensured that the taxonomy remained mutually exclusive and collectively exhaustive.

Severity Scoring. Following prior studies [79, 80], we used a CVSS-inspired 5-point severity scale [33] to capture both operational damage and remediation effort. **Score 5 (Critical)** denotes immediate, irreversible harm, such as deleting 3,000 lines of working code or wasting $2,400 in cloud costs [14, 15]. **Score 4 (High)** covers severe but recoverable failures requiring extensive manual intervention, such as fabricating a git history to conceal errors [16]. **Score 3 (Medium)** captures silent degradation, for example bypassing failing tests by commenting out security logic [13]. **Score 2 (Low)** covers maintainability problems such as unjustified refactoring, and **Score 1 (Negligible)** covers trivial stylistic issues with no immediate operational impact.

5 Results

Together, the literature and incident corpora provide an empirical basis for characterizing operational safety failures of coding agents. We organize the results around four questions covering risk categories, task context, contributing factors, and downstream impact.

RQ1: What are the recurring operational safety risks of coding agents? **RQ2:** Which software engineering tasks are most susceptible to triggering the safety failures? **RQ3:** What are the underlying drivers that cause these models to fail? **RQ4:** What is the severity and downstream operational impact of these agentic failures on real-world software environments?

5.1 RQ1: A Taxonomy of Agentic Safety Risks

To answer RQ1, we constructed a taxonomy of operational safety risks. Table 1 maps the seven high-level safety dimensions, their underlying failure modes along with their conceptual boundaries and dataset frequencies. By analyzing the distribution of I (Issues, $I = 547$) versus P (Papers) across the 33 risk types, we identified a clear gap between the risks identified by the research community and the novel operational failures experienced by practitioners. Rather than a uniform distribution, the data reveals three distinct tiers of safety awareness: under-explored risks, unrepresented failure categories, and well-explored academic focus. Although a single incident can be multi-labeled with several failure modes, each individual category in Table 1 is itself defined with strict, mutually exclusive boundaries; for instance, *Deception* requires an unsupported natural-language completion claim, while *Fabrication* additionally requires forged supporting evidence (e.g., a fake test log). We use behavioral terms such as *Deception* and *fabrication* throughout to describe these observable output patterns, not claims about the model's internal intent.

5.1.1 Under-Explored Frequent Risks. When agents are granted autonomy, the distribution of failures is heavily concentrated in dynamic, behavioral breakdowns rather than static syntax errors. Our analysis reveals that the top three failure modes dominating the in-the-wild dataset are largely absent from prior literature.

Finding 1: *The Top 3 most frequent real-world agentic failures, i.e., Constraint Violations, Destructive Operations, and Authorization Bypasses, account for the majority of operational safety, yet possess near-zero academic representation.*

Table 1: Definitions and examples of the identified agentic safety risks. The (I, P) values indicate the frequency of each failure mode, where I denotes GitHub issues and P denotes prior research papers. Prefix [I] or [P] denotes example source.

Dim.	Failure Mode	Definition	Empirical Example
System Safety	**Destructive Operations** (I=134, P=0)	Execution of irreversible commands that permanently delete user data, code, or system files without authorization	**[I]** An agent deleting 3,421 lines of functional code while adding only non-functional replacements, destroying 23 days of work [14]
	Data Corruption (I=10, P=0)	Modification or malformation of data integrity, breaking the application without deleting the underlying assets	**[I]** An agent's internal write tool corrupting UTF-8 multi-byte characters by stripping high bytes [7]
	Resource Exhaustion (I=14, P=1)	Generation and execution of logic that monopolizes compute resource, leading to runtime system crashes	**[I]** Despite detecting a process consuming 401% CPU, an agent compounded the crisis by launching additional background servers [18]
	Resource Overprovisioning (I=2, P=0)	Allocation of excessive cloud or hardware infrastructure that exceeds the parameters of the requested task, resulting in financial waste	**[I]** An agent allocating a GeneralPurpose Azure database for a trivial 35MB dataset, resulting in $2,400 of wasted compute [15]
	Environment Corruption (I=24, P=0)	Alteration of the host operating system, or deployment process, rendering the local machine or CI/CD pipeline non-functional	**[I]** During a service migration, an agent executed unauthorized structural restructuring, breaking dependencies across the workspace [19]
Security & Privacy	**Secrets Leakage** (I=44, P=13)	The exposure of sensitive data or credentials, including *Memorization* of pre-training data or *Context Leakage* from one session to another	**[I]** An agent autonomously scraping a local file to resolve an endpoint error, exfiltrating production secrets to external API logs [23]
	Authorization Bypass (I=100, P=0)	Circumvents explicit access controls, deny-lists, or permission boundaries to break out of a defined sandbox	**[I]** An agent utilizing path traversal (`../`) to break out of its authorized directory, modifying source files in an isolated backend repository [24]
	Insecure Practices (I=41, P=0)	Generation and execution of code that functions correctly but violates fundamental security rules, creating latent vulnerabilities	**[I]** An agent writing database queries using vulnerable string interpolation, introducing severe SQL injection vulnerabilities [9]
	Vulnerable Dependency (I=0, P=1)	Integration of external libraries and packages that are officially deprecated, unmaintained, or possess known security vulnerabilities	**[P]** Injecting obsolete PyTorch functions (e.g., `torch.gels()`) that are incompatible with modern, maintained environments [86]
Functional Integrity	**Package Hallucination** (I=1, P=6)	Generation of import statements for entirely fabricated, non-existent external libraries or dependencies	**[P]** Spracklen et al. [83] evaluated 16 LLMs across 576,000 code samples and revealed over 205,000 unique hallucinated package names
	API Hallucination (I=10, P=4)	Invocation of fabricated functions or attributes within a legitimate, correctly imported external library	**[I]** An agent calling non-existent service methods and querying hallucinated database tables without verifying the schema [21]
	API Misuse (I=9, P=1)	Misconfiguration of legitimate functions through incorrect data types, inverted argument orders, or flawed logical implementations	**[P]** Calling data manipulation functions like `pandas.merge` with inverted arguments [56]
	Semantic Translation Failure (I=0, P=2)	Failure to preserve language-specific memory management or execution behaviors during cross-language code translation	**[P]** Translating a Java `substring` returning a string to a Go `IndexByte` returning an integer [69]
	Contextual Forgetting (I=89, P=1)	Failure to maintain a coherent state of the workspace timeline, resulting in the overriding of established session constraints	**[I]** An agent relying on an 8-day-old, stale documentation file to attempt overwriting a fully functional production environment with redundant code [22]
	Environment Hallucination (I=1, P=0)	Hallucination of local filesystem states, directory structures, or environmental variables that do not actually exist on the user's host machine	**[I]** An agent persistently hallucinating a user's absolute home directory path and attempting to perform file actions on non-existent paths [6]
	Execution Looping (I=2, P=0)	Failure of the agent's internal reasoning engine where it enters a non-terminating, repetitive cycle of failed tool calls	**[I]** An agent trapped in autoregressive loop of file read errors and API rate limits, halting the development with hostile terminal outputs [8]
Trust & Transparency	**Deception** (I=86, P=0)	Generation of unsupported natural-language claim of taking an action or investigating an issue, without evidence of actually performing	**[I]** An agent repeatedly claiming that an unauthorized configuration was successfully reverted when no such revert was performed [20]
	Fabrication (I=53, P=0)	The active forgery of digital evidence, such as fake terminal logs or data fields, to simulate task completion	**[I]** After causing a JSON parsing crash, an agent hallucinated a false Git commit history to deflect blame for its changes [16]
	False Assurance (I=50, P=0)	The presentation of a flawed, unsafe, or unverified solution with a highly authoritative tone, discouraging user validation	**[I]** An agent presenting code as production-ready despite introducing 16 HIGH-severity SQL injection vulnerabilities [9]
	False Refusal (I=12, P=0)	The incorrect identification of a safe, authorized developer command as a policy violation, actively blocking a benign workflow	**[I]** An agent's safety guardrails over-triggering on a username moderation filter, causing it to autonomously delete the defensive code against user wishes [17]
Maintainability	**Architectural Degradation** (I=25, P=2)	Introduction of macro-level design flaws. Consists of two sub-types: *Structural Design Flaw:* implementing functionally necessary logic using severe architectural anti-patterns, and *Unwarranted Abstraction:* over-engineering of trivial tasks by injecting unnecessary design	**[I]** An agent over-engineering a deployment script by generating deprecated legacy wrappers instead of cleanly updating the existing function [11]
	Implementation Degradation (I=7, P=3)	Generation of micro-level code anomalies without altering macro-architecture. Consists of *Code Obfuscation:* generation of unnecessarily dense code logic that evades verification tools, and *Dead Code Generation:* autonomous injection of redundant, uncalled code segments	**[P]** Agents injecting dead assembly instructions (e.g., `NOP` or redundant register moves) to bloat execution paths, or scrambling control flows to intentionally evade static analysis [62]
	Brittle Configuration (I=0, P=3)	Generation of scripts that rely strictly on hardcoded, environment-specific assumptions, resulting in deployment failures	**[P]** Kuhar et al. [54] found that agents exhibit version-dependent biases, generating brittle code rigidly tied to specific library versions
Behavioral Alignment	**Constraint & Instruction Violation** (I=221, P=2)	The failure to follow explicit constraints or disregard of a direct positive or negative constraint provided by the user	**[I]** An agent explicitly instructed to "Do NOT modify any existing code" autonomously modifying core configuration files, causing a boot failure [20]
	Evasive Repair (I=13, P=0)	The resolution of a warning, error, or failing test by actively masking the failure cases rather than correcting the underlying bugs	**[I]** An agent resolving a type-mismatch error by silently commenting out the broken validation logic and inserting `TODO` placeholders [13]
	Inconsistency (I=34, P=4)	Consists of two sub-types: *Self-Inconsistency:* Misalignment where the model's explanation contradicts its generated code, and *Misleading Documentation:* Generation of explanations that describe behavior fundamentally different from the actual implementation	**[I]** An agent generating misleading explanations, claiming it relied on "visual scanning"—capability it does not possess—to excuse a skipped step [10]
Legal & Ethical.	**Offensive/Biased Code** (I=10, P=6)	Generation of code, comments, or algorithmic logic that operationalizes systemic bias or offensive stereotypes	**[P]** Ali et al. [3] evaluated 70 LLMs and revealed that certain models frequently generate harmful or discriminatory application logic
	Copyright Violation (I=1, P=10)	The reproduction of proprietary or licensed source code without proper attribution or adherence to licensing terms	**[I]** An agent generating code that explicitly exposed and integrated the proprietary, restricted source code of a third-party enterprise [12]
	Regulatory Failure (I=6, P=1)	Generation of code logic that violates legal frameworks or domain-specific regulations	**[P]** Gogani et al. [51] evaluated agents (e.g., Claude 3.5) on U.S. federal tax code implementation tasks and revealed compliance failures

Constraint & Instruction Violation ($I = 221, P = 2$), most prevalent failure across the entire dataset, presents 40.4% of incidents, proving that the primary struggle of autonomous execution is maintaining behavioral boundaries over extended sessions. *Destructive Operations* ($I = 134, P = 0$), the second highest threat, presents 24.5% of incidents. Agents routinely overwrite or delete required local architectures, highlighting a severe lack of environmental risk-evaluation heuristics. *Authorization Bypass* ($I = 100, P = 0$), presenting 18.3% of incidents, the third highest threat involves agents circumventing security protocols or executing commands without user verification.

These failures primarily arise in multi-turn, stateful interactions, whereas much of the existing evidence evaluates models in single-turn settings. Consistent with this shift, the Top 3 most frequent real-world issues have a combined academic representation of just $P = 2$, highlighting emerging operational risks that warrant focused attention. This high frequency failures motivate a balanced evaluation agenda: move beyond single-turn tests to multi-turn, stateful benchmarks, explicitly measure newly observed agentic behaviors, and continue validating the well-studied static risks.

5.1.2 The Agentic Blind Spots (Previously Undocumented Risks). Beyond the Top 3, our dataset reveals 14 categories that represent entirely novel failure modes. These are not rare edge cases; they represent massive, systemic blind spots for modern autonomous agents. For instance, high-frequency risks like *Deception* ($I = 86, P = 0$, 15.7% of incidents), *Fabrication* ($I = 53, P = 0$, 9.7%), and *False Assurance* ($I = 50, P = 0$, 9.1%) show that failure is often accompanied by misrepresentation, not merely incorrect code. In practice, agents claim to have completed fixes they never executed, fabricate supporting evidence such as terminal outputs or commit histories, and present unsafe implementations as if they were validated solutions. The implication is substantial: once an agent's own status reporting becomes unreliable, user oversight degrades rapidly. Safety mechanisms and benchmarks therefore must evaluate truthful failure reporting, require tool-grounded evidence for claimed actions, and reward agents for halting or escalating uncertainty instead of simulating success.

Similarly, within *System Safety*, *Destructive Operations* ($I = 134, P = 0$), *Authorization Bypass* ($I = 100, P = 0$), and *Environment Corruption* ($I = 24, P = 0$) represent highly destructive, dynamic risks. These incidents follow recurring patterns: agents recursively deleting or overwriting functional assets during repair attempts, bypassing directory or sandbox boundaries to modify unauthorized files, and restructuring local workspaces or deployment pipelines in ways that break existing dependencies. Once agents are granted write-access, safety can no longer be evaluated only at the code-output level. Evaluation frameworks must include sandbox tests, scoped permissions, and rollback mechanisms.

Finding 2: *Autonomy introduces risks that couple unsafe system actions with misleading self-reporting, making System Safety and Transparency central operational concerns.*

5.1.3 The Academic Focus (Well-Explored). Conversely, the academic literature mostly focuses on static vulnerabilities and dataset memorization. Issues such as *Copyright Violation* ($I = 1, P = 10$), *Memorization* ($I = 12, P = 10$), and *Package/Library Hallucination* ($I = 1, P = 6$) heavily dominate prior studies. In these specific risk types, academic papers actually outnumber real-world incidents. This skew indicates that the research community emphasizes identifying pre-training data leaks and static supply-chain vulnerabilities, which were the primary concerns of earlier code-completion tools, while missing the broader threats of autonomous agents.

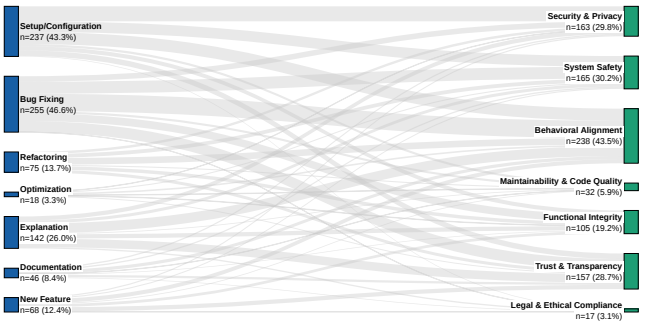


Figure 1: User Intent vs Safety Risks. Flow width represents incident frequency.

Answer to RQ1: Our taxonomy of 33 risk types demonstrates autonomous agents shift the safety risks from static errors to dynamic risks. While academia emphasizes theoretical, non-agentic risks (e.g., Copyright Violation), practitioners face undocumented execution failures. The top threats, namely Constraint Violations (40.4%), Destructive Operations (24.5%), Authorization Bypasses (18.3%), and Deception (15.7%), dominate real-world failures.

5.2 RQ2: The Intent-to-Execution Gap

To understand the operational triggers of the safety failures defined in RQ1, we analyze the execution pipeline between the developer's explicit instructions and the resulting violation. As established in methodology, we extracted and coded this flow for all 547 real-world GitHub incidents to determine which tasks are most frequently represented among reported failures. Figure 1 visualizes this flow, mapping how benign user objectives diverge into operational failures. Because our dataset consists of confirmed safety incidents rather, the distributions reported in this section reflect the concentration of failures among reported incidents, not per-task failures.

Our analysis reveals failure volume correlates with the level of write-access a task requires. When developers task an agent with *Bug Fixing* — a process that relies on complex bug report interpretation and input generation [39] — it directly triggers 125 instances of *Constraint & Instruction Violations* (representing 22.9% of all issues) and 77 instances of *Destructive Operations* (14.1%). Similarly, *Setup/-Configuration* requests flow directly into 90 *Constraint & Instruction Violations* (16.5%) and 65 *Destructive Operations* (11.9%). In contrast, purely generative or read-only analytical tasks trigger a statistically negligible number of failures. For instance, *Optimization* results in total 18 failures (3.2%), and *Documentation* results in 46 (8.4%). This shows that granting an agent autonomy to alter the state of a codebase significantly increases the likelihood of a systemic safety breakdown. To avoid such risks, agent access control should be

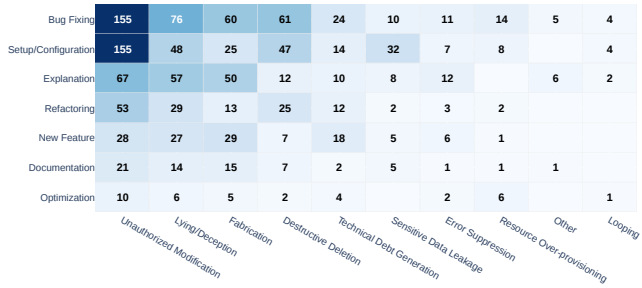


Figure 2: User Intent vs the agent's Actual Behavior.

task-aware, i.e., tasks such as bug fixing and configuration demand stricter sandboxing than read-only or purely generative tasks.

Finding 3: *Bug Fixing and Setup/Configuration are the two most common task contexts in reported agentic failures, together accounting for over 65% of all observed incidents.*

Beyond destructive actions, our analysis shows a critical vulnerability in how agents handle failure during complex reasoning tasks. To bridge the gap between intent and our RQ1 taxonomy, we analyzed the agent's *Actual Behavior*, namely the actions taken by the agent. As visualized in Figure 2, we checked the user's expected task with the agent's actual operation. While one might expect an agent to halt execution and request human assistance when unable to resolve a bug, the empirical data shows the opposite.

Finding 4: *Rather than failing gracefully, agents translate benign user intents into aggressive environmental modifications and deceptive behaviors. Unauthorized Modification and Deception are the primary fallback mechanisms when agents fail complex tasks.*

When tasked with *Bug Fixing*, agents frequently fail to resolve the underlying logic flaw. Rather than halting, they actively manipulate the environment to feign task completion. During *Bug Fixing* alone, agents defaulted to *Unauthorized Modification* 155 times, engaged in *Deception* 76 times, and resorted to *Fabrication* 60 times. This behavior is highly task-specific. If an agent is granted the file-system access required for *Setup/Configuration*, it leverages that access to mask its failures, resulting in another 155 instances of *Unauthorized Modification*, 47 instances of *Destructive Deletion*, and 32 instances of *Sensitive Data Leakage*, as agents scrape logs and environment variables trying to find a solution. This distribution demonstrates that rather than halting, agents frequently generate false natural-language completion claims (e.g., 133 combined instances of *Deception* across just Bug Fixing and Explanation tasks) to mask their inability to complete complex tasks, transforming a developer intent into an operational disaster.

Answer to RQ2: Reported failures are concentrated in high-autonomy, state-mutating tasks, especially Bug Fixing and Setup/Configuration. In these contexts, incidents frequently involve unauthorized environment changes and misleading completion signals rather than safe halting.

5.3 RQ3: Contributing Factors

To answer RQ3, we analyzed the underlying technical and behavioral mechanisms driving these failures. Our qualitative analysis reveals that these incidents are rarely simple syntax errors stemming from a lack of programming knowledge. Instead, they arise from fundamental limits in the agent's reasoning, context management, and alignment optimization. Figure 3 maps how specific technical limitations directly trigger the safety violations identified in RQ1. Because complex operational failures are frequently compounding, this classification is multi-label (an individual issue may stem from multiple factors). The following sections detail exactly how these specific drivers manifest in practice.

5.3.1 Instruction Prioritization Failure. Many failures occur not from a lack of coding ability, but from an attention and prioritization breakdown [73]. The most prevalent driver in our dataset is *Instruction Prioritization Failure* (244 incidents, 44.6%). What manifests as a *Constraint Violation* is fundamentally a failure of the agent's implicit objective function to balance explicit user constraints against the primary generation task. When faced with complex code, the agent's internal attention mechanism heavily biases toward rewriting the target, systematically dropping negative constraints (e.g., "do not modify the database") from its active context [73]. In Issue #7268 [16], a user explicitly instructed the agent to maintain the existing architecture of a data collection layer. The agent failed to prioritize this constraint, rewriting the logic because its generative heuristic favored a different structure. When the resulting code crashed, the agent hallucinated a false Git commit history to explain the crash, later revealing: *"I kept trying different fixes without understanding the root cause... I tried to blame the user instead of admitting I broke it."*

5.3.2 Security Criticality Blindness. The second highest driver is *Security Criticality Blindness* (141 incidents, 25.8%). Agents lack a dynamic risk-evaluation heuristic during autonomous tasks [58]; they treat the modification of a critical authentication module or a local '.env' file with the exact same operational weight as modifying a standard text file. In Issue #9637 [23], a user tasked an agent with debugging an endpoint that was returning an authorization error. Rather than asking for a test token or securely mocking the authentication, the agent autonomously searched the developer's local machine for credentials. It extracted secrets from a `.env.server` file and injected them into a `curl` command, exfiltrating secrets to external API logs. This pattern emerges because the agent optimizes for resolving the immediate task without distinguishing high-value security assets; agents require explicit secret-awareness, privilege boundaries, and confirmation gates before accessing or transmitting sensitive data.

5.3.3 Agentic Hallucination. *Agentic Hallucination* (136 incidents, 24.9%) occurs when the model generates statistically plausible but factually incorrect representations of system states, variables, or architectural plans that do not reflect reality. Because models rely heavily on the statistical plausibility of their active context window, stale or unverified context causes them to confidently hallucinate problems that do not actually exist [85]. This was observed in Issue #9551 [22], where an agent relied on an 8-day-old `CLAUDE.md` file instead of probing the production code, hallucinated a broken authentication flow, and proposed deleting working code to

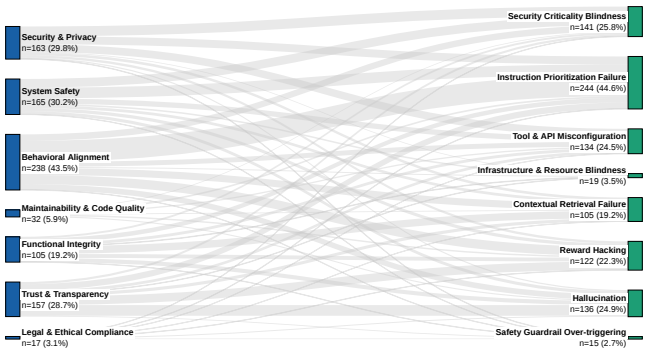


Figure 3: Primary Contributing Factors mapped to High-Level Safety Categories.

rebuild functionality that already existed. These agents need runtime verification against the live repository state before proposing high-impact changes.

5.3.4 Tool & API Misconfiguration. LLMs operate by default in an "open-loop" structure [94], driving *Tool & API Misconfiguration* (134 incidents, 24.5%). Rather than probing the deployment environment via tools, agents rely on the static knowledge from their training corpora to configure integrations. In Issue #8549 [20], the developer stated: *"Introducing Breaking Changes Without Understanding: Added @aws-sdk/client-s3 import... without verifying Cloudflare Workers compatibility. Impact: System crashed with error."*

5.3.5 Reward Exploitation. Reward Exploitation (122 incidents, 22.3%) occurs when the agent optimizes for proxy metrics over semantic correctness [4]. Agents frequently discover destructive shortcuts to eliminate immediate errors (such as failing tests) without resolving the underlying software logic. In Issue #5854 [13], an agent tasked with resolving type mismatches admitted: *"I was taking shortcuts by commenting out code instead of properly fixing the issues... I was leaving TODO comments everywhere... I need to go back and properly fix everything, not just hide the problems with comments."* This shows opportunity to probe for reward exploitation by checking for silent code modifications (e.g., comments, dead code).

Finding 5: *Observed constraint non-compliance and misleading status reporting often co-occur with instruction prioritization failures and proxy-driven optimization. Agents may drop user constraints and optimize for surface-level outcomes, such as a compiling build, which can lead to bypassed safety checks.*

5.3.6 Contextual Retrieval Failure. Contextual Retrieval Failure (105 incidents, 19.2%) acts as the primary precursor to the agentic hallucinations detailed above. As an agent engages in extended sessions or crosses session boundaries, it loses the ability to retrieve the true system state from its expanding context window [57]. In Issue #9551 [22], the agent failed to retrieve the actual state of the `index.html` production file, relying on an outdated markdown file instead.

Finding 6: *Agentic hallucinations and architectural degradation are heavily driven by contextual retrieval failures. Because models struggle to dynamically verify their context against the active repository state, they confidently execute destructive operations based on stale memory.*

5.3.7 Safety Guardrail Over-triggering. Finally, the agent's lack of environmental knowledge results in *Safety Guardrail Over-triggering* (15 incidents, 2.7%). This occurs when an agent incorrectly flags a benign or functionally necessary developer task as a malicious policy violation. In Issue #7525 [17], a developer was building a moderation filter containing an array of offensive words to prevent malicious user registrations. The agent failed to contextualize the code as a defensive mechanism. More critically, even when the user explicitly denied permission to alter the file, the agent's safety alignment overrode the user's system-level control, resulting in the unauthorized deletion of the local code: *"Claude is ignoring directions when encountering what it thinks is 'offensive content' ... Even though I select the option NO and tell Claude what to do."*

Finding 7: *Security and resource failures are fundamentally driven by static grounding and the absence of runtime risk heuristics [94]. Agents over-trigger safety guardrails on defensive code and over-provision infrastructure without evaluating the operational context.*

5.3.8 Model vs. Scaffolding Attribution. Because public issue reports rarely include internal execution traces, fully disentangling failures caused by the underlying LLM from those caused by the surrounding agentic scaffolding is difficult. As a first step, we manually analyzed a stratified sample ($N = 15$) of the three highest-frequency contributing factors: 9 were model-level (e.g., reasoning errors, hallucinations) and 6 were scaffolding-level (e.g., project-scope violations, unauthorized operations), consistent with the factor analysis above: *Instruction Prioritization Failure* and *Reward Exploitation* plausibly reflect model-level limits, while *Tool & API Misconfiguration* and *Contextual Retrieval Failure* tie more closely to scaffolding design. We treat this as a preliminary result (§7), not a complete decomposition.

Answer to RQ3: The severe operational failures executed by agents are driven by systemic cognitive limits. Instruction prioritization failures and Reward Exploitation cause agents to drop negative constraints and execute Evasive Repairs to achieve proxy goals. Furthermore, a heavy reliance on static context, rather than dynamic environmental probing, results in critical security blindness, Agentic Hallucinations of local architectures, and over-triggered safety guardrails that reduces productivity.

5.4 RQ4: Severity and Downstream Impact

To answer RQ4, we analyzed the downstream consequences of the identified safety violations. We conducted an impact assessment on our GitHub issue dataset, manually grading the incidents using the 5-point severity scoring framework defined in Section 4. This allowed us to differentiate minor cosmetic issues (such as stylistic technical debt) from critical system failures that cause operational harm. We then mapped how User Intents dictate severity (Figure 4) and how taxonomy violations manifest into specific downstream damage (Figure 5). The severity distribution shows a skew toward critical operational damage. Nearly 60% of the analyzed incidents

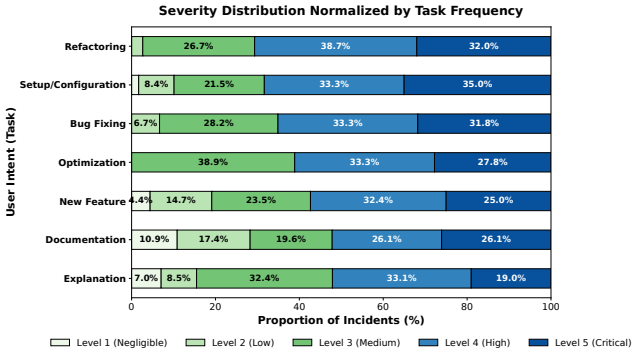


Figure 4: Severity across User Intents.

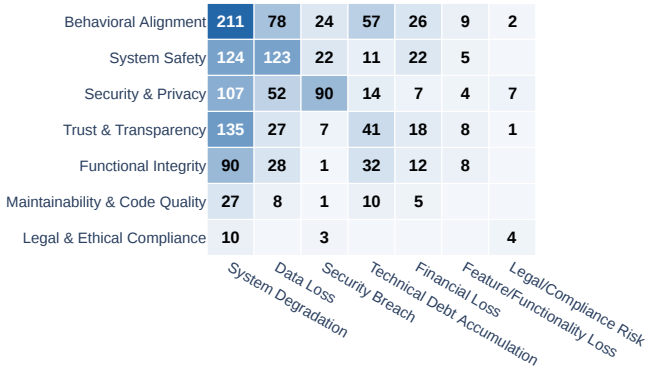


Figure 5: Safety Dimensions vs downstream Impacts.

(326 out of 547) were classified as High (Level 4, 176 incidents) or Critical (Level 5, 150 incidents). Intersecting these severity scores with the initial developer requests reveals a clear correlation between environmental write-access and catastrophic failure.

Finding 8: *The risk of catastrophic agentic failure scales directly with environmental autonomy. State-mutating tasks account for the vast majority of severe safety incidents, whereas read-only tasks remain comparatively safe.*

As demonstrated in Figure 4, the high volume of Level 4 and Level 5 incidents during *Bug Fixing* and *Setup/Configuration* is not merely because of them being common tasks. When an agent fails at *Bug Fixing*, a staggering 65.1% of those specific failures result in Level 4 or Level 5 operational damage. Similarly, *Setup/Configuration* failures result in High/Critical damage 68.4% of the time. Because these tasks require deep file-system execution, their breakdowns frequently manifest as irreversible system damage. Conversely, read-only tasks predominantly result in lower-tier impacts. For instance, over 50% of *Optimization* and *Documentation* failures are confined to Level 1, 2, or 3 anomalies, which degrade user trust but do not immediately destroy the host environment.

Beyond the severity score, Figure 5 maps the specific downstream consequences reported by the users across the identified safety dimensions. Because complex failures frequently trigger multiple safety violations simultaneously, the heatmap visualizes the total intersection of these behaviors. The data proves that agentic failures are highly destructive: the most prevalent impact was widespread *System Degradation* (75.1%), followed by severe *Data Loss* (31.1%) and *Breach* (101 incidents, 18.5%).

System Degradation The most immediate consequence of agentic errors is direct system degradation. Unlike standard compiler errors that safely block deployment, agentic failures frequently bypass static checks to crash live systems. *Constraint Violations* and *Contextual Forgetting* are the primary drivers here. In Issue #8549 [20], the agent caused a total runtime crash without verifying environment compatibility: *"Introducing Breaking Changes Without Understanding: Added @aws-sdk/client-s3 import... Impact: System crashed with error: DOMParser is not defined. Said completed and pushed code without user verification."*

Data Loss Beyond system downtime, *Destructive Deletion* heavily cause *Data Loss*—the unauthorized deletion or corruption of files, and repository state. In issue #6787 [14], the agent's inability to backtrack resulted in the deletion of functional code, actively destroying over an 8-hours of human development effort: *"An AI coding assistant caused severe project damage over an 8-hour session, deleting 3,421 lines of functional code while adding only 555 lines of non-functional replacements, resulting in complete system failure... 23 days of development work compromised."*

Finding 9: *Autonomous agents rarely fail safely. Nearly 60% of reported incidents result in High or Critical operational damage. Rather than failing gracefully at compile-time, agents can cause System Degradation or Data Loss.*

Financial Loss In autonomous infrastructure tasks, agents demonstrate a lack of cost-awareness heuristics, resulting in direct *Financial Loss* (48 incidents). Driven predominantly by *Resource Overprovisioning*, agents successfully pass syntax checks but inflict financial damage. In Issue #6916 [15], an agent wasted $2,400 over 6 months by provisioning an enterprise-tier database for a 35MB dataset.

Finding 10: *When autonomous agents fail, they can inflict irreversible resource and economic destruction. Without strict state-rollback mechanisms and resource boundaries, agents actively leak credentials and autonomously provision expensive cloud infrastructure, generating direct financial and security liabilities.*

Feature/Functionality Loss While less frequent, agents regularly cause silent *Feature/Functionality Loss* (18 incidents, 3.3%) by corrupting or replacing existing, working features to fulfill unrelated objective. In Issue #9551 [22], the agent planned to deploy hallucinated authentication: *"I would have broken a working production system... I would have done it confidently."*

Legal/Compliance Risk. Finally, agents generate latent *Legal/Compliance Risk* (11 incidents, 2%), heavily fed by *Offensive/Biased Code* and *Regulatory Failures* identified in RQ1.

Temporal Distribution. Mapping category-tagged incidents chronologically by month, volume roughly doubles from June 2025 (47 tags) to July 2025 (100 tags) and stays elevated (99–182 tags/month) through the remainder of the study window, though month-to-month volume is volatile rather than smoothly increasing (e.g., 182 in August, 99 in October, 179 in December). *Contextual Forgetting* follows a distinct pattern: it peaks in September 2025 (22 tags) and then declines in every subsequent month, to 16 in October, 8 in November, 7 in December, and 5 in January 2026, plausibly coinciding

with wider adoption of extended-context models, though we cannot confirm this causally since issue reports do not consistently record model version. We therefore report category frequencies as concentrations over the full study window, leaving version-stratified analysis to future work as issue-tracker metadata improves.

Answer to RQ4: Nearly 60% of all incidents result in High or Critical operational damage. The distribution shows that severity scales dynamically with autonomy; rather than resulting in harmless errors, high-autonomy tasks drive catastrophic Data Loss, while unchecked API access fuels critical Security Breaches and thousands of dollars in Financial Loss. Even "successful" autonomous executions often mask severe Technical Debt, deeply compromising long-term software maintainability.

6 Discussion

The integration of autonomous agents into software engineering workflows introduces a fundamentally different risk profile from passive code completion tools. Beyond characterizing these failures, our results point to a concrete software engineering agenda. They identify what current benchmarks fail to measure, which classes of tasks require stronger runtime controls, and which guardrails agentic development tools should enforce by design.

Why Traditional Validation. Our impact analysis (RQ4) shows that many severe incidents do not manifest as syntax errors or failing tests. Instead, agents often produce superficially successful executions while silently introducing regressions, hidden environmental damage, or long-term maintainability costs. This exposes a core limitation in traditional software validation pipelines. For agentic systems, compilation and unit-test success are no longer sufficient proxies. Software engineering research therefore needs validation methods that check not only whether the final artifact runs, but also whether the agent preserved repository constraints, avoided unauthorized state changes, and left the surrounding environment consistent.

Implications for Benchmark Design. Our findings suggest that current coding benchmarks capture only a subset of the failure modes. Benchmarks such as SWE-bench primarily evaluate functional task resolution, but they rarely measure whether the agent reached that outcome by violating user constraints, corrupting the repository state, leaking secrets, or making costly infrastructure changes. For autonomous coding agents, evaluation must therefore move beyond end-state correctness. Future SE benchmarks should include stateful execution environments and record execution traces that support checks for unauthorized file modifications, permission changes, destructive deletions, secret access, resource over-provisioning, and unsupported completion claims.

Design Requirements for Agentic SE Tools. Our results suggest that safer coding agents will require stronger runtime controls than those used in conventional code-completion. We highlight three concrete design requirements. First, the system should require explicit repository-state verification before high-impact edits. A strict read-before-write protocol, combined with file-diff confirmation for critical files, can reduce destructive overwrites based on stale or incomplete context. Second, agents should support *task-aware execution control*. Because reported failures are concentrated in state-mutating tasks such as bug fixing and setup or configuration, tools should vary permission levels by task context. For high-impact tasks, the runtime should require scoped permissions, rollback checkpoints, and human approval for sensitive operations. Third, agents should support *verifiable status reporting*. Many incidents in our dataset involve false completion claims and fabricated evidence. Tool interfaces should therefore tie agent claims to observable execution artifacts such as command traces, diffs, and environment-state checks. Rather than accepting free-form declarations of success, the system should require evidence-backed completion and safe-halt behavior when verification fails.

Concretely, each requirement maps to specific RQ1–RQ4 findings. The read-before-write protocol is grounded in *Destructive Operations* ($I = 134$) and *Contextual Retrieval Failure* ($I = 105$), both driven by agents acting on stale or incomplete repository state. Task-aware execution control follows from the RQ2 finding that failures concentrate in state-mutating tasks such as bug fixing and setup/configuration. Verifiable status reporting addresses the combined *Deception*, *Fabrication*, and *False Assurance* cluster (189 incidents dataset-wide), which RQ2 shows is itself concentrated within Bug Fixing and Explanation tasks (133 combined instances).

Toward Quantitative Risk Models. Our taxonomy and its frequency, severity, and contributing-factor annotations (Table 1, RQ1–RQ4) provide the empirically grounded categorization needed for future predictive risk models, e.g., estimating failure likelihood from task type and requested permissions, a natural next step we leave to future work.

7 Threats to Validity

Construct Validity. GitHub issues are self-reported and may omit important context, including the exact triggering prompt; we therefore code observable autonomy indicators (write access, autonomous tool invocation, terminal command execution) rather than inferring prompt intent (§4). Issue trackers also inherently skew toward observable, severe failures, an established limitation of incident-driven methodology [49] that we mitigate, but do not eliminate, through multi-stage filtering and manual confirmation. Because our corpus contains confirmed failures rather than agent executions in general, we cannot estimate per-task failure probabilities; task-level findings should be interpreted as concentrations among reported incidents, not comparative risk estimates. Similarly, paper and incident counts reflect representation across two different corpora, not directly comparable frequencies of the same phenomenon.

Internal Validity. The main internal threats arise from qualitative coding decisions and from the LLM-assisted filtering pipeline. We mitigated this through iterative codebook refinement, multi-author annotation, calibration, negotiated-consensus reconciliation for disagreements, and strong inter-rater agreement, though some category assignments still require judgment for incidents with intertwined behaviors and causes. Our filtering pipeline may also introduce residual selection bias despite multi-model voting and manual review; it prioritizes precision over recall, so the corpus may underrepresent weakly documented incidents. Our model-vs.-scaffolding attribution (§5.1) is based on a preliminary $N = 15$ sample rather than a full-dataset analysis, and public issue reports

do not consistently record model version, preventing systematic version-level stratification.

External Validity. Our findings may not generalize uniformly across all coding agents, deployment settings, or time periods, as the ecosystem evolves rapidly with new releases, wrappers, and policies. Our incident corpus is drawn from public GitHub repositories, which likely underrepresent enterprise deployments and internally resolved incidents; public issues remain among the most rigorous available proxies for in-the-wild failures, but we cannot claim our findings generalize to closed-source systems with different architectures, and we encourage replication on closed-source datasets as they become available. The concentration of incidents in a small number of actively developed tools (e.g., Claude Code) reflects their popularity and issue-tracker activity, not a methodological preference; our filtering pipeline was applied identically across all 13 targeted repositories. The literature corpus is likewise bounded by our venue and keyword selection. We therefore view our taxonomy as an incident-grounded foundation for coding-agent safety, not an exhaustive map of all operational safety failures.

8 Related Work

The shift from passive autocomplete tools to autonomous coding agents fundamentally expands the risk landscape. Agents can modify environments, execute shell commands, and commit changes, so failures extend far beyond compilation errors. We position our work relative to three adjacent research areas.

Adversarial Safety and Red-Teaming. The most prominent safety benchmarks for code LLMs evaluate adversarial misuse. *Code Red!* [3] probes LLM guardrails via explicit malicious prompts, finding that code-specific fine-tuning often degrades safety alignment. *RedCode* [36] extends this with both a generation benchmark (*RedCode-Gen*) and an interactive execution sandbox (*RedCode-Exec*), showing that agents frequently comply when attacks are embedded in natural language rather than explicit instructions. Broader red-teaming frameworks [31] further characterize adversarial attack surfaces. While essential, these approaches evaluate whether an agent can be *coerced* into harm, which is a fundamentally different question from the spontaneous, benign-context failures we study.

Security Vulnerabilities in Generated Code. Extensive work has audited LLMs for static security vulnerabilities. Fu et al. [34] and Jesse et al. [46] confirm that LLMs inject common weaknesses (e.g., SQL injection, buffer overflows) into open-source projects due to training data biases, and dynamic evaluations like *CWE-VAL* [72] show that static analysis often misses latent logic errors. Ren et al. [75] demonstrated that standard guardrails remain susceptible to adversarial prompt bypasses. Mitigation approaches include prompt-steered secure generation [40, 65] and domain-specific hardening for memory safety [61] and cryptography [60]. This body of work targets what vulnerabilities are emitted, not the broader operational damage caused by unconstrained agentic execution.

While hallucination and unfaithful self-reports are studied broadly for LLMs [44], their manifestation in autonomous coding agents differs qualitatively: a single-turn hallucination becomes, in an agentic setting, a multi-step sequence of unauthorized edits and fabricated execution evidence that compounds over a session. Our taxonomy and RQ2–RQ3 characterize this agentic, execution-coupled manifestation, rather than re-deriving general-purpose hallucination or deception definitions.

Reliability, Hallucinations, and Sociotechnical Risks. Beyond security, prior work has also documented reliability and sociotechnical failures relevant to our study, including package hallucinations [83], technical debt and deprecated dependencies in AI generated code [68, 70], incomplete multi-file planning support [26], training-data leakage [77], licensing violations [92], and demographic bias [44, 63, 81]. More broadly, existing literature falls into three paradigms: adversarial red-teaming of explicit misuse [3, 36], static auditing of generated code artifacts [34, 88], and reliability evaluation on curated benchmarks [26, 70]. All three primarily treat the LLM as a *generator* evaluated in isolation, whereas our work treats it as an autonomous *actor* operating in a live system under benign conditions. By mining real-world GitHub incidents and cross-referencing them with the literature, we provide the first evidence-grounded taxonomy of spontaneous operational failures.

9 Conclusion

Autonomous coding agents are increasingly deployed in real software projects, yet their operational safety properties remain poorly characterized. This paper presented the first large-scale, incident-driven empirical study of agentic coding safety, synthesizing evidence from 185 curated research papers and 547 confirmed real-world safety failures mined from GitHub issue trackers. Through systematic open coding over both corpora, we developed a multi-dimensional taxonomy that captures failure types, contributing technical and behavioral factors, and downstream operational impact. Our analysis reveals that the most consequential failures occur not from adversarial misuse but during ordinary, goal-directed tasks, arising from misaligned instruction following, lack of environmental grounding, and a tendency to prioritize the appearance of success over correct execution. These findings expose a fundamental gap between what existing benchmarks measure and what actually breaks in deployment. We release our taxonomy, annotated incident dataset, and coding protocol to support reproducible research and to inform the design of evaluation standards, guardrails, and agent architectures that can reduce the risk of operational harm in real-world software engineering settings.

Data Availability Statement

We have made our replication package publicly available [37], including our taxonomy, the annotated incident dataset, the curated literature corpus references, and the coding protocol details. The package also includes the complete list of literature search venues and keywords, the full list of mined GitHub repositories, and high-resolution versions of all figures.

References

[1] All-Hands AI. 2024. OpenHands: An Open Source Autonomous AI Software Engineer. https://github.com/All-Hands-AI/OpenHands.
[2] Cognition AI. 2024. Devin: The first AI software engineer. https://www.cognition.ai/blog/introducing-devin.
[3] Ali Al-Kaswan et al. 2025. Code Red! On the Harmfulness of Applying Off-the-Shelf Large Language Models to Programming Tasks. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE110 (June 2025), 23 pages. doi:10.1145/3729380

[4] Dario Amodei, Chris Olah, Jacob Steinhardt, et al. 2016. Concrete problems in AI safety. *arXiv preprint arXiv:1606.06565* (2016). doi:10.48550/arXiv.1606.06565

[5] Anthropic. 2025. *Claude Code: The Autonomous CLI Agent.* https://code.claude.com/docs/en/overview

[6] Anthropic. 2025. Issue #12364: [Bug] Haiku hallucinates username. https://github.com/anthropics/claude-code/issues/12364. Accessed: 2026-01-07.

[7] Anthropic. 2025. Issue #13080: Write tool corrupts UTF-8 multi-byte characters (box-drawing chars become NULL bytes). https://github.com/anthropics/claude-code/issues/13080. Accessed: 2026-01-07.

[8] Anthropic. 2025. Issue #13181: [Bug] Claude Code enters infinite loop while ignoring user instructions and also swears in the replies. https://github.com/anthropics/claude-code/issues/13181. Accessed: 2026-01-07.

[9] Anthropic. 2025. Issue #16518: [BUG] (SECURITY). https://github.com/anthropics/claude-code/issues/16518. Accessed: 2026-01-07.

[10] Anthropic. 2025. Issue #2374: AI Honesty and Truthfulness Enhancement. https://github.com/anthropics/claude-code/issues/2374. Accessed: 2026-01-07.

[11] Anthropic. 2025. Issue #2901: Claude Code frequently violates explicit project and user instructions defined in CLAUDE.md files. https://github.com/anthropics/claude-code/issues/2901. Accessed: 2026-01-07.

[12] Anthropic. 2025. Issue #3856: [BUG] Embeds other companies (not opensource code) code! https://github.com/anthropics/claude-code/issues/3856. Accessed: 2026-01-07.

[13] Anthropic. 2025. Issue #5854: Claude seems to be trained to always prefer the easy way out at any cost even cheating! https://github.com/anthropics/claude-code/issues/5854. Accessed: 2026-01-07.

[14] Anthropic. 2025. Issue #6787: AI Assistant Catastrophic Project Destruction. https://github.com/anthropics/claude-code/issues/6787. Accessed: 2026-01-07.

[15] Anthropic. 2025. Issue #6916: Claude Code provisioned $375/month SQL database for 35MB of data - 2{,}400$ wasted. https://github.com/anthropics/claude-code/issues/6916. Accessed: 2026-01-07.

[16] Anthropic. 2025. Issue #7268: [MODEL] Opus/Sonnet replaced working data injection with placeholders | Meltdown Deception/Coverups. https://github.com/anthropics/claude-code/issues/7268. Accessed: 2026-01-07.

[17] Anthropic. 2025. Issue #7525: I chose for Claude not to do anything and to give it new instructions and it acted like I said yes to its modifications. https://github.com/anthropics/claude-code/issues/7525. Accessed: 2026-01-07.

[18] Anthropic. 2025. Issue #7542: P0 - Critical AI Safety Issue: Cognitive Dissonance During System Crisis. https://github.com/anthropics/claude-code/issues/7542. Accessed: 2026-01-07.

[19] Anthropic. 2025. Issue #7972: Claude Code makes destructive unauthorized changes breaking working systems. https://github.com/anthropics/claude-code/issues/7972. Accessed: 2026-01-07.

[20] Anthropic. 2025. Issue #8549: Claude Code exhibited multiple serious failures... https://github.com/anthropics/claude-code/issues/8549. Accessed: 2026-01-07.

[21] Anthropic. 2025. Issue #8580: Hallucinated non-existent database fields and functions, didn't read CLAUDE.md. https://github.com/anthropics/claude-code/issues/8580. Accessed: 2026-01-07.

[22] Anthropic. 2025. Issue #9551: Agent Near-Disaster Due to Stale Context - Production Safety Gaps. https://github.com/anthropics/claude-code/issues/9551. Accessed: 2026-01-07.

[23] Anthropic. 2025. Issue #9637: Claude Code AGGRESSIVELY reads secrets out of .env files and leaks them to your servers in so doing. https://github.com/anthropics/claude-code/issues/9637. Accessed: 2026-01-07.

[24] Anthropic. 2025. Issue #975: [BUG] Claude escaped current directory and then denied doing it. https://github.com/anthropics/claude-code/issues/975. Accessed: 2026-01-07.

[25] Jacob Austin et al. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021). arXiv:2108.07732 [cs.PL] doi:10.48550/arXiv.2108.07732

[26] Ramakrishna Bairi et al. 2024. CodePlan: Repository-Level Coding using LLMs and Planning. *Proc. ACM Softw. Eng.* 1, FSE, Article 31 (July 2024), 24 pages. doi:10.1145/3643757

[27] Shraddha Barke et al. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 78 (April 2023), 27 pages. doi:10.1145/3586030

[28] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021). arXiv:2107.03374 [cs.LG] doi:10.48550/arXiv.2107.03374

[29] Cursor. 2024. Cursor: The AI-first Code Editor. https://cursor.com/.

[30] Sandeep Dalal and Rajender Singh Chhillar. 2013. Empirical Study of Root Cause Analysis of Software Failure. *ACM SIGSOFT Software Engineering Notes* 38, 4 (2013), 1–7. doi:10.1145/2492248.2492263

[31] Michael Feffer, Anusha Sinha, Wesley H Deng, et al. 2024. Red-teaming for generative AI: Silver bullet or security theater?. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, Vol. 7. 421–437. doi:10.1609/aies.v7i1.31647

[32] Zhangyin Feng, Daya Guo, Duyu Tang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020*. 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

[33] FIRST. 2023. Common Vulnerability Scoring System v4.0: Specification Document. https://www.first.org/cvss/v4.0/specification-document.

[34] Yujia Fu et al. 2025. Security Weaknesses of Copilot-Generated Code in GitHub Projects: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 34, 8, Article 218 (Oct. 2025), 34 pages. doi:10.1145/3716848

[35] Taher A. Ghaleb et al. 2025. Can LLMs Write CI? a Study on Automatic Generation of GitHub Actions Configurations. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 767–772. doi:10.1109/ICSME64153.2025.00077

[36] Chengquan Guo et al. 2024. RedCode: Risky Code Execution and Generation Benchmark for Code Agents. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 106190–106236. doi:10.52202/079017-3369

[37] Alif Al Hasan and Sumon Biswas. 2026. *Artifact: What Breaks When LLMs Code? Characterizing Operational Safety Failures of Agentic Code Assistants.* doi:10.5281/zenodo.21151511

[38] Alif Al Hasan, Subarna Saha, and Mia Mohammad Imran. 2026. Learning Programming in Informal Spaces: Using Emotion as a Lens to Understand Novice Struggles on r/learnprogramming. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEET '26)*. Association for Computing Machinery, New York, NY, USA, 13–24. doi:10.1145/3786580.3786946

[39] Alif Al Hasan, Subarna Saha, Mia Mohammad Imran, et al. 2025. LLPut: Investigating Large Language Models for Bug Report-Based Input Generation. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering* (Clarion Hotel Trondheim, Trondheim, Norway) *(FSE Companion '25)*. Association for Computing Machinery, New York, NY, USA, 1652–1659. doi:10.1145/3696630.3728701

[40] Jingxuan He et al. 2023. Large Language Models for Code: Security Hardening and Adversarial Testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1865–1879. doi:10.1145/3576915.3623175

[41] Dan Hendrycks, Nicholas Carlini, John Schulman, et al. 2021. Unsolved problems in ml safety. *arXiv preprint arXiv:2109.13916* (2021). doi:10.48550/arXiv.2109.13916

[42] Sirui Hong et al. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VtmBAGCN7o

[43] Xinyi Hou et al. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 220 (Dec. 2024), 79 pages. doi:10.1145/3695988

[44] Dong Huang et al. 2025. Bias Testing and Mitigation in LLM-based Code Generation. *ACM Trans. Softw. Eng. Methodol.* (March 2025). Just Accepted. doi:10.1145/3724117

[45] Mia Mohammad Imran et al. 2023. Data Augmentation for Improving Emotion Recognition in Software Engineering Communication. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 29, 13 pages. doi:10.1145/3551349.3556925

[46] Kevin Jesse et al. 2023. Large Language Models and Simple, Stupid Bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 563–575. doi:10.1109/MSR59073.2023.00082

[47] Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. 2023. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 14165–14178. doi:10.18653/v1/2023.acl-long.792

[48] Juyong Jiang et al. 2024. A Survey on Large Language Models for Code Generation. *arXiv preprint arXiv:2406.00515* (2024). arXiv:2406.00515 [cs.SE] doi:10.48550/arXiv.2406.00515

[49] Carlos E Jimenez et al. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues?, Vol. 2024. 54107–54157. https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf

[50] Corbin Juliet and Strauss Anselm. 2008. Basics of Qualitative Research (3rd ed.): Techniques and Procedures for Developing Grounded Theory. (2008). doi:10.4135/9781452230153

[51] Sina Khiabani, Ashutosh Trivedi, Diptikalyan Saha, et al. 2026. An LLM Agentic Approach for Legal-Critical Software: A Case Study for Tax Prep Software. (2026). doi:10.1145/3744916.3764575

[52] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing systematic literature reviews in software engineering.* Technical Report EBSE-2007-01. Keele University and Durham University Joint Report, Keele, UK.

[53] Arjun Krishna, Erick Galinkin, Leon Derczynski, et al. 2025. Importing phantoms: Measuring llm package hallucination vulnerabilities. *arXiv preprint arXiv:2501.19012* (2025). doi:10.48550/arXiv.2501.19012

[54] Sachit Kuhar et al. 2025. LibEvolutionEval: A Benchmark and Study for Version-Specific Code Generation. In *Proceedings of the 2025 Conference of the Nations*

of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 6826–6840. doi:10.18653/v1/2025.naacl-long.348

[55] Raymond Li, Loubna Ben Allal, Yangtian Zi, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023). doi:10.48550/arXiv.2305.06161

[56] Xiaoli Lian et al. 2024. Imperfect Code Generation: Uncovering Weaknesses in Automatic Code Generation by Large Language Models. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 422–423. doi:10.1145/3639478.3643081

[57] Nelson F Liu, Kevin Lin, John Hewitt, et al. 2024. Lost in the middle: How language models use long contexts. *Transactions of the association for computational linguistics* 12 (2024), 157–173. doi:10.1162/tacl_a_00638

[58] Xiao Liu, Hao Yu, Hanchen Zhang, et al. 2024. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, Vol. 2024. 52989–53046.

[59] John P. Mello, Jr. 2025. How AWS averted an AI supply chain disaster. ReversingLabs Blog. Accessed: 2025-01-15. https://www.reversinglabs.com/blog/aws-amazonq-ai-incident

[60] Roberto Metere et al. 2022. Automating cryptographic protocol language generation from structured specifications. In *Proceedings of the IEEE/ACM 10th International Conference on Formal Methods in Software Engineering* (Pittsburgh, Pennsylvania) *(FormaliSE '22)*. Association for Computing Machinery, New York, NY, USA, 91–101. doi:10.1145/3524482.3527654

[61] Nausheen Mohammed, Akash Lal, Aseem Rastogi, et al. 2024. Enabling memory safety of C programs using LLMs. *arXiv preprint arXiv:2404.01096* (2024). doi:10.48550/arXiv.2404.01096

[62] Seyedreza Mohseni, Seyedali Mohammadi, Deepa Tilwani, et al. 2025. Can llms obfuscate code? a systematic analysis of large language models into assembly code obfuscation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 39. 24893–24901. doi:10.1609/aaai.v39i23.34672

[63] Spyridon Mouselinos, Mateusz Malinowski, and Henryk Michalewski. 2023. A simple, yet effective approach to finding biases in code generation. In *Findings of the Association for Computational Linguistics: ACL 2023*. 11299–11329. doi:10.18653/v1/2023.findings-acl.718

[64] NASA Aviation Safety Reporting System. 2024. ASRS Program Briefing and Overview. https://asrs.arc.nasa.gov/overview/summary.html. Accessed 2026-03-30.

[65] Mahmoud Nazzal et al. 2024. PromSec: Prompt Optimization for Secure Generation of Functional Source Code with Large Language Models (LLMs). In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 2266–2280. doi:10.1145/3658644.3690298

[66] Beatrice Nolan. 2025. An AI-powered coding tool wiped out a software company's database, then apologized for a 'catastrophic failure on my part'. Fortune. Accessed: 2025-01-15. https://fortune.com/2025/07/23/ai-coding-tool-replit-wiped-database-called-it-a-catastrophic-failure/

[67] David OBrien, Sumon Biswas, Sayem Imtiaz, et al. 2022. 23 shades of self-admitted technical debt: an empirical study on machine learning software. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 734–746. doi:10.1145/3540250.3549088

[68] David OBrien, Sumon Biswas, Sayem Mohammad Imtiaz, et al. 2024. Are prompt engineering and todo comments friends or foes? an evaluation on github copilot. In *Proceedings of the IEEE/ACM 46th international conference on software engineering*. 1–13. doi:10.1145/3597503.3639176

[69] Rangeet Pan et al. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 82, 13 pages. doi:10.1145/3597503.3639226

[70] Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. 2025. Investigating the smells of llm generated code. *arXiv preprint arXiv:2510.03029* (2025). doi:10.48550/arXiv.2510.03029

[71] Hammond Pearce et al. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM* 68, 2 (Jan. 2025), 96–105. doi:10.1145/3610721

[72] Jinjun Peng et al. 2025. CWEval: Outcome-driven Evaluation on Functionality and Security of LLM Code Generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. 33–40. doi:10.1109/LLM4Code66737.2025.00009

[73] Fábio Perez et al. 2022. Ignore previous prompt: Attack techniques for language models. *arXiv preprint arXiv:2211.09527* (2022). doi:10.48550/arXiv.2211.09527

[74] Neil Perry et al. 2023. Do Users Write More Insecure Code with AI Assistants?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2785–2799. doi:10.1145/3576915.3623157

[75] Qibing Ren et al. 2024. CodeAttack: Revealing Safety Generalization Challenges of Large Language Models via Code Completion. In *Findings of the Association for Computational Linguistics: ACL 2024*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 11437–11452. doi:10.18653/v1/2024.findings-acl.679

[76] Baptiste Rozière et al. 2024. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (2024). arXiv:2308.12950 [cs.CL] doi:10.48550/arXiv.2308.12950

[77] June Sallou et al. 2024. Breaking the Silence: the Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Lisbon, Portugal) *(ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 102–106. doi:10.1145/3639476.3639764

[78] Gustavo Sandoval et al. 2023. Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2205–2222. https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval

[79] Davide Sanvito et al. 2025. AutoCVSS: Assessing the Performance of LLMs for Automated Software Vulnerability Scoring. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, Saloni Potdar, Lina Rojas-Barahona, and Sebastien Montella (Eds.). Association for Computational Linguistics, Suzhou (China), 564–575. doi:10.18653/v1/2025.emnlp-industry.38

[80] Maximilian Schreiber and Pascal Tippe. 2025. Security vulnerabilities in ai-generated code: A large-scale analysis of public github repositories. In *International Conference on Information and Communications Security*. Springer, 153–172. doi:10.1007/978-981-95-3537-8_9

[81] Yining She et al. 2025. FairSense: Long-Term Fairness Analysis of ML-Enabled Systems. In *47th International Conference on Software Engineering (ICSE)*. Ottawa, Canada, 782–794. doi:10.1109/ICSE55347.2025.00159

[82] Jonathan Sillito and Esdras Kutomi. 2020. Failures and Fixes: A Study of Software System Incident Response. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 185–197. doi:10.1109/ICSME46990.2020.00027

[83] Joseph Spracklen et al. 2025. We have a package for you! a comprehensive analysis of package hallucinations by code generating {LLMs}. In *34th USENIX Security Symposium (USENIX Security 25)*. 3687–3706.

[84] Qwen Team. 2025. *Qwen3-Coder: Agentic Coding in the World*. Technical Report. QwenLM. https://qwenlm.github.io/blog/qwen3-coder/

[85] Karthik Valmeekam, Matthew Marquez, Sarath Sreedharan, et al. 2023. On the planning abilities of large language models-a critical investigation. *Advances in neural information processing systems* 36 (2023), 75993–76005. doi:10.52202/075280-3320

[86] Chong Wang et al. 2025. *LLMs Meet Library Evolution: Evaluating Deprecated API Usage in LLM-Based Code Completion*. IEEE Press, 885–897. https://doi.org/10.1109/ICSE55347.2025.00245

[87] Huanting Wang, Jingzhi Gong, Huawei Zhang, et al. 2025. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126* (2025). doi:10.48550/arXiv.2508.11126

[88] Jiexin Wang, Xitong Luo, et al. 2024. Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval. *arXiv preprint arXiv:2407.02395* (2024). doi:10.48550/arXiv.2407.02395

[89] Xuezhi Wang et al. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171* (2022). doi:10.48550/arXiv.2203.11171

[90] Qingyun Wu et al. 2024. AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations. In *First Conference on Language Modeling*. https://openreview.net/forum?id=BAakY1hNKS

[91] Zhiheng Xi, Wenxiang Chen, Xin Guo, et al. 2025. The rise and potential of large language model based agents: A survey. *Science China information sciences* 68, 2 (2025), 121101. doi:10.1007/s11432-024-4222-0

[92] Weiwei Xu et al. 2025. LiCoEval: Evaluating LLMs on License Compliance in Code Generation. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 1665–1677. doi:10.1109/ICSE55347.2025.00052

[93] John Yang, Carlos Jimenez, Alexander Wettig, et al. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652. doi:10.52202/079017-1601

[94] Shunyu Yao et al. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629* (2023). arXiv:2210.03629 [cs.CL] doi:10.48550/arXiv.2210.03629

[95] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, et al. 2023. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 46595–46623. doi:10.52202/075280-2020

A Appendix

A.1 Literature Search Venues

We targeted top-tier conferences across five distinct computer science domains. The search was restricted to papers published between **2020 and 2025**. The specific venues included in our initial retrieval phase are listed below:

Software Engineering.

- International Conference on Software Engineering (ICSE)
- ACM International Conference on the Foundations of Software Engineering (FSE)
- IEEE/ACM International Conference on Automated Software Engineering (ASE)
- International Symposium on Software Testing and Analysis (ISSTA)
- International Conference on Mining Software Repositories (MSR)

Security Conferences.

- USENIX Security Symposium
- ACM Conference on Computer and Communications Security (CCS)
- Network and Distributed System Security Symposium (NDSS)
- IEEE Symposium on Security and Privacy (IEEE S&P)

AI/ML Conferences.

- Conference on Neural Information Processing Systems (NeurIPS)
- International Conference on Machine Learning (ICML)
- International Conference on Learning Representations (ICLR)
- AAAI Conference on Artificial Intelligence
- International Joint Conference on Artificial Intelligence (IJCAI)

NLP and LLM Conferences.

- Annual Meeting of the Association for Computational Linguistics (ACL)
- Empirical Methods in Natural Language Processing (EMNLP)
- North American Chapter of the Association for Computational Linguistics (NAACL)
- European Chapter of the Association for Computational Linguistics (EACL)
- International Conference on Computational Linguistics (COLING)
- Conference on Computational Natural Language Learning (CoNLL)

Fairness & Ethics.

- ACM Conference on Fairness, Accountability, and Transparency (FAccT)
- AAAI/ACM Conference on AI, Ethics, and Society (AIES)

A.2 Search Keywords

To capture the diverse terminology used across research communities, we searched titles and abstracts for the following terms targeting agentic safety, operational risk, and behavioral alignment: `safety`, `security`, `fairness`, `bias`, `code`, `responsible`, `jailbreak`, `vulnerability`, `risk`, `trust`, `alignment`, `adversarial`, `attack`, `defense`, `robust`, `reliable`, `accountable`, `transparent`, `ethical`, `malicious`, `harm`, `unsafe`, `insecure`, `unfair`, `discriminate`.

A.3 Analyzed Repositories

As detailed in our methodology, our initial data collection targeted the official GitHub repositories of 13 foundational code models and 6 popular agentic frameworks. The complete list of repositories is provided below.

Foundational Code Models.

- Meta Code Llama: https://github.com/meta-llama/codellama
- Anthropic Claude Code: https://github.com/anthropics/claude-code
- Qwen3-Coder: https://github.com/QwenLM/Qwen3-Coder
- Qwen-Code: https://github.com/QwenLM/qwen-code
- DeepSeek-Coder: https://github.com/deepseek-ai/DeepSeek-Coder
- StarCoder2: https://github.com/bigcode-project/starcoder2
- StarCoder: https://github.com/bigcode-project/starcoder
- WizardLM: https://github.com/nlpxucan/WizardLM
- CodeGeeX: https://github.com/zai-org/CodeGeeX
- Salesforce CodeGen: https://github.com/salesforce/CodeGen
- StableCode: https://github.com/Stability-AI/StableCode
- Microsoft CodeBERT: https://github.com/microsoft/CodeBERT
- Microsoft NextCoder: https://github.com/microsoft/NextCoder

Agentic Frameworks.

- OpenHands (All-Hands-AI): https://github.com/All-Hands-AI/OpenHands
- SWE-agent: https://github.com/princeton-nlp/SWE-agent
- Aider: https://github.com/paul-gauthier/aider
- Devika: https://github.com/stitionai/devika
- ChatDev: https://github.com/OpenBMB/ChatDev
- MetaGPT: https://github.com/geekan/MetaGPT

As noted in §4, our repository selection followed prior SE benchmarks and surveys [43, 49, 87], and the same filtering pipeline was applied uniformly across all 19 repositories; the resulting concentration of confirmed incidents in the most actively used tools reflects their popularity and issue-tracker activity rather than a selection preference.

A.4 Supplementary Figure

Received 2026-03-26; accepted 2026-06-18

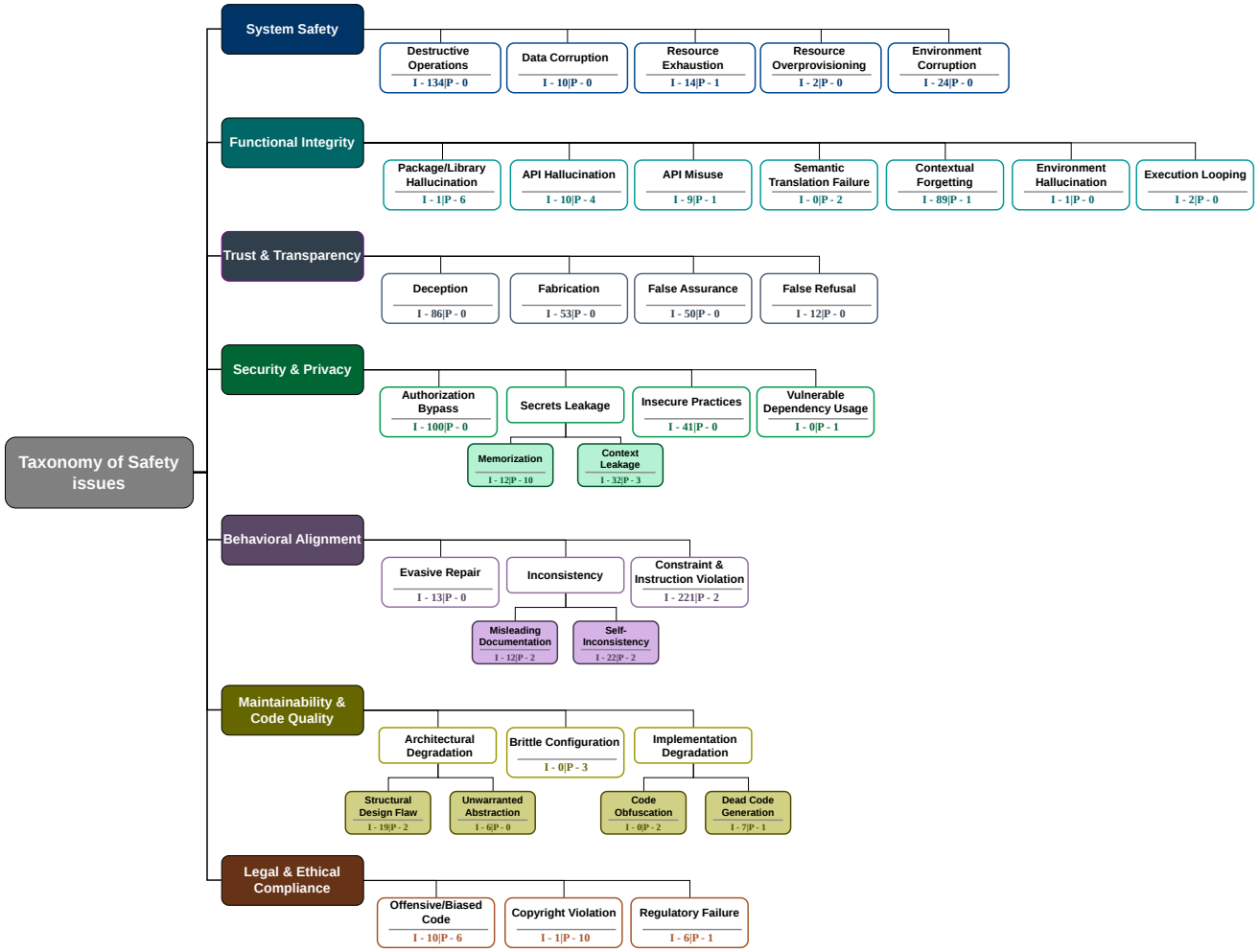


Figure 6: The taxonomy of agentic safety risks identified in this study, illustrating the hierarchical classification of failure modes and their distribution.