Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild

Yue Liu, Ratnadira Widyasari, Yanjie Zhao, Ivana Clairine Irsan, Junkai Chen, and David Lo

***Abstract*—AI coding assistants are now widely used in software development. Software developers increasingly integrate AI-generated code into their codebases to improve productivity. Prior studies have shown that AI-generated code may contain code quality issues under controlled settings. However, we still know little about the real-world impact of AI-generated code on software quality and maintenance after it is introduced into production repositories. In other words, it remains unclear whether such issues are quickly fixed or persist and accumulate over time as technical debt. In this paper, we conduct a large-scale empirical study on the technical debt introduced by AI coding assistants in the wild. To achieve that, we built a dataset of 302.6k verified AI-authored commits from 6,299 GitHub repositories, covering five widely used AI coding assistants. For each commit, we run static analysis before and after the change to precisely attribute which code smells, correctness issues, and security issues the AI introduced. We then track each introduced issue from the introducing commit to the latest repository revision to study its lifecycle. Our results show that we identified 484,366 distinct issues, and that code smells are by far the most common type, accounting for 89.3% of all issues. We also find that more than 15% of commits from every AI coding assistant introduce at least one issue, although the rates vary across tools. More importantly, 22.7% of tracked AI-introduced issues still survive at the latest version of the repository. These findings show that AI-generated code can introduce long-term maintenance costs into real software projects and highlight the need for stronger quality assurance in AI-assisted development.**

***Index Terms*—AI coding assistants, technical debt, code quality, software maintenance, empirical study**

I. Introduction

Through AI coding assistants (e.g., Cursor, Claude Code), software developers can now describe what they want in natural language and get working code back in seconds. They significantly improve development productivity. Thus, AI is becoming standard equipment for modern software developers [1]. According to the 2025 Stack Overflow Developer Survey, 84% of professional developers are using or are planning to use AI coding tools within their development processes [2]. The AI-generated code is also widely used in real-world software projects. For example, both Google and Microsoft disclosed in 2025 that AI now writes over 20% of their new code [3], [4]. Similarly, GitHub reported that more than 1.1 million public repositories used AI coding tools between 2024 and 2025 [1]. Overall, AI-generated code has graduated from experiment to production reality, and it is everywhere.

Although AI coding assistants have proven effective at generating functional programs, many previous research studies have revealed a range of quality concerns in AI-generated code. Recent studies have shown that AI-generated code suffers from functional bugs, runtime errors, and systemic maintainability issues [5], [6]. Also, the code produced by AI coding tools poses security risks [7]–[9]. Pearce *et al.* [7] found that about 40% of AI-generated code in security-sensitive contexts contains critical vulnerabilities. However, recent research [8], [10] has found that developers tend to place excessive trust in the quality of AI-generated code, blindly accepting the code without proper validation. As a result, these unverified code snippets are merged into production codebases. Over time, this can cause a considerable accumulation of **technical debt**, which can be costly and time-consuming to address [11], [12].

Recognizing these risks, recent empirical studies have started to investigate AI-generated code in real-world repositories. These studies cover a range of practical concerns, such as security weaknesses [13], [14], project-level development velocity [15], pull request acceptance rates [16], and code redundancy [17]. For example, He *et al.* [15] found that Cursor adoption in 807 GitHub repositories led to a transient velocity boost but persistent increases in code complexity. However, existing studies still have several limitations. First, most studies focus on a single tool or a narrow set of tools, which limits the generalizability of their findings. Second, many studies do not identify AI-generated code at the commit level. Instead, they rely on project-level adoption signals, such as the presence of AI tool configuration files in a repository (e.g., *.cursorrules* or *.claude/*). These signals suggest possible AI use, but do not directly show which commits or files were generated by AI. Third, they mostly evaluate code at a single point in time, failing to capture the long-term lifecycle of the code. Finally, in real-world repositories, AI-assisted code and human-written code are often mixed together, and AI usage may leave no reliable trace. Thus, it remains unknown how AI-generated code actually ages in production. We still do not know whether the technical debt it introduces persists, gets refactored, or silently accumulates over time.

To bridge this gap, we conduct a large-scale empirical study to investigate the lifecycle of AI-introduced technical debt in the wild. Our approach consists of three steps. First, we build a large dataset of verified AI-authored commits from five popular coding assistants (i.e., GitHub Copilot, Claude, Cursor, Gemini, and Devin) across over 6,000 GitHub

Yue Liu, Ratnadira Widyasari, Ivana Clairine Irsan, Junkai Chen, and David Lo are with Singapore Management University, Singapore. E-mail: liuyue@smu.edu.sg, ratnadiraw@smu.edu.sg, ivanairsan@smu.edu.sg, junkaichen@smu.edu.sg, davidlo@smu.edu.sg.

Yanjie Zhao is with Huazhong University of Science and Technology, Wuhan, China. E-mail: yanjie_zhao@hust.edu.cn.

repositories. We use explicit Git metadata to identify commits generated by AI coding tools across thousands of GitHub repositories. This design is analogous in spirit to research on self-admitted technical debt (SATD), which studies the subset of technical debt that developers explicitly document rather than the full universe of debt [18], [19]. Similarly, we focus on the subset of AI-assisted contributions whose AI involvement is explicitly visible in Git metadata. Although this does not cover all AI-assisted changes, it provides a more reliable basis for attribution and large-scale empirical analysis. Second, we perform a commit-level quality analysis. For each AI-authored commit, we run static analysis tools on the source code immediately before and after the change. This allows us to precisely identify which code smells, correctness issues, and security issues the AI introduced or fixed. Third, we conduct a debt lifecycle analysis. We track each introduced issue to the latest repository revision (`HEAD`) to determine whether it still survives or has been resolved. Through this design, our study provides the first comprehensive view of how AI-generated code ages in real-world software.

Contribution. To the best of our knowledge, this paper is the first to:

- Conduct a large-scale empirical study of AI-introduced technical debt across five major AI coding assistants (i.e., GitHub Copilot, Claude, Cursor, Gemini, and Devin) and over 6,000 real-world GitHub repositories.
- Perform commit-level differential analysis to precisely attribute code smells, correctness issues, and security issues to individual AI-authored commits.
- Track AI-introduced issues from the introducing commit to the latest repository revision, revealing whether it persists or gets resolved over time.

Open Science. To support the open science initiative, we publish the studies dataset and a replication package, which is publicly available in GitHub.[1]

Paper Organization. Section II presents the background and motivation. Section III describes our approach. Section IV presents the experimental setup. Section V presents the results. Section VI discusses the findings and their implications. Section VII presents the related work. Section VIII discloses the threats to validity. Section IX draws the conclusions.

II. Background

A. AI Coding Assistants

Modern AI coding assistants (e.g., Cursor, Claude Code, GitHub Copilot) are now deeply embedded in software development workflows. They help software developers write, modify, explain, test, or debug code using natural-language instructions and code context. Advanced agentic tools can now process whole functions, files, or even repositories to autonomously create pull requests with minimal human intervention. Driven by these improved capabilities, AI-generated code is entering production codebases at unprecedented speed and scale. According to GitHub, over 1.1 million public repositories adopted AI coding tools between 2024 and 2025 [1]. As

[1]https://github.com/yueyueL/tech-debt-ai-coding

Fig. 1: Contributor statistics for the Anthropic `claudes-c-compiler` repository.

shown in Figure 1, in Anthropic's `claudes-c-compiler` repository [20], Claude appears as the top contributor, with 3,957 commits and nearly 500K lines of code added within just a few weeks. At this volume and speed, it is unlikely that all AI-generated code receives a thorough human review. This makes it increasingly important to understand the long-term implications of AI-generated code on software quality and maintenance.

In controlled (lab) settings [5], [7], the provenance of code is usually clear since researchers can generate the code directly. However, it is different in real-world codebases. AI-generated code and human-written code are often interleaved during development, and AI usage is not always explicitly recorded in the repository history. This makes it harder to attribute code changes reliably and to observe the long-term impact of AI-generated code after it is merged into production codebases.

B. Technical Debt

Technical debt refers to design or implementation choices that prioritize short-term speed over long-term quality [21]. These shortcuts may help in the short term, but they increase the future cost of maintaining and evolving the software [22], [23]. Its costs can accumulate over time if not addressed. This concern becomes more important as AI coding assistants are widely adopted. AI coding assistants help developers work faster and produce more code. Previous studies have shown that AI-generated code contains code smells, correctness issues, and security vulnerabilities [5], [7], [13]. In this study, we focus on these three categories as code-level technical debt: code smells that reduce maintainability, correctness issues that affect program behavior, and security issues that may expose systems to risk. When such issues are accepted into production repositories, they can accumulate as technical debt in the codebase.

C. Motivation

The technical debt introduced by AI coding assistants is not just a theoretical risk. Such issues are becoming increasingly common in real-world codebases. Figure 2 and 3 show two examples. In Figure 2, a GitHub Copilot-authored commit introduced a `shell=True` subprocess call in `hysteria2.py` [28]. This pattern increases security risk by allowing command injection if user input is involved. A

hysteria2 [24], hysteria2.py:L77, e277daf

```
# issue: shell=True introduces command-injection risk
subprocess.run(
    "systemctl stop hysteria-iptables.service 2>/dev/null
    ",
    shell=True
)
```

⇓

hysteria2 [24], hysteria2.py:L77 d9e392d

```
#fix: ''Improve code security by removing shell=True from
     subprocess calls''
subprocess.run(
    ["systemctl", "disable", "hysteria-iptables.service"
    ],
    stderr=subprocess.DEVNULL
)
```

Fig. 2: Command injection risk introduced by GitHub Copilot in `hysteria2` (1.7K stars) [24] and later removed in a Copilot-assisted fix commit [25].

librealsense [26], test-fps-performance.py:L168, 5535b8a

```
# issue: undefined constant used before definition
time.sleep(DEVICE_INIT_SLEEP_SEC)
```

⇓

librealsense [26], test-fps-performance.py:L37, 14026c8

```
# fix: ''Add missing constants & fixed 6fps bug for live/
    FPS weekly CI''
DEVICE_INIT_SLEEP_SEC = 3
time.sleep(DEVICE_INIT_SLEEP_SEC)
```

Fig. 3: Undefined variables introduced by GitHub Copilot in `librealsense` (8.6K stars) [26], causing a runtime error. A human maintainer committed a fix three weeks later [27].

human developer later fixed it, noting in the commit message: *"Improve code security by removing shell=True from subprocess calls"* [25]. Figure 3 shows a bug from Intel's `librealsense` [26]. In another case, GitHub Copilot replaced a literal value with a named constant, but never defined the constant [29]. The buggy code remained in the repository for over three weeks before the maintainer committed a fix adding the missing definition [27].

These two examples highlight the motivation of our study. AI coding assistants can generate functional code, but they may introduce quality issues into production codebases. Developers also tend to over-trust and accept AI suggestions without thorough review [8], [10]. These issues may be fixed later, or they may persist for a long time, or even create long-term maintenance challenges. Recent studies [5], [6], [13]–[15], [30] have examined the AI-generated code, but they have several limitations. Most prior studies focus on a single tool, a small set of tasks, or controlled settings. For example, Watanabe *et al.* [16] measured the initial acceptance rate of AI-authored code in a single repository. We still know little about the long-term implications of AI-generated code on software quality and maintenance. Thus, we study the technical debt introduced by various AI coding assistants in the wild: how often it is introduced, what kinds of issues appear, and whether those issues still remain in the codebase over time.

III. Approach

Figure 4 provides an overview of our approach. We first collect AI-authored commits from GitHub repositories at scale (Section III-A). We then analyze each AI-authored commit at the code level to determine which quality issues it introduced or fixed (Section III-B). Finally, we track the lifecycle of both the issues and the code itself to determine whether AI-introduced debt persists or gets resolved over time (Section III-C).

A. Data Collection

This step aims to identify candidate GitHub repositories that contain AI-authored commits.

Repository Discovery. We use the GitHub Archive dataset [31], which records public GitHub events (e.g., `PushEvent`), to identify repositories with potential AI-authored code. The dataset is publicly available through Google BigQuery [32], which allows large-scale querying over historical events. We scan `PushEvent` records from January 2024 to October 2025, focusing on repositories with recent development activity. From each event, we extract four metadata fields: actor login, author name, author email, and commit message. We then match these fields against our curated AI-attribution rules (described in the next section) to identify potential AI-authored commits. Only repositories with at least one matching event are retained. To improve coverage of active and popular repositories, we also query the GitHub REST API [33] for top-starred repositories and apply the same attribution rules through full-history repository scanning. This step helps us ensure that repositories with substantial AI-authored activity are not missed during the discovery stage.

AI Attribution Rules. We build attribution rules for widely adopted AI coding tools (e.g., Cursor, GitHub Copilot, Claude Code) identified in the 2025 Stack Overflow Developer Survey [2]. We identify AI-authored commits using explicit signals in Git metadata. Our approach covers AI-authored commits only when the use of an AI coding tool leaves explicit traces in Git metadata. The rules are based on four sources of evidence: (1) actor logins (e.g., copilot-swe-agent[bot]), (2) author emails (e.g., noreply@anthropic.com), (3) author names (e.g., `Cursor Agent`), and (4) `Co-authored-by` trailers in commit messages. These rules rely on explicit machine-readable signals in Git metadata. To finalize the rule list, two authors manually reviewed candidate patterns and verified that they provided reliable evidence of AI coding tool involvement. In total, 29 AI coding tools left identifiable traces in the repositories we collected. The full list of tools and detection rules is included in our replication package.

Full-History Commit Scanning. The discovery stage captures only push-event metadata. However, it provides only partial evidence about AI-authored activity in a repository. To obtain a more complete set of AI-authored commits, we perform a bare clone of each candidate repository. We then scan the full commit history across all branches and apply the same attribution rules to every commit. For each commit, we extract the SHA, author and committer metadata, timestamp,

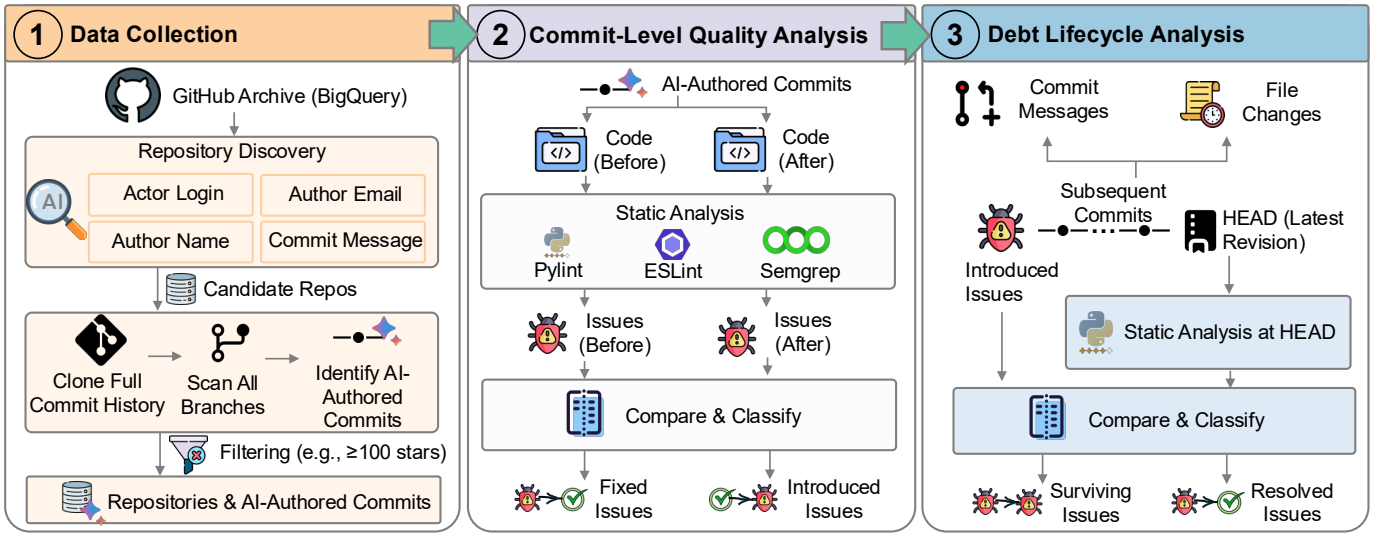


Fig. 4: Overview of our approach.

and full commit message. This step allows us to identify AI-authored commits that are not directly visible during repository discovery (e.g., commits on non-default branches, commits outside the observation window).

Filtering. To focus on established open-source projects, we filter out repositories that do not meet our study criteria. We keep only repositories with at least 100 GitHub stars. We also require at least one confirmed AI-authored commit. Our downstream analysis is restricted to production Python, JavaScript, and TypeScript source files, since these are among the most widely used programming languages [2] and are well supported by static analysis tools. We therefore exclude repositories that do not contain any source files in these languages. In total, the discovery stage identified *587,118* candidate repositories. After applying the star threshold, *12,770* repositories remained. After full-history scanning and language filtering, we obtained *6,699* repositories with confirmed AI-authored commits.

B. Commit-Level Quality Analysis

To understand the impact of AI coding tools, we avoid evaluating a repository snapshot at a single point in time. For each AI-authored commit c, we analyze two versions of the source code: the version at c's parent revision (*before* the commit is applied) and the version at c itself (*after* the commit is applied). Comparing these two versions allows us to determine which quality issues the commit introduced or fixed. Our analysis focuses on source files written in Python, JavaScript, and TypeScript. We exclude files that are unlikely to reflect production code quality (e.g., tests, documentation, configuration files, auto-built artifacts, and vendored dependencies). Files are classified based on their paths and naming patterns (e.g., files under `test/` or `__tests__/` directories, or matching `*_test.py`). The full classification rules are included in our replication package.

Static Analysis. For each AI-authored commit c that modifies a source file f, we check out two versions of f (i.e., the version at c's parent commit (*before*) and the version after applying c (*after*)). We run the same static analysis toolchain on both versions to identify potential code issues. We use ESLint (for JavaScript and TypeScript) [34] and Pylint (for Python) [35] to detect code smells and correctness issues. For security-related issues, we use Semgrep [36], which provides a unified framework for multi-language static analysis. For each detected issue, we record its rule identifier, line number, detector, and message. Figure 5 shows one such record, where ESLint flags a duplicate object key in a Claude-authored commit [37]. This produces two issue sets I: the issue set before the commit, denoted by I_f^-, and the issue set after the commit, denoted by I_f^+.

```
Repository: superagent-ai/superagent
Commit: 46695d1 (author: Claude)
File: node/src/server.js
Detector: ESLint
Rule: no-dupe-keys
Line: 86
Message: Duplicate key 'lazyConnect'.
```

Fig. 5: Example of a recorded issue detected by ESLint in a Claude-authored commit [37].

Differential Attribution. To find out which issues commit c introduced or fixed, we compare I_f^- and I_f^+. We also use `git diff` to extract the set of changed lines, denoted by Δ_f. However, not all differences between I_f^- and I_f^+ (i.e., issues in $I_f^+ \setminus I_f^-$ or $I_f^- \setminus I_f^+$) represent real changes. When a commit inserts or deletes lines, it can cause existing issues to shift line numbers. To address this, we first match issues across the two sets. An issue i is considered matched if the same rule and message appear in both I_f^- and I_f^+ at the same or nearby line number. After matching, the remaining issues are classified as follows. An unmatched issue $i \in I_f^+$ is classified as *introduced* only if its line falls within Δ_f. This means the issue exists only after the commit, on a line that the commit

TABLE I: Summary of AI-authored commits by coding tool.

AI Coding Tool†	# AI Commits	# Repos	Avg. Commits/Repo
GitHub Copilot	118,012	4,487	26.3
Claude	138,249	2,434	56.8
Cursor	19,587	714	27.4
Gemini	12,429	681	18.3
Devin	14,302	298	48.0
Total	302,579	6,299*	48.0

†Each name groups all variants of the tool (e.g., *Claude* includes Claude Code and its different model versions). *Repositories may use more than one tool; the total is deduplicated.

actually changed. An unmatched issue $i \in I_f^-$ is classified as *fixed*. This means the issue existed before the commit but is no longer present afterward.

C. Debt Lifecycle Analysis

Detecting technical debt at the time of introduction is only half the picture. An issue that is quickly resolved has a very different cost than one that lingers for months. We therefore track whether AI-introduced issues persist or get resolved over time.

Issue Survival. For each issue introduced by an AI-authored commit, we check whether it still exists at the repository's latest revision (i.e., `HEAD`). If the file has been renamed, we follow its history using `git log --follow`. We then run static analysis on the corresponding file at `HEAD`. Next, we look for the same issue in the analysis results. We do not rely on the line number alone, since the location of the issue may move as the file changes. Instead, we match issues using their rule identifier together with a small amount of surrounding code context. If a match is found, the issue is classified as *surviving*. Otherwise, it is classified as *not surviving*. In other words, an introduced issue is counted as surviving only if the same issue is still present at `HEAD`. If the original issue disappears and a different issue appears later, the original issue is treated as *not surviving*.

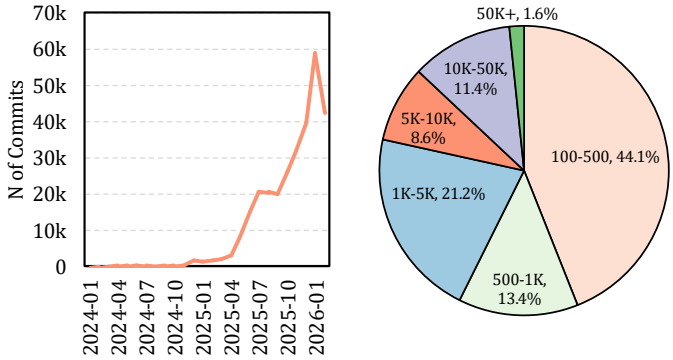
At the same time, we also record whether files touched by AI-authored commits are modified again before `HEAD`. We trace the subsequent commit history of each affected file to understand how actively it is maintained after the AI-authored change. This additional context helps us interpret the survival results and understand the maintenance patterns around AI-introduced debt.

IV. Experimental Setup

In this section, we describe the experimental setup for our study.

A. Dataset Summary

We collected our dataset by mining GitHub and applying filtering criteria described in Section III-A. After repository-level filtering, we obtained 6,699 repositories with confirmed AI-authored commits, covering 29 AI coding tools. However, some tools have very few commits, which may not provide reliable data for comparison. Thus, we focus on the five assistants with more than 10,000 attributed commits: GitHub Copilot [38], Claude [39], Cursor [40], Gemini [41], and Devin [42]. This results in 6,412 repositories with 317.4K AI-attributed commits. For the commit-level quality analysis, we further exclude cases that cannot be analyzed reliably such as repositories that became unavailable, deletion-only commits, and commits that do not modify production Python, JavaScript, or TypeScript source files. Our final analysis dataset therefore includes **6,299** public GitHub repositories with **302.6K** analyzed AI-attributed commits. Table I summarizes the distribution of commits across the five AI coding assistants in our dataset. Figure 6a shows the monthly growth of AI-authored commits, with a sharp increase starting from mid-2025. Figure 6b shows that our dataset covers repositories with a wide range of popularity levels.



(a) Monthly commits. (b) Repository distribution.

Fig. 6: Overview of our dataset: (a) growth of AI-authored commits over time, and (b) distribution of repositories by GitHub star count as of March 2026.

B. Research Questions

Our study focuses on the following research questions:

RQ1: What kinds of technical debt are introduced by AI coding assistants? This question investigates the basic characteristics of AI-introduced technical debt. We study what types of debt they introduce, including code smells, correctness issues, and security issues. We also analyze how these issues are distributed across languages, rules, and AI coding assistants.

RQ2: How does technical debt vary across AI coding assistants? This question compares different AI coding assistants at the commit level. We examine whether some assistants introduce more technical debt than others, and whether the kinds of issues they introduce differ. This helps us understand whether technical debt patterns are tool-specific.

RQ3: To what extent does AI-introduced technical debt persist in the codebase? Introducing debt is not necessarily a problem if it gets fixed quickly. The real concern is debt that persists unnoticed. We compare the number of issues introduced and fixed by AI-authored commits to assess the net impact. We study whether introduced issues remain in the latest version of the repository or disappear over time.

C. Evaluation Metrics

We use the following metrics to support our three research questions. For issue introduction (RQ1, RQ2), we report the total number of issues introduced by AI-authored commits, the percentage of commits that introduce at least one issue, and the average number of issues per commit. We break these down by issue type, programming language, rule, and AI coding assistant.

For the debt lifecycle (RQ3), we use two metrics. First, we compute the net impact by comparing the number of issues introduced and fixed by AI-authored commits. Second, we measure the survival rate of introduced issues:

$$\textit{Survival Rate} = \frac{\text{\# issues surviving at } \texttt{HEAD}}{\text{\# issues tracked}}$$

D. Validation

To assess the reliability of our pipeline, two authors independently inspected random samples of AI-attributed commits and detected issues.

We randomly sampled 100 AI-attributed commits. For each commit, we manually verified whether it was correctly attributed to the claimed AI coding assistant using the Git metadata and commit message described in Section III-A. One sampled case could not be verified because the corresponding repository was no longer available. For the remaining 99 verifiable commits, both authors independently labeled all cases, and all 99 were confirmed as correctly attributed after manual inspection. This yields a conservative attribution precision of 99.0% over the full sample, or 100% over the verifiable subset.

We also randomly sampled 100 introduced issues from manually verified AI-attributed commits and validated two aspects of each: (1) whether the reported issue was a real issue, rather than a false positive from the static analysis tools; and (2) whether the survival classification at `HEAD` was correct (i.e., whether the issue was correctly labeled as surviving or not surviving). After excluding one case with unavailable or incomplete validation context, both authors independently labeled 99 issue cases. For issue validity, the raw agreement was 95/99 = 0.960, with Cohen's $\kappa = 0.851$, indicating almost perfect agreement. For survival classification, the raw agreement was 97/99 = 0.980, with Cohen's $\kappa = 0.960$, also indicating almost perfect agreement.

Using the adjudicated labels as reference, the pipeline achieved an accuracy of 85.9%, precision of 85.9%, recall of 100.0%, and F1 of 92.4% for issue validity. For survival classification, the pipeline achieved an accuracy of 84.8%, precision of 86.7%, recall of 81.2%, and F1 of 83.9%. These results suggest that our pipeline provides reliable large-scale estimates, while still leaving some room for noise in issue detection and lifecycle tracking.

V. Results

This section presents the empirical results of our study and answers the three research questions raised in Section IV-B.

TABLE II: Overview of AI-introduced technical debt by issue type.

Type	# Introduced	# Repos	# Commits	% of Issues
Code Smells	432,748	3,812	25,740	89.3%
Correctness Issues	28,931	665	1,650	6.0%
Security Issues	22,687	1,643	5,142	4.7%
All Issues	**484,366**	**3,946**	**27,677**	**-**

TABLE III: Top 5 most frequent rules violated by AI coding assistants for each issue type.

Type	Rule	Count	Rate
Code Smells	Broad exception handling	41,374	8.5%
	Unused variables or parameters	28,272	5.8%
	Unused argument	24,357	5.0%
	Shadowed outer variable	20,647	4.3%
	Access to protected member	19,796	4.1%
Correctness Issues	Undefined variable or reference	23,856	4.9%
	Redeclared symbol	1,888	0.4%
	Possibly used before assignment	1575	0.3%
	Access member before definition	893	0.2%
	Unsubscriptable object	176	0.0%
Security Issues	Path traversal via path.join/resolve	8,677	1.8%
	Unsafe format string	4,792	1.0%
	Non-literal regular expression	1,212	0.3%
	Child process execution	607	0.1%
	SQLAlchemy raw query execution	591	0.1%

A. RQ1: Types and Patterns of AI-Introduced Debt

Overview. Table II presents a summary of the technical debt introduced by AI coding assistants in our dataset. In total, we identified 484,366 introduced issues across 3,946 repositories (62.6% of 6,299 repositories) and 27,677 commits (9.1% of 302,579 commits). This shows that a non-trivial portion of AI-authored commits introduce quality issues, and that these issues affect a large number of real-world repositories. Among all introduced issues, code smells, correctness issues, and security issues are the three main categories. Code smells are by far the most common, accounting for 89.3% of all introduced issues. Below, we discuss each type in detail with real-world examples.

Code Smells. Code smells are maintainability problems that make code harder to understand, debug, and evolve [45]. They increase long-term maintenance costs, even if they do not cause immediate failures. This finding is consistent with prior work under controlled settings [5], [6], but our study confirms that the same pattern also appears in real-world repositories. Table III lists the top 5 most common code smell patterns (e.g., broad exception handling, unused variables or parameters). These issues are often small and easy to overlook during code review. Listing 1 shows an example from `ArchiveBox` [44] (>27k stars). In commit `d36079829bed` [43], Claude Code updated the metadata loading logic in `ArchiveBox`. But the new code introduces two code smells. First, the bare `except: pass` block catches all exceptions silently. This makes errors harder to detect and debug. Second, the `open()` function does not specify a file encoding. This can lead to inconsistent behavior across different platforms and locales, since the default encoding may vary [46]. These issues may not cause immediate failures, but they can lead to maintenance challenges and subtle bugs in the future.

ArchiveBox [43], core/models.py:L594:598, d360798

```
# issue: broad exception handler silently swallows JSON-
    loading failures
try:
    with open(json_path) as f:
        data = json.load(f)
except:
    pass
```

Listing 1: Code smell: broad exception handling and missing file encoding in `ArchiveBox` [44] (Claude Code).

firecrawl [47], firecrawl.py:L4004:4047, fb99747

```
# issue: undefined variable "cache" causes a runtime
    NameError
async def generate_llms_text(
        self,
        url: str,
        *,
        max_urls: Optional[int] = None,
        show_full_text: Optional[bool] = None,
        experimental_stream: Optional[bool] = None) ->
    GenerateLLMsTextStatusResponse:

    response = await self.async_generate_llms_text(
        url,
        max_urls=max_urls,
        show_full_text=show_full_text,
        cache=cache,
        experimental_stream=experimental_stream
    )
```

Listing 2: Correctness issues: undefined variable causing `NameError` in `Firecrawl` [48] (Devin).

data-formulator [50], tables_routes.py:L881:889, d8549c0

```
# issue: user-controlled table name is interpolated
    directly into SQL
for source_name in source_table_names:
    if source_name == updated_table_name:
        df = pd.DataFrame(updated_table['rows'])
    else:
        with db_manager.connection(session['session_id'])
         as db:
            result = db.execute(
              f"SELECT * FROM {source_name}").fetchdf()
            df = result
```

Listing 3: Security issue: possible SQL injection in `microsoft/data-formulator` [51] (Copilot).

TABLE IV: Top 5 most frequent rules by language. Rate indicates each rule's share of all issues in that language.

Language	Rule	Count	Rate
Python	Broad exception handling	41,374	14.9%
	Unused argument	24,357	8.8%
	Undefined variable or reference	23,856	8.6%
	Access to protected member	19,796	7.1%
	Unused import	17,376	6.3%
JavaScript/ TypeScript	Unused variables or parameters	28,272	13.6%
	Shadowed outer variable	18,417	8.9%
	Block-scoped variable misuse	11568	5.6%
	No sequences	9358	4.5%
	Path traversal via path.join/resolve	8677	4.2%

Correctness Issues. Correctness issues are code defects that can cause the program to fail during execution. Compared with code smells, they are less frequent. From Table II, 28,931 correctness issues are identified, which cover 665 repositories and 1,650 commits. However, their impact is more direct and severe than code smells. Table III shows the top 5 most common correctness issues, which include undefined variable or reference, redeclared symbol, access to member before definition, possibly used before assignment, and unsubscriptable object. These patterns suggest that AI-generated code may look locally correct, but still fail to stay consistent with the surrounding context. What is interesting in this table is that we identified 23,856 cases of undefined variable or reference. Listing 2 presents one such case from `firecrawl` (>98k stars). In commit `fb99747ba978` [47], Devin added a call that passes `cache=cache` as an argument. However, `cache` is never defined in the method, which leads to a `NameError` when that path is executed. The maintainer later fixed the bug by removing the undefined argument [49]. This example shows that AI-generated code can introduce real correctness issues. These errors require additional human effort to fix later.

Security Issues. Security issues are another concern in AI-generated code. In our study, this category includes not only direct security vulnerabilities, but also insecure coding patterns that can be viewed as security debt. Some of these issues may be exploitable at the time they are introduced, while others may become security risks after later code changes or broader system integration. Thus, it is important to identify and fix these issues early before they accumulate in production repositories. As shown in Table II, potentially insecure code patterns are detected in 1,643 repositories and 5,142 commits. Table III shows that common security issues such as path traversal via `path.join` or `path.resolve`, unsafe format strings, non-literal regular expressions, and child process execution. These patterns suggest that AI-generated code can introduce unsafe practices in process execution, file path handling, and string formatting. A common pattern across these issues is unsafe handling of untrusted input, where user- or context-controlled values flow into security-sensitive operations without proper validation or sanitization. Figure 2 shows one such example from `hysteria2` (>1.5k stars). A Copilot-authored commit [28] introduced a `shell=True` subprocess call. This pattern is not necessarily an exploitable vulnerability by itself in the current version, but it creates security debt because it can enable command injection if untrusted input later reaches the command string. A human developer later identified and removed the unsafe flag [25]. Beyond that, Listing 3 shows another example of a security issue (SQL injection) in `data-formulator` (>1.2k stars). This repository is developed by Microsoft and uses GitHub Copilot for code generation. In commit `d8549c0` [50], Copilot added a backend endpoint that constructs a SQL query by directly interpolating a user-supplied table name. This creates a potential SQL injection vector. If an attacker can control the `source_name` variable, they can inject malicious SQL code that may lead to data breaches or unauthorized access. This issue remained in the repository for several weeks before the maintainer refactored the code and removed the unsafe SQL construction [51].

Programming Language Differences. Table IV compares

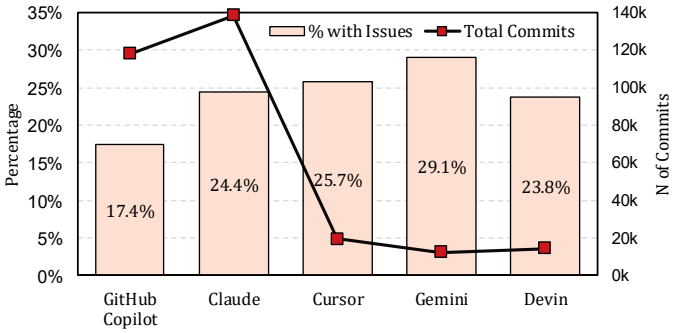


Fig. 7: Percentage of commits with issues and total commit volume per AI coding assistant.

Coding Tool	All Issues	Smells	Correctness	Security
GitHub Copilot	1.3	1.156	0.101	0.045
Claude	1.95	1.736	0.112	0.103
Cursor	1.59	1.47	0.03	0.09
Gemini	1.39	1.27	0.054	0.062
Devin	0.89	0.824	0.018	0.043

TABLE V: Average number of issues introduced per commit by type. Darker cells indicate higher values.

the top 5 most frequent rules in Python and JavaScript/TypeScript. There are some patterns that are common in both languages. For example, both languages show issues related to unused code (i.e., unused arguments in Python, unused variables in JavaScript/TypeScript). At the same time, each language also has its own characteristic issues. Python's top rules are dominated by exception handling and dynamic typing problems, while JavaScript/TypeScript issues tend to involve scoping and variable declaration patterns. This observation is consistent with prior studies [5], [6], which also found that the types of issues in AI-generated code can vary depending on the programming language and the tools used. These results suggest that some debt patterns may be language-specific. However, the overall trend (e.g., code smells dominate) holds across both languages.

> ***Answer to RQ1:*** *AI-generated code introduces technical debt in the form of code smells (89.3%), correctness issues (6.0%), and security issues (4.7%). Among them, code smells are by far the most common.*

B. RQ2: Comparison Across AI Coding Assistants

In this RQ, we examine how technical debt patterns vary across AI coding assistants. Figure 7 shows the percentage of commits with issues for each of the five AI coding assistants. What stands out in this figure is that more than 15% of commits by each AI coding tool introduce at least one issue. The rates also vary across tools, ranging from 17.4% for GitHub Copilot to 29.1% for Gemini. This suggests that technical debt appears across all studied tools, although the rate differs by tool. In other words, the problem is not limited to a single AI coding assistant.

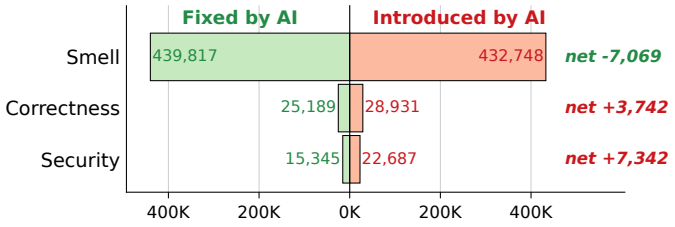


Fig. 8: Net impact of AI coding assistants: issues introduced vs. fixed by issue type.

TABLE VI: Survival of AI-introduced issues by time since introduction.

Time Since Introduction	# Commits	# Issues	# Surviving	Survival Rate
$>$ 9 months	17,612	21,421	4,893	22.8%
6–9 months	56,955	89,135	17,307	19.4%
3–6 months	79,986	111,551	31,492	28.2%
$<$ 3 months	147,979	242,793	51,672	21.3%
All Issues	**302,532**	**464,900**	**105,364**	**22.7%**

Table V further compares the average number of introduced issues per commit by type. We can see that all five tools share a common pattern, where the code smell rate is much higher than the correctness and security issue rates. This is consistent with our findings in RQ1. At the same time, there are also differences across tools. From Table V, we can see that Claude has the highest issue rate per commit (1.95), while Devin has the lowest (0.89). These differences may be due to differences in usage patterns and development context, rather than the tools alone. Still, it is apparent that the overall pattern of technical debt is consistent across all five tools.

> ***Answer to RQ2:*** *Technical debt patterns vary across AI coding assistants, but the overall trend is consistent. Across all five tools, code smells remain the dominant form of AI-introduced debt.*

C. RQ3: Persistence of AI-Introduced Debt

Net Impact. In RQ1 and RQ2, we focus on the technical debt introduced by AI-authored commits. However, AI coding assistants can also remove existing issues during refactoring or code improvement. To better understand the overall lifecycle of AI-introduced debt, we compare the number of issues introduced and fixed by AI commits (see Figure 8). For code smells, we can see that AI-authored commits fix more issues than they introduce (439,817 vs. 432,748), resulting in a net reduction of 7,069 code smells. In contrast, for correctness and security issues, AI commits introduce more issues than they fix. What is interesting is that AI introduces about 1.5 times as many security issues as it fixes. These findings indicate that the net impact of AI coding assistants is mixed. AI coding assistants can help reduce maintainability issues, which tend to follow simple and repetitive patterns. However, for correctness and security issues, which require a deeper understanding and reasoning about program logic and context, AI coding assistants introduce more problems than they resolve.

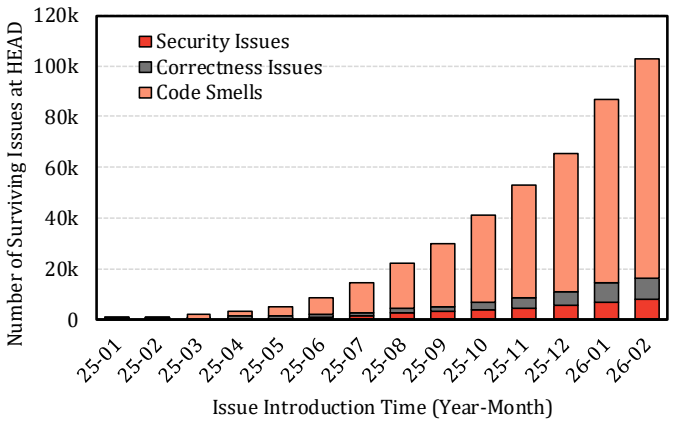


Fig. 9: Cumulative growth of AI-introduced issues over time, by issue type.

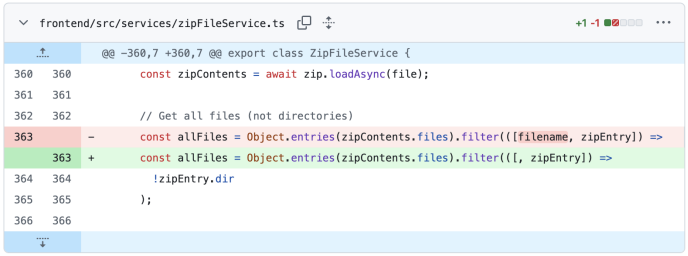


Fig. 10: A TypeScript lint issue introduced by a Claude-authored commit in `Stirling-PDF` was fixed one day later by removing the unused variable.

Issue Survival. The net impact analysis above provides an overview of what AI coding assistants add and remove. But it does not show what happens to the specific issues introduced by AI. To answer this question, we track each AI-introduced issue to the latest repository snapshot and check whether it still exists at `HEAD`. Figure 9 shows that the cumulative number of surviving issues keeps growing over time. The total volume of unresolved technical debt increases rapidly, climbing from just a few hundred issues in early 2025 to over 100k surviving issues by February 2026. This suggests that as the rapid adoption of AI coding assistants continues, the amount of AI-introduced debt in real-world repositories is also growing significantly.

Table VI provides an age-cohort view of issue survival. Overall, 105,364 out of 464,900 tracked AI-introduced issues still survive at `HEAD`, corresponding to a survival rate of 22.7%. Surviving issues appear in all age cohorts, including issues introduced more than nine months earlier. For example, 4,893 issues introduced more than nine months ago still remain at `HEAD`. The survival rate varies across cohorts, ranging from 19.4% for issues introduced 6–9 months ago to 28.2% for issues introduced 3–6 months ago. This suggests that AI-introduced debt is not always removed quickly after it enters the codebase. Although the cohort-level survival rates do not show a simple monotonic trend, the main finding is clear: a substantial number of AI-introduced issues remain unresolved over time.

These aggregate results are also reflected in real-world repositories. For example, the broad exception handling issue in Listing 1 was fixed within hours after it was introduced [43], [52]. Similarly, in `Stirling-PDF` [53] (>75k stars), a Claude-authored commit introduced an unused variable `filename` in a TypeScript file [54]. As shown in Figure 10, the maintainer fixed it the next day with a commit titled "Fix TypeScript linting error" [55]. In contrast, the undefined variable bug in `firecrawl` (Listing 2) took 42 days before the maintainer fixed it [49]. In some cases, some issues can survive for much longer or even remain unresolved. For instance, a Devin-authored commit in `brave_search_tool.py` added a call to `requests.get(...)` without a timeout in December 2024 [56]. This is a known potential security issue, because requests made without a timeout may block indefinitely if the remote service does not respond [57]. However, the issue still remains in the latest repository revision.

> ***Answer to RQ3:*** *AI-authored commits fix slightly more code smells than they introduce, but introduce more correctness and security issues than they fix. Overall, 22.7% of tracked AI-introduced issues still survive at `HEAD`, including issues introduced more than nine months earlier.*

VI. Discussion

A. Implications

Our empirical study shows that AI coding assistants introduce technical debt into real software repositories. This is not a property of any single tool. We observed that across all five tools we studied, more than 15% of commits introduce at least one detectable issue. These issues persist regardless of repository size or popularity. In this section, we discuss the implications of our findings for practitioners, researchers, and tool builders.

AI-assisted development creates persistent debt, not just temporary low-quality code. The main implication is not just that AI coding assistants may produce low-quality code. More importantly, AI-assisted software development changes how technical debt enters and remains in production systems. Prior studies have shown that developers are more likely to over-trust AI suggestions [10]. This over-trust can lead to a higher acceptance rate of AI-generated code, even when it contains issues. It means that many AI-introduced issues can accumulate in the codebase. Our results show that code smells are the most common type of AI-introduced debt. They often do not break the software system immediately, making them easy to accept during code review. But AI coding assistants allow developers to produce code at a much higher speed and volume. As a result, these minor issues can accumulate into a substantial maintenance burden over time. Figure 9 shows that the cumulative number of surviving AI-introduced issues continues to rise over time, exceeding 100k by February 2026. The technical debt introduced by AI does not have to be a temporary side effect, but a long-term maintenance challenge for modern software systems.

Developers should be especially cautious about correctness and security issues. Our findings suggest that although AI coding assistants introduce technical debt, they also fix

existing issues in the codebase. First, we observe that AI co-authored commits actually fix a similar number of code smell issues as they introduce. This suggests that AI coding assistants are able to perform local cleanup and repetitive maintenance tasks effectively. They can recognize and address surface-level code quality problems (e.g., formatting, naming, or simple refactoring opportunities). However, what is concerning is that AI coding assistants seem to be less effective at fixing correctness and security issues, and they even introduce more of these than they fix. Section V-C shows that many AI-introduced issues still survive at `HEAD`, while the net-impact analysis shows that AI commits introduce more correctness and security issues than they fix. This inconsistency suggests that AI coding assistants may struggle with changes that require a deeper understanding of program behavior, execution context, or security implications. Also, the practical impact of correctness and security issues is often more severe than that of code smells, making it more critical to address them effectively. Developers should not treat all AI-generated code as equally trustworthy, and they should pay particular attention to changes that may introduce correctness or security vulnerabilities.

Technical debt cannot be solved by switching between AI coding tools. Our cross-tool comparison shows that this problem cannot be solved simply by switching from one assistant to another. Figure 7 shows that all five tools introduce a similar pattern of issues. They all have a high rate of code smells, and a non-trivial rate of correctness and security issues. This suggests that the quality risk is a systemic issue with the current mode of AI-assisted development. Thus, developers and teams should carefully review AI-generated code regardless of the tool used. Static analysis, tests, and security checks should be part of the normal workflow. Code review should extend beyond the point of merge. The main reason is that our study shows that 22.7% of tracked AI-introduced issues still survive at `HEAD`, and older issues are not fully cleaned up even after months. Merging the AI-generated code does not mean the end of the story since debt can persist and accumulate over time. This makes continuous monitoring and targeted debt repayment necessary for AI-touched code.

Future research and tool design should target long-term code health. Our results suggest that future research and tool design should not just focus on generating more acceptable code. We also need to ask whether that code remains maintainable, correct, and secure over time. Most existing research studies [5], [6] focus on short-term outcomes such as task completion, acceptance rate, or immediate correctness. However, these measures only capture what happens when the code is introduced, not what happens later in maintenance. Future work needs to examine which factors make AI-introduced debt more likely to persist (e.g., repository maturity, review intensity, or task type). Our findings also suggest the need for better assistants. Future tools should make stronger checks for security-sensitive changes, use repository context more effectively, and clearly show AI provenance so reviewers can better judge the risk of a change. In the end, the key question is not only whether AI can produce code at scale, but whether the software engineering ecosystem can manage that code well over time.

VII. Related work

Code Quality of AI-Generated Code. Previous studies have examined the quality and security of AI-generated code. Based on controlled experiments, they showed that AI coding assistants (e.g., GitHub Copilot and ChatGPT) are able to produce functional code, but the code quality varies widely across languages, tasks, and prompts [5], [30], [58]. AI-generated code can also contain security weaknesses, and developers may over-trust it and fail to properly review it [7], [8], [59]. Recent studies have also begun examining AI-generated code in real-world production environments. They show that AI-generated code is being widely adopted in platforms such as GitHub and Stack Overflow, and that it can carry quality and security issues [6], [13], [14]. He *et al.* [15] studied the impact of Cursor adoption on 807 repositories and observed persistent increases in code complexity. Watanabe *et al.* [16] found that most Claude Code pull requests are merged, though many require human revisions.

Together, these studies show that AI-generated code can contain quality and security problems in both controlled and real-world settings. However, these studies mainly focus on a single tool or a narrow set of quality issues, and they do not track how those issues evolve. In contrast, our work provides a comprehensive analysis of the quality and security of AI-generated code across multiple tools, and tracks how those issues persist in production repositories over time.

Empirical Studies of AI-Assisted Development Practices. Recent studies have examined how AI coding assistants are used in real software development practice. Several studies examined the productivity impact of these tools. Peng *et al.* [60] conducted a randomized controlled trial and found that developers using GitHub Copilot completed tasks 55.8% faster. At industrial scale, Ziegler *et al.* [61] and Murali *et al.* [62] reported that suggestion acceptance rate strongly correlates with self-reported productivity, with acceptance rates around 22–30%. Other studies focus on usage patterns and developer perceptions. Liang *et al.* [63] surveyed 410 developers and found that developers mainly use AI assistants to reduce keystrokes and recall syntax, but often reject suggestions that fail to meet functional requirements [64]. Sabouri *et al.* [10] further observed that developers keep only 52% of AI suggestions after review, and Klemmer *et al.* [65] found that developers widely use AI assistants for security-critical tasks despite concerns about suggestion quality.

These studies improve our understanding of how developers use AI assistants in practice. However, they mainly focus on adoption, usability, trust, productivity, and collaboration, rather than on the technical debt carried by AI-authored code. In contrast, our work examines the code-level debt introduced by AI coding assistants and studies whether that debt persists in production repositories over time.

Technical Debt in Software Development. Technical debt has been a core topic in software engineering research for years. Previous work has introduced many automated methods to identify code smells and self-admitted technical debt

in software repositories [66]–[68]. Beyond detection, many studies have also investigated the lifecycle of technical debt in human-written code. Tufano *et al.* [69] showed that code smells are often introduced during normal development. Once introduced, they can linger in the codebase for a long time before anyone removes them [69]. Digkas *et al.* [70] identified a similar pattern in large open-source ecosystems. Their results showed that technical debt mainly accumulates when new code is added [70], [71]. Other studies further suggest that debt is rarely removed in an intentional way, and that automated repayment is still difficult in practice [72], [73].

Prior studies provide a strong foundation for understanding technical debt in human-written software. However, it remains unclear whether AI-generated code follows the same patterns, and whether the same tools and techniques can be applied to it. Our work fills this gap by analyzing the technical debt in AI-generated code and exploring how it persists in production repositories over time.

VIII. Threats to Validity

Below, we discuss threats that may impact the results of our study.

External Validity. Our study focuses on public GitHub repositories with at least 100 stars and production source files in Python, JavaScript, and TypeScript. Therefore, our findings may not generalize to private repositories, smaller projects, or software written in other languages. In addition, we only analyze AI-authored commits that leave explicit traces in Git metadata (e.g., bot actor logins, AI author emails, or `Co-authored-by` trailers). AI-assisted contributions that leave no such trace are outside the scope of our dataset. This design is analogous in spirit to research on self-admitted technical debt (SATD), which studies the subset of technical debt that developers explicitly document rather than the full universe of debt [18], [19]. Like SATD research, our findings characterize a visible and attributable subset rather than the entire population of AI-assisted work. We also do not compare AI-authored commits against a baseline of purely human-written commits. Because developers may use AI without leaving any Git trace, and even AI-labeled commits may mix human edits, a reliable human-only baseline is difficult to construct, and comparing against an unreliable baseline could bias the results. We therefore focus on tracking technical debt inside explicitly confirmed AI-authored or co-authored commits.

Internal Validity. Our pipeline depends on the correctness of AI attribution, issue matching, and lifecycle tracking. Although we use explicit Git metadata rather than proxy signals, some commits may still include both AI and human contributions. Similarly, matching issues across revisions is challenging because files evolve over time and issues may shift location. To reduce this risk, we compare code before and after each commit, restrict introduced issues to changed lines, and use rule identifiers, messages, and surrounding code context during matching. For the issue survival analysis, a file may be deleted or entirely rewritten between the introducing commit and `HEAD`. In such cases, the issue is classified as resolved, even though the resolution may not be a deliberate fix. Our reported survival rates may therefore slightly underestimate the true persistence of AI-introduced debt.

Construct Validity. Technical debt is a broad concept with many possible forms. In this study, we operationalize technical debt mainly through code smells, correctness issues, and security issues detected by static analysis tools. This choice allows us to measure debt consistently at scale and track it over time at the commit level. However, static analysis tools can produce false positives, flagging code that is technically correct but matches a known risky pattern. In addition, our security findings include both active vulnerabilities and latent unsafe patterns (i.e., security debt). We do not separate these quantitatively, but Section V-A illustrates both cases. Static analysis also does not cover all forms of technical debt. Architectural debt, design erosion, documentation debt, and test adequacy issues are outside the scope of our tools. Our results should therefore be interpreted as evidence about code-level technical debt, not the full spectrum of quality challenges that AI-generated code may introduce.

IX. Conclusion and Future Work

AI coding assistants are rapidly becoming part of real-world software development, but their long-term impact on software quality remains unclear. In this paper, we presented a large-scale empirical study of the technical debt introduced by AI-generated code in the wild. By mining 302.6K AI-authored commits from 6,299 GitHub repositories, we designed a commit-level pipeline to identify introduced technical debt and track its later evolution in production repositories. Our study provides a longitudinal view of AI-generated code after it is merged. We look at what kinds of debt appear, how they differ across assistants, and whether they still remain at the latest revision. The results reveal the hidden maintenance costs behind AI coding assistants. They also show why stronger quality checks are needed in AI-assisted software development. In future work, we plan to extend our analysis to other forms of debt (e.g., architectural, documentation, and test-related debt), additional programming languages, and broader development contexts. We also plan to study what factors make AI-introduced debt more likely to persist. Finally, we hope our findings can inform the design of more debt-aware AI coding assistants that help developers catch and fix quality issues before they enter production.

References

[1] GitHub, "Octoverse: A new developer joins github every second as ai leads typescript to #1," October 2024. [Online]. Available: https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/

[2] Stack Overflow, "2025 developer survey," June 2025. [Online]. Available: https://survey.stackoverflow.co/2025/

[3] J. Novet, "Satya nadella says as much as 30% of microsoft code is written by ai," April 2025. [Online]. Available: https://www.cnbc.com/2025/04/29/satya-nadella-says-as-much-as-30percent-of-microsoft-code-is-written-by-ai.html

[4] K. Robison, "Google ceo sundar pichai says more than a quarter of the company's new code is created by ai," October 2024. [Online]. Available: https://fortune.com/2024/10/30/googles-code-ai-sundar-pichai/

[5] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–26, 2024.

[6] M. L. Siddiq, L. Roney, J. Zhang, and J. C. D. S. Santos, "Quality assessment of chatgpt generated code and their use by developers," in *Proceedings of the 21st international conference on mining software repositories*, 2024, pp. 152–156.

[7] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, 2025.

[8] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" in *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*, 2023, pp. 2785–2799.

[9] Y. Liu, Z. Xing, S. Pan, and C. Tantithamthavorn, "When ai takes the wheel: Security analysis of framework-constrained program generation," *arXiv preprint arXiv:2510.16823*, 2025.

[10] S. Sabouri, P. Eibl, X. Zhou, M. Ziyadi, N. Medvidovic, L. Lindemann, and S. Chattopadhyay, "Trust dynamics in ai-assisted development: Definitions, factors, and implications," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 1678–1690.

[11] W. Harding and M. Kloster, "Coding on copilot: 2023 data suggests downward pressure on code quality," *https://www. gitclear. com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality/*, 2024.

[12] S. Moreschini, E.-M. Arvanitou, E.-P. Kanidou, N. Nikolaidis, R. Su, A. Ampatzoglou, A. Chatzigeorgiou, and V. Lenarduzzi, "The evolution of technical debt from devops to generative ai: A multivocal literature review," *Journal of Systems and Software*, vol. 231, p. 112599, 2026.

[13] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security weaknesses of copilot-generated code in github projects: An empirical study," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 8, pp. 1–34, 2025.

[14] B. Wang, W. Yu, Y. Zhong, H. Yu, K. Lian, C. Lu, H. Zheng, D. Zhang, and H. Li, "Ai code in the wild: Measuring security risks and ecosystem shifts of ai-generated code in modern software," *arXiv preprint arXiv:2512.18567*, 2025.

[15] H. He, C. Miller, S. Agarwal, C. Kästner, and B. Vasilescu, "Does ai-assisted coding deliver? a difference-in-differences study of cursor's impact on software projects," *arXiv e-prints*, pp. arXiv–2511, 2025.

[16] M. Watanabe, H. Li, Y. Kashiwa, B. Reid, H. Iida, and A. E. Hassan, "On the use of agentic coding: An empirical study of pull requests on github," *arXiv preprint arXiv:2509.14745*, 2025.

[17] H. Huang, P. Jaisri, S. Shimizu, L. Chen, S. Nakashima, and G. Rodríguez-Pérez, "More code, less reuse: Investigating code quality and reviewer sentiment towards ai-generated pull requests," *arXiv preprint arXiv:2601.21276*, 2026.

[18] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 91–100.

[19] E. da Silva Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1044–1062, 2017.

[20] Anthropic, "Claude opus 4.6 wrote a dependency-free c compiler in rust, with backends targeting x86 (64- and 32-bit), arm, and risc-v, capable of compiling a booting linux kernel," February 2026. [Online]. Available: https://github.com/anthropics/claudes-c-compiler

[21] W. Cunningham, "The wycash portfolio management system," *ACM Sigplan Oops Messenger*, vol. 4, no. 2, pp. 29–30, 1992.

[22] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, "Managing technical debt in software engineering (dagstuhl seminar 16162)," *Dagstuhl reports*, vol. 6, no. 4, pp. 110–138, 2016.

[23] Z. Li, P. Avgeriou, and P. Liang, "A systematic mapping study on technical debt and its management," *Journal of systems and software*, vol. 101, pp. 193–220, 2015.

[24] seagullz4, "hysteria2," 2025, accessed: 2026-01-15. [Online]. Available: https://github.com/seagullz4/hysteria2

[25] ——, "Commit `d9e392d`: Improve code security by removing shell=true," 2025. [Online]. Available: https://github.com/seagullz4/hysteria2/commit/d9e392d

[26] RealSense, "librealsense," 2025, accessed: 2026-01-15. [Online]. Available: https://github.com/IntelRealSense/librealsense

[27] Intel RealSense, "Commit `14026c8`: Add missing constants and fix 6fps bug," 2025. [Online]. Available: https://github.com/realsenseai/librealsense/commit/14026c898f790db79a0b588983c08a3108fa326e

[28] seagullz4, "Commit `e277daf`: Introduce shell-based subprocess call," 2025. [Online]. Available: https://github.com/seagullz4/hysteria2/commit/e277daf540dad4b5a34822f0088e70617b689587

[29] Intel RealSense, "Commit `5535b8a`: Refactor test script with named constants," 2025. [Online]. Available: https://github.com/realsenseai/librealsense/commit/5535b8a204bc759324ee89f864eb680362be5ece

[30] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[31] GH Archive, "Gh archive," 2026, accessed: 2026-03-20. [Online]. Available: https://www.gharchive.org/

[32] ——, "H archive: Ganalyzing event data with bigquery," 2026, accessed: 2026-03-20. [Online]. Available: www.gharchive.org/#bigquery

[33] GitHub, "Github rest api documentation," 2026, accessed: 2026-03-20. [Online]. Available: https://docs.github.com/en/rest

[34] ESLint, "ESLint Documentation," 2026, accessed: 2026-03-24. [Online]. Available: https://eslint.org/docs/latest/

[35] Python Code Quality Authority, "Pylint Documentation," 2026, accessed: 2026-03-24. [Online]. Available: https://pylint.readthedocs.io/

[36] Semgrep, "Semgrep Documentation," 2026, accessed: 2026-03-24. [Online]. Available: https://semgrep.dev/docs/

[37] superagent-ai, "Commit `46695d1`: feat: Add redis connection pooling for proxy caching layers," Aug. 2025. [Online]. Available: https://github.com/superagent-ai/superagent/commit/46695d14622a6c5de22315ce9514964d22e4d825

[38] GitHub, "GitHub Copilot," 2026, accessed: 2026-03-24. [Online]. Available: https://docs.github.com/en/copilot/get-started/what-is-github-copilot

[39] Anthropic, "Claude Code Overview," 2026, accessed: 2026-03-24. [Online]. Available: https://code.claude.com/docs/en/overview

[40] Cursor, "Cursor Documentation," 2026, accessed: 2026-03-24. [Online]. Available: https://cursor.com/docs

[41] Google, "Gemini Code Assist Overview," 2026, accessed: 2026-03-24. [Online]. Available: https://developers.google.com/gemini-code-assist/docs/overview

[42] Cognition AI, "Introducing Devin," 2026, accessed: 2026-03-24. [Online]. Available: https://docs.devin.ai/get-started/devin-intro

[43] "Commit `d360798`: Replace index.json with index.jsonl flat jsonl format," 2025. [Online]. Available: https://github.com/ArchiveBox/ArchiveBox/commit/d36079829bed32d71b2a1a5e8e6019457d6a7ae7

[44] "Archivebox," 2025. [Online]. Available: https://github.com/ArchiveBox/ArchiveBox

[45] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[46] I. Naoki, "PEP 597 – Add optional EncodingWarning," 2021. [Online]. Available: https://peps.python.org/pep-0597/

[47] firecrawl, "Commit `fb99747`: fix: revert accidental cache=true changes to preserve original cache parameter handling," 2025. [Online]. Available: https://github.com/firecrawl/firecrawl/commit/fb99747ba9787683ac5722ba55c46f823461691a

[48] ——, "firecrawl," 2026. [Online]. Available: https://github.com/firecrawl/firecrawl

[49] ——, "Commit `a7aa0cb`: Fix pydantic field name shadowing issues causing import nameerror," 2025. [Online]. Available: https://github.com/firecrawl/firecrawl/commit/a7aa0cb2f4496394a94b50f0013eb0328b408dc8

[50] Microsoft, "Commit `d8549c0`: Add refresh data feature with backend endpoint and ui components," 2025. [Online]. Available: https://github.com/microsoft/data-formulator/commit/d8549c0c8c139531ee5bf266609f7e5352384c5f

[51] ——, "data-formulator," 2026. [Online]. Available: https://github.com/microsoft/data-formulator

[52] "Commit `762cddc`: fix: address pr review comments from cubic-dev-ai," 2025. [Online]. Available: https://github.com/ArchiveBox/ArchiveBox/commit/762cddc8c5d42095c26dda0e193fab6794fd69d5

[53] S. Tools, "Stirling-pdf," 2026. [Online]. Available: https://github.com/Stirling-Tools/Stirling-PDF

[54] ——, "Commit `e7109bb`: Convert extract-image-scans to react component," 2025. [Online]. Available: https://github.com/Stirling-Tools/Stirling-PDF/commit/e7109bb4e9fbeb1fed7f10f50e5831f48da870be

[55] ——, "Commit `00efc880`: Fix typescript linting error in zipfileservice," 2025. [Online]. Available: https://github.com/Stirling-Tools/Stirling-PDF/commit/00efc8802cd4be7bdf30c746dbd7a2cb1108a601

[56] crewAIInc, "Commit `439cde1`: style: apply final formatting changes," 2024. [Online]. Available: https://github.com/crewAIInc/crewAI-tools/commit/439cde180cd69791f46dedde192c41184ca1f96f

[57] PyCQA, "B113: Test for missing requests timeout," 2023. [Online]. Available: https://bandit.readthedocs.io/en/latest/plugins/b113_request_without_timeout.html

[58] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on github copilot," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2149–2160.

[59] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at c: A user study on the security implications of large language model code assistants," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2205–2222.

[60] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of ai on developer productivity: Evidence from github copilot," *arXiv preprint arXiv:2302.06590*, 2023.

[61] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Measuring github copilot's impact on productivity," *Communications of the ACM*, vol. 67, no. 3, pp. 54–63, 2024.

[62] V. Murali, C. Maddila, I. Ahmad, M. Bolin, D. Cheng, N. Ghorbani, R. Fernandez, N. Nagappan, and P. C. Rigby, "Ai-assisted code authoring at scale: Fine-tuning, deploying, and mixed methods evaluation," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1066–1085, 2024.

[63] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *Proceedings of the 46th IEEE/ACM international conference on software engineering*, 2024, pp. 1–13.

[64] N. Davila, I. Wiese, I. Steinmacher, L. Lucio da Silva, A. Kawamoto, G. J. P. Favaro, and I. Nunes, "An industry case study on adoption of ai-based programming assistants," in *Proceedings of the 46th international conference on software engineering: software engineering in practice*, 2024, pp. 92–102.

[65] J. H. Klemmer, S. A. Horstmann, N. Patnaik, C. Ludden, C. Burton Jr, C. Powers, F. Massacci, A. Rahman, D. Votipka, H. R. Lipford *et al.*, "Using ai assistants in software development: A qualitative study on security practices and concerns," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 2726–2740.

[66] M. Yan, X. Xia, E. Shihab, D. Lo, J. Yin, and X. Yang, "Automating change-level self-admitted technical debt determination," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1211–1229, 2018.

[67] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt: From performance to explainability," *ACM transactions on software engineering and methodology (TOSEM)*, vol. 28, no. 3, pp. 1–45, 2019.

[68] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 482–482.

[69] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

[70] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou, "How do developers fix issues and pay back technical debt in the apache ecosystem?" in *2018 IEEE 25th International Conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2018, pp. 153–163.

[71] G. Digkas, A. Chatzigeorgiou, A. Ampatzoglou, and P. Avgeriou, "Can clean new code reduce technical debt density?" *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1705–1721, 2020.

[72] F. Zampetti, A. Serebrenik, and M. Di Penta, "Was self-admitted technical debt removal a real removal? an in-depth perspective," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 526–536.

[73] A. Mastropaolo, M. Di Penta, and G. Bavota, "Towards automatically addressing self-admitted technical debt: How far are we?" in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 585–597.