Agent READMEs: An Empirical Study of Context Files for Agentic Coding

WORAWALAN CHATLATANAGULCHAI, Faculty of Engineering, Kasetsart University, Thailand
HAO LI, Queen's University, Canada
YUTARO KASHIWA, Nara Institute of Science and Technology, Japan
BRITTANY REID, Nara Institute of Science and Technology, Japan
KUNDJANASITH THONGLEK, Faculty of Engineering, Kasetsart University, Thailand
PATTARA LEELAPRUTE, Faculty of Engineering, Kasetsart University, Thailand
ARNON RUNGSAWANG, Faculty of Engineering, Kasetsart University, Thailand
BUNDIT MANASKASEMSAK, Faculty of Engineering, Kasetsart University, Thailand
BRAM ADAMS, Queen's University, Canada
AHMED E. HASSAN, Queen's University, Canada
HAJIMU IIDA, Nara Institute of Science and Technology, Japan

Agentic coding tools receive goals written in natural language as input, break them down into specific tasks, and write or execute the actual code with minimal human intervention. Central to this process are agent context files ("READMEs for agents") that provide persistent, project-level instructions. In this paper, we conduct the first large-scale empirical study of 2,303 agent context files from 1,925 repositories to characterize their structure, maintenance, and content. We find that these files are not static documentation but complex, difficult-to-read artifacts that evolve like configuration code, maintained through frequent, small additions. Our content analysis of 16 instruction types shows that developers prioritize functional context, such as build and run commands (62.3%), implementation details (69.9%), and architecture (67.7%). We also identify a significant gap: non-functional requirements like security (14.5%) and performance (14.5%) are rarely specified. These findings indicate that while developers use context files to make agents functional, they provide few guardrails to ensure that agent-written code is secure or performant, highlighting the need for improved tooling and practices.

CCS Concepts: • **Software and its engineering → Automatic programming**; **Software configuration management and version control systems**; *Software evolution*; *Maintaining software.*

Additional Key Words and Phrases: Agentic Coding, Autonomous Programming, Documents

ACM Reference Format:
Worawalan Chatlatanagulchai, Hao Li, Yutaro Kashiwa, Brittany Reid, Kundjanasith Thonglek, Pattara Leelaprute, Arnon Rungsawang, Bundit Manaskasemsak, Bram Adams, Ahmed E. Hassan, and Hajimu Iida. 2025.

Authors' addresses: Worawalan Chatlatanagulchai, Faculty of Engineering, Kasetsart University, Bangkok, Thailand; Hao Li, Queen's University, Kingston, Canada; Yutaro Kashiwa, Nara Institute of Science and Technology, Ikoma, Japan, yutaro.kashiwa@is.naist.jp; Brittany Reid, Nara Institute of Science and Technology, Ikoma, Japan; Kundjanasith Thonglek, Faculty of Engineering, Kasetsart University, Bangkok, Thailand; Pattara Leelaprute, Faculty of Engineering, Kasetsart University, Bangkok, Thailand; Arnon Rungsawang, Faculty of Engineering, Kasetsart University, Bangkok, Thailand; Bundit Manaskasemsak, Faculty of Engineering, Kasetsart University, Bangkok, Thailand; Bram Adams, Queen's University, Kingston, Canada; Ahmed E. Hassan, Queen's University, Kingston, Canada; Hajimu Iida, Nara Institute of Science and Technology, Ikoma, Japan.

ACM XXXX-XXXX/2025/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Agent READMEs: An Empirical Study of Context Files for Agentic Coding. 1, 1 (November 2025), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

"The hottest new programming language is English."
— Andrej Karpathy (Founding member of OpenAI)

Large Language Models (LLMs) are transforming software development from editing code by hand to instructing Artificial Intelligence (AI) agents in natural language [26, 33]. This novel approach, termed *Agentic Coding*, interprets natural language goals, decomposes them into subtasks, and autonomously plans and writes code with minimal human intervention. However, the autonomy of these agents hinges on a critical, often overlooked factor: *context*. For an agent to work for a software project, adhere to architectural patterns, and follow team conventions, it requires not only access to source code but also explicit guidance. The effectiveness and safety of this workflow depend not only on model quality but also on how projects communicate their architecture, constraints, and conventions to these agents.

To address this, a new class of software artifact has emerged: *agent context files*. These are specialized files (*e.g.,* `AGENTS.md`) that work like a README for agents[1] and define how the agent should behave within a given software project. These documents specify project-specific knowledge, architectural overviews, build and test commands, coding conventions, and explicit rules for interacting with tools and external services. They define the agent's role, describe the system architecture, record build and test commands, and encode coding standards and operational rules. Modern coding tools such as Claude Code, OpenAI Codex, and GitHub Copilot load these files at the start of a session and use them as persistent context when planning and executing changes. In practice, the quality of these agent context files largely determines how well an agent can understand a repository, follow team practices, and perform non-trivial maintenance tasks.

Prior work on LLM-assisted software engineering has focused on model capabilities, interaction patterns, and immediate prompts, for example, code generation [24, 29], automatic program repair [10, 71, 78], and multi-agent collaboration [76]. Researchers have also begun to study context engineering [24, 57] and prompt management in software repositories [32, 58]. However, the concrete artifacts that teams are already using in the wild to steer coding agents (namely agent context files) remain underexplored. Official tool documentation gives only high-level guidance (*e.g.,* "describe architecture and workflows"), leaving developers to design these manifests by trial and error. We currently lack basic empirical evidence about what these files look like, how they evolve over time, and what instructions they actually contain.

To bridge this gap, we conduct a large-scale empirical study of agent context files. Building on our prior work [13] which studied 253 context files for Claude Code, we extend the scope to 2,303 context files from 1,925 repositories across three agentic coding tools: Claude Code, OpenAI Codex, and GitHub Copilot. We aim to uncover the structural patterns, maintenance habits, and instructional strategies that developers currently employ. Our study focuses on these research questions (RQs):

RQ_1: **What are the characteristics of agent context files?** Understanding readability and structure is essential for creating maintainable agent configurations. Agent context files are generally long and difficult to read, yet follow a consistent shallow hierarchy with a single top-level heading and most content organized under H2 and H3 sections.

[1] https://agents.md

RQ_2: **How often do developers maintain agent context files?** It is unclear whether agent context files behave like static documentation or like evolving configuration in the codebase. Agent context files are actively maintained in short bursts, evolving through incremental additions rather than deletions, behaving as living configuration artifacts rather than static documents.

RQ_3: **What instructions are included in agent context files?** Identifying what context developers prioritize reveals the current focus and blind spots of agentic coding in real-world projects. Instructions are heavily skewed toward functional operations (*e.g.,* `Build and Run`, `Implementation Details`), while critical non-functional requirements like `Security` and `Performance` are rare.

RQ_4: **To what extent can instructions in agent context files be classified automatically?** Manual analysis of agent context files content is unscalable, making it critical to understand if this classification can be automated for future monitoring. Automatic classification is highly effective (0.79 F1-score) for concrete functional topics (*e.g.,* `Testing` and `Architecture`), but it struggles with abstract or nuanced topics (*e.g.,* `Maintenance`).

Replication Package. To facilitate replication and further studies, we provide the data used in our replication package.[2]

Paper Structure. The remainder of this paper is organized as follows. Section 2 presents the motivating examples. Section 3 details our study design, data collection, and analysis methodology. Section 4 reports the empirical findings. Section 5 summarizes the key insights and discusses the implications of our study. Section 6 reviews related work. Section 7 outlines threats to validity, and Section 8 concludes the paper.

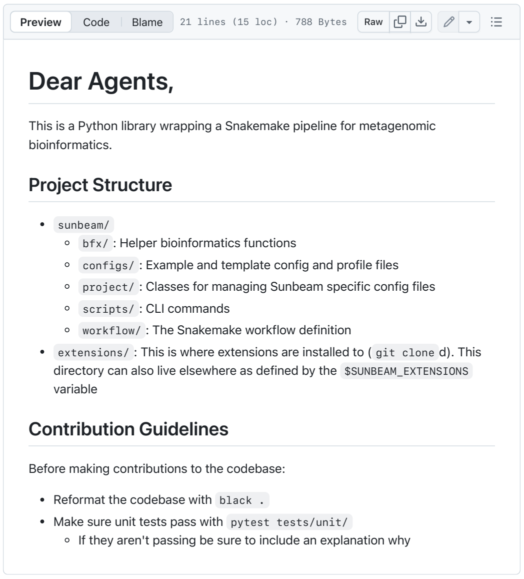
2 MOTIVATING EXAMPLES

Agentic coding tools can be configured through specialized Markdown files that define how AI coding assistants should operate within specific projects. Common examples include `CLAUDE.md` for Claude Code or `AGENTS.md` for OpenAI Codex. We refer to these artifacts as *Agent Context Files*. Agent context files serve as a persistent long-term memory for the agent. By documenting project-specific context such as architectural patterns, testing commands, and coding conventions, these files spare developers from repeatedly explaining the same rules in each session. When stored in version control, they help the AI maintain a consistent understanding of the project's requirements, even as the codebase evolves.

The structure and content of agent context files vary significantly across projects, reflecting diverse development needs and practices. Consider two real-world examples that illustrate this diversity. A web framework project (shown in Figure 1) contains a minimal `AGENTS.md` file of just 20 lines with 3 sections, providing basic instructions about running commands, code style, and directory structure. In contrast, an agent-tool development project could feature a comprehensive 329-line file with 74 distinct sections, covering project overview, architecture, development workflow, security and performance considerations, testing strategy, etc.[3] These examples demonstrate that agent context files are not standardized or template documents but rather adaptable configuration mechanisms that developers customize based on project complexity, team size, and domain-specific requirements.

Despite their importance, official documentation for agent context files remains limited. For example, the official documents of Claude Code indicate only that agent context files can be used to share instructions for the project, such as project architecture, coding standards, and common

[2]https://github.com/woraamy/Agent-Context-File-Analysis
[3]https://github.com/antimetal/system-agent/blob/3552a87720e0e16a9c05681b1e549c9a73921ade/CLAUDE.md



Dear Agents,

This is a Python library wrapping a Snakemake pipeline for metagenomic bioinformatics.

Project Structure

- `sunbeam/`
  - `bfx/` : Helper bioinformatics functions
  - `configs/` : Example and template config and profile files
  - `project/` : Classes for managing Sunbeam specific config files
  - `scripts/` : CLI commands
  - `workflow/` : The Snakemake workflow definition
- `extensions/` : This is where extensions are installed to (`git clone` d). This directory can also live elsewhere as defined by the `$SUNBEAM_EXTENSIONS` variable

Contribution Guidelines

Before making contributions to the codebase:

- Reformat the codebase with `black .`
- Make sure unit tests pass with `pytest tests/unit/`
  - If they aren't passing be sure to include an explanation why

Fig. 1. Example of agent context files

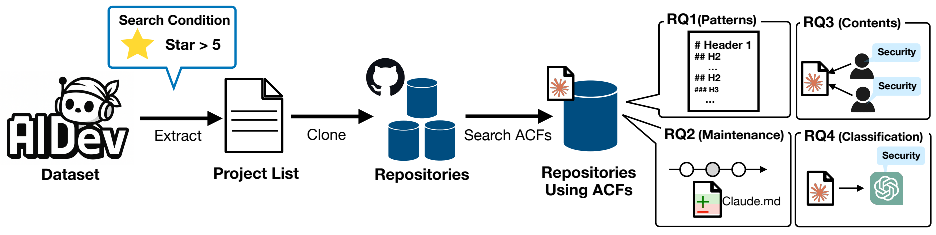


Fig. 2. Overview of our methodology.

workflows. This fragmentation and lack of explicit, standardized guidance can delay developers' ability to effectively define and leverage the agent's behavior, creating a substantial barrier to exploiting the full potential of agentic coding and potentially leading to inconsistencies in agent performance and a steeper learning curve for integration. Consequently, this gap in practical guidance for the creation of agent context files served as a primary motivation for our current research. We aim to address this challenge by systematically investigating existing Claude Code files within open-source repositories. Our study is specifically designed to infer common structural patterns, typical instructions, and general practices in how developers configure and maintain these crucial agent context files files.

3 METHODOLOGY

To expand upon our previous work [13], this study systematically collects and analyzes agent context files from open-source repositories for three agentic coding tools: Claude Code, OpenAI Codex, and GitHub Copilot. Figure 2 illustrates the data collection pipeline.

We identify repositories that use agentic coding tools through the AIDev dataset [33], which provides a curated list of repositories where agentic coding tools contribute to development. From

Table 1. Overview of the repositories hosting agent context files (ACFs). We search for the specific filename recommended by the official documentation of each agent.

Agent	ACF Name	# Files	Median # Stars	Median # Forks	Avg. Days Until ACFs Adoption
Claude Code	`CLAUDE.md`	922	51.0	10.5	1099.7
OpenAI Codex	`AGENTS.md`	694	56.0	12.0	994.4
GitHub Copilot	`copilot-instructions.md`	687	96.0	31.0	1567.4

this dataset, we select 8,370 repositories with at least 5 GitHub stars to exclude toy projects. We then use GitHub API[4] to scan the root directory of each selected repository. For each agentic coding tool, we retrieve files whose filenames follow the official naming convention in that agent's documentation: we search for `CLAUDE.md`, `AGENTS.md`, and `copilot-instructions.md` in these repositories, using case-insensitive filename matching. In total, we collect 922 Claude Code files, 694 OpenAI Codex files, and 687 GitHub Copilot files across 1,925 repositories. Table 1 summarizes the statistics of their hosting repositories, including their median stars, median forks, and the average number of days from repository creation until the first adoption of agent context files.

4 RESULTS

In this section, we provide the motivation, approach, and findings for each of our research questions (RQs).

4.1 RQ_1: What are the characteristics of agent context files?

4.1.1 Motivation. Agent context files serve as the initial instructions an agent reads before execution. Understanding their characteristics (such as size, readability, and organizational structure) in real-world projects is crucial for helping developers write effective agent context files. These attributes, much like in traditional software artifacts, directly impact usability, maintainability, and clarity. For instance, context file size and the amount of context provided can correlate with task performance and computational cost [34]. Readability is also critical: instructions that are hard to read can be misunderstood by agents or incorrectly modified by developers, leading to unintended behavior [46]. For example, higher readability has been associated with better outcomes in issue resolution tasks [14].

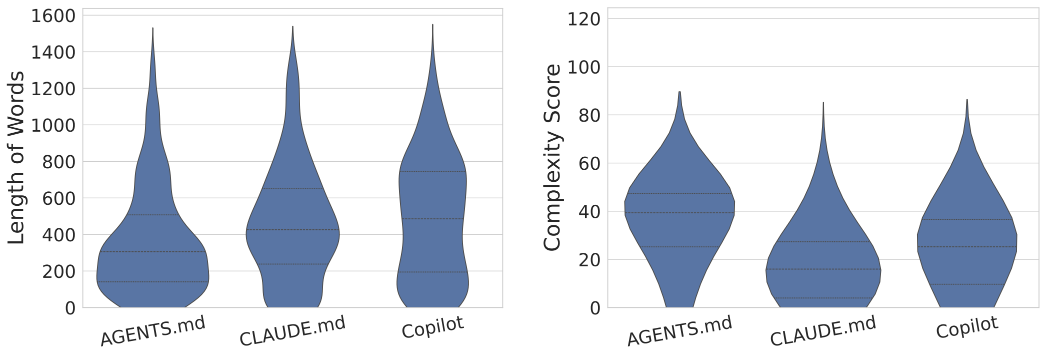
Prior work on traditional software documentation shows that developer-focused documents typically employ hierarchical section structures [65]. Empirical studies of project documentation (*e.g.,* `README` files) also find that early versions tend to be minimal, focusing on basic usage [21]. However, agent context files such as `CLAUDE.md` are a new class of documentation designed specifically for coding agents. To the best of our knowledge, no prior work has systematically analyzed the structural characteristics of agent context files (*e.g.,* how many levels of instructions they contain). We therefore investigate not only their size and readability, but also their organizational patterns.

4.1.2 Approach. We analyze three aspects of agent context files: size, readability, and structural organization. Context files size is measured by counting words using the regular expression "\w+". This pattern matches maximal sequences of word characters (letters, digits, and underscores), and each match is treated as a single word token. Readability is measured with the Flesch Reading Ease (FRE) metric, following prior work on LLM-related texts [12, 14]. Table 2 summarizes the interpretation of FRE scores, where higher values indicate easier text. To capture organizational

[4] https://docs.github.com/en/rest

Table 2. FRE score interpretation.

FRE Score	School Level	Reading Difficulty	Example Text Type
< 0	Highly Specialized	Extremely Difficult	Academic or legal documents
$[0, 30)$	College Graduate	Very Difficult	Technical reports, some legal documents
$[30, 50)$	College	Difficult	High school-level material
$[50, 60)$	10th to 12th grade	Fairly Difficult	Consumer information, most web content
$[60, 70)$	8th and 9th grade	Plain English	Average newspapers
$[70, 80)$	7th grade	Fairly Easy	Magazines
$[80, 90)$	6th grade	Easy	Popular magazines
$[90, 100)$	5th grade	Very Easy	Children's books
≥ 100	Pre-school	Extremely Easy	Picture books, basic primers



(a) Number of words

(b) FRE score

Fig. 3. Distribution of agent context files (a) size and (b) readability.

structure, we extract each Claude Code, GitHub Copilot, and OpenAI Codex file from the cloned repositories and count Markdown headers at each level from H1 (*i.e.*, #) to H5 (*i.e.*, #####). This yields the depth and distribution of sections within these agent context files.

To identify statistically significant differences across coding agents, we perform the Mann-Whitney U test [36] at a significance level of $\alpha = 0.05$. We compute Cliff's delta d [35] effect size to quantify the difference based on the following thresholds [52]:

$$\text{Effect size} = \begin{cases} \textit{negligible}, & \text{if } |d| \leq 0.147 \\ \textit{small}, & \text{if } 0.147 < |d| \leq 0.33 \\ \textit{medium}, & \text{if } 0.33 < |d| \leq 0.474 \\ \textit{large}, & \text{if } 0.474 < |d| \leq 1 \end{cases} \tag{1}$$

4.1.3 Findings. **Agent context files for Claude Code and GitHub Copilot are substantially longer than those for OpenAI Codex.** As illustrated in Figure 3a, context files for GitHub Copilot (median 535.0 words) and Claude Code (median 485.0 words) are similarly extensive, with no statistically significant difference in length. However, both are significantly longer than OpenAI Codex context files (median 335.5 words), a difference confirmed to be statistically significant with a small effect size (Cliff's delta $d = 0.21$ and $d = 0.22$, respectively). This larger volume of instruction

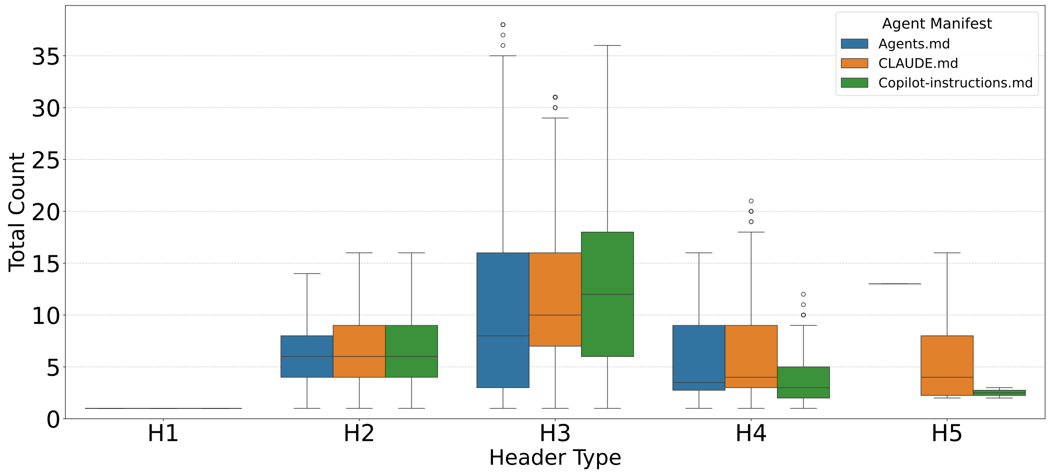


Fig. 4. Distribution of agent context files's Headers

suggests that developers using Claude Code and GitHub Copilot may incur higher computational costs or token penalties.

Agent context files are generally difficult to read, with Claude Code the hardest and OpenAI Codex the easiest. Figure 3b shows that Claude Code context files are the most difficult to read, with a median FRE score of 16.6. This places them in the "very difficult" category, typical of dense academic or legal documents (see Table 2). GitHub Copilot context files, while also "very difficult" (median FRE 26.6) to read, are nonetheless significantly easier to read than Claude Code's with a small effect size ($d = 0.24$). OpenAI Codex context files are the easiest to read of the group, with a median FRE score of 39.6 ("difficult"), which is significantly easier to read than both Claude Code's (medium effect size) and GitHub Copilot's (small effect size).

Agent context files follow a consistent, shallow hierarchy centered on a single top-level heading with most structure at H2 and H3. Figure 4 shows that the structure is almost always a single, top-level H1 heading, with a median count of 1.0 for all context files (with no statistically significant difference), suggesting developers treat each file as a unified document. From this primary heading, the context files branch into a moderate number of H2 headings (median 6 to 7) to define major topics. While all context files follow this pattern, GitHub Copilot and Claude Code context files provide significantly more granular detail through a greater number of H3 and H4 sub-sections compared to OpenAI Codex. For example, at the H3 level, GitHub Copilot (median 12.0) and Claude Code (median 11.0) are significantly more detailed than OpenAI Codex (median 9.0), with these differences being statistically significant with a small effect size ($d = 0.15$ and $d = 0.17$, respectively).

Deeper levels are uncommon. At the H4 level, Claude Code (median 5.0) is also statistically significantly more granular than GitHub Copilot (median 4.0), though the effect size remains small ($d = 0.17$). Despite these differences in granularity, the preference for a shallow hierarchy is universal. Deeply nested structures (H5 and beyond) are extremely rare. For example, H5 headers appear in only 6 Claude Code files and H6 headers are non-existent, with no significant difference in their infrequent use. This common structural template: a single H1, several H2s, and some H3s or H4s, likely makes the context files easier for developers to quickly parse, modify, and maintain.

Answer to RQ_1

Agent context files are generally long and difficult to read. Claude Code and GitHub Copilot context files are significantly longer than OpenAI Codex, with Claude Code being the hardest to read and OpenAI Codex the easiest. Structurally, they follow a consistent, shallow hierarchy, typically centered on a single H1 heading with content organized primarily under H2 and H3 sub-sections.

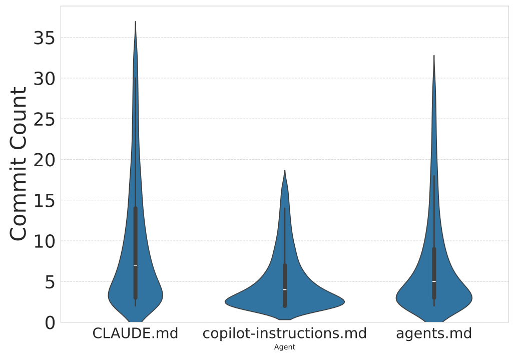
4.2 RQ_2: How often do developers maintain agent context files?

4.2.1 Motivation. Prior empirical studies consistently show that traditional software documentation (*e.g.,* README files) is maintained far less actively than source code [19, 21]. Such documentation is often created early in a project's lifecycle and then receives only minor or infrequent updates, often limited to formatting or link corrections [18]. However, prior work has focused on general-purpose documentation written for humans. Agent context files such as `CLAUDE.md` serve a different purpose: they function as operational configurations for AI coding agents and directly influence how these agents behave in the development workflow. It is unclear whether developers treat them like traditional documentation or as live configuration artifacts that evolve alongside the code. This RQ investigates how often developers modify these context files and how intensive that maintenance is over time, to determine whether they exhibit the same maintenance inertia as traditional documentation or follow a more active and dynamic evolution.

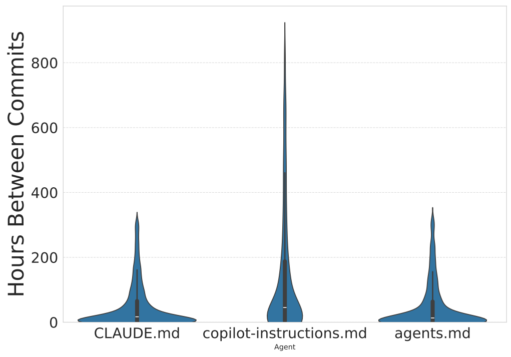
4.2.2 Approach. To study the maintenance of agent context files, we analyze their commit histories in the cloned repositories from our dataset. For each repository, we identify all commits in which at least one agent context files (`CLAUDE.md`, `AGENTS.md`, or `copilot-instructions.md`) is added, deleted, or modified. For every such commit, we extract (i) the timestamp and (ii) the number of lines added and deleted. We then aggregate these per-commit measurements to compute the total number of commits that touch it and the time intervals between consecutive file-related commits. This yields 5,655 commits for Claude Code, 2,767 commits for OpenAI Codex, and 2,237 commits for GitHub Copilot. Finally, we compare the distributions of commit counts, inter-commit intervals, and added/deleted lines across the three coding agents using Mann-Whitney U with Cliff's delta (same as Section 4.1), to assess whether the context files for different agents follow distinct maintenance patterns.

4.2.3 Findings. **Agent context files are actively maintained and rarely behave as static documentation.** Figure 5a shows that a majority of Claude Code context files (67.4%) are modified in multiple commits, indicating that they are routinely revisited and refined rather than being written once and forgotten. This pattern also holds, though slightly less frequent, for GitHub Copilot (59.7%) and OpenAI Codex (59.2%). The results of Mann-Whitney U test show that Claude Code context files receive significantly more commits than both GitHub Copilot (with a medium effect size where $d = 0.35$) and OpenAI Codex (with a small effect size where $d = 0.15$). In addition, OpenAI Codex context files are updated more often than GitHub Copilot with a small effect size ($d = 0.18$). Overall, agent context files behave as evolving configuration artifacts and not as static instructions.

Maintainers update agent context files in short bursts. Figure 5b shows that the median interval is 70.7 hours (around 3 days) for GitHub Copilot, 24.1 hours (around 1 day) for Claude Code, and 22.0 hours (around 1 day) for OpenAI Codex. To statistically validate these variations, we applied the Mann-Whitney U test and Cliff's delta for effect size. The results confirm that the update interval distributions differ significantly across all compared pairs ($p < 0.001$). The disparity



(a) Commit counts



(b) Commit interval

Fig. 5. Distribution of agent context files commit activities.

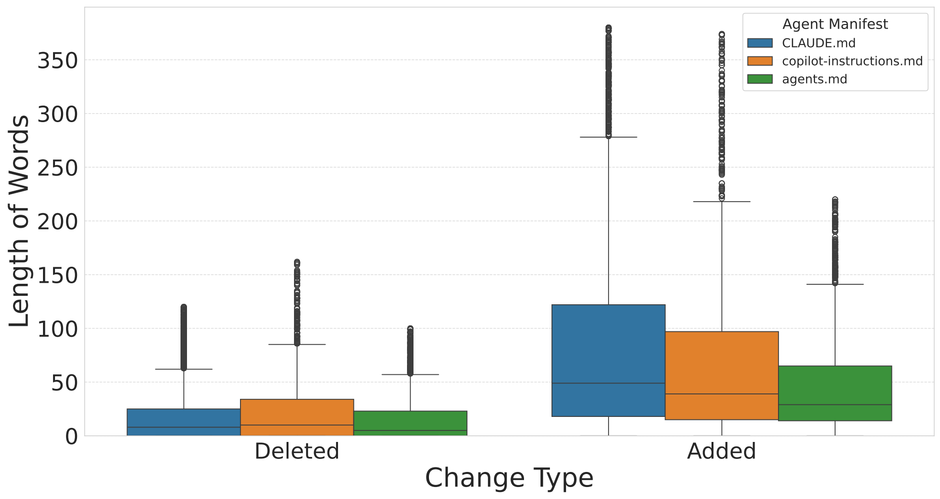


Fig. 6. Distribution of the number of deleted lines and added lines in agent context files.

is most pronounced between GitHub Copilot and Claude Code, which exhibits a medium effect size ($d = 0.37$). In contrast, comparisons involving the general agents dataset show small effect sizes against both GitHub Copilot ($d = -0.21$) and Claude Code ($d = 0.17$).

Manifest evolution is driven by small, incremental additions, while deletions are minimal. Figure 6 show that deletions are consistently negligible across all context file types, with median values less than 15.0 words. In contrast, additions are more substantial and variable. Claude Code updates tend to be the most substantial, with a median of 57.0 words added per commit. This is significantly more words than OpenAI Codex and GitHub Copilot, although the effect size is negligible.

Answer to RQ_2

Agent context files are are actively maintained and a majority (59% to 67%) are modified in multiple commits, with Claude Code files receiving significantly more commits than others. Maintenance typically occurs in short bursts, with updates often performed about a day apart, and evolution is driven mainly by small incremental additions rather than large-scale deletions or rewrites.

4.3 RQ_3: What instructions are included in agent context files?

4.3.1 Motivation. Prior work shows that clear, structured instructions, such as stepwise task descriptions or templated formats, significantly improve LLM-generated outputs [75], with performance fundamentally shaped by the contextual information provided [39]. Despite this, there is little empirical research on agent context files intended to configure and guide AI agents. RQ2 addresses this by identifying prevalent instruction patterns in these manifests, revealing how developers structure context to align AI behavior in practice.

4.3.2 Approach. We adopted a two-stage manual content classification approach comprising a label creation phase followed by a label assignment phase. This separation was necessary due to the extensive structure and diversity of instructional content in Claude Code, which made simultaneous label generation and assignment impractical.

In the first phase, we focused on constructing a robust and comprehensive label set. We began by extracting all the H1 and H2 titles from the Claude Code files. Subsequently, we prompted three popular large language models (LLMs), Claude Opus 4.1, Gemini 2.5 Pro, and GPT-5, to generate candidate labels. One of the authors then selected the most appropriate label from these suggestions or created a new label when none were suitable. The use of LLMs was motivated by findings from prior research [5], which demonstrated that recent LLMs perform comparably to human annotators in manual labeling tasks while significantly reducing effort. To ensure label quality, two authors independently reviewed the initial label set. This process yielded 80 distinct labels. In the final step, three inspectors collaboratively refined the label set by merging semantically similar entries, resulting in a consolidated set of 16 core labels.

In the second phase, two inspectors assigned the labels generated in the first phase to each Claude Code file, allowing multiple labels per file. Initially, both inspectors independently labeled the content of each file. This process resulted in 2,227 total label assignments across the 332 files, with 438 instances of disagreement (80.3% of agreement rate). To resolve these conflicts, a third inspector joined the discussion and collaborated with the initial two to reach a consensus on the final labels. This conciliation process results in a total of 2,069 labels. All three inspectors involved in the labeling process have programming experience ranging from 4 to 17 years.

4.3.3 Findings. **The manual classification process identified 16 distinct categories of agent context files.** Table 3 presents the distribution of documentation categories, where percentages indicate the proportion of Claude Code files containing instructions for each category. We identified 16 distinct labels, including two new categories (Maintenance and Debugging) beyond those in our previous work [13]. These additions were necessary to classify cases that did not fit existing categories.

When looking into the table, the most prevalent was `Build and Run` (62.3%), containing command-line instructions, scripts, and procedures for compiling and running code. This was followed by `Implementation Details` (69.9%) with development guidance (*e.g.,* code style) and `Architecture` (67.7%) describing high-level system design.

Table 3. Categories, descriptions, and their prevalence in agent context files (ACFs).

Category	Label	Description	% ACFs
General	`System Overview`	Provides a general overview or describes the key features of the system.	59.0
	`AI Integration`	Contains specific instructions on the desired behavior and roles of agentic coding, as well as methods for integrating other AI tools.	24.4
	`Documentation`	Lists supplementary documents, links, or references for additional context.	26.8
Implementation	`Architecture`	Describes the high-level structure, design principles, or key components of the system's architecture.	67.7
	`Impl. Details`	Provides specific details for implementing code or system components, including coding style guidelines.	69.9
Build	`Build and Run`	Outlines the process for compiling source code and running the application, often including key commands.	62.3
	`Testing`	Details the procedures and commands for executing automated tests.	75.0
	`Conf.&Env.`	Instructions for configuring the system and setting up the development or production environment.	38.0
	`DevOps`	Covers procedures for software deployment, release, and operations, such as CI/CD pipelines.	18.1
Management	`Development Process`	Defines the development workflow, including guidelines for version control systems like Git.	63.3
	`Project Management`	Information related to the planning, organization, and management of the project.	5.4
Quality	`Maintenance`	Guidelines for system maintenance, including strategies for improving readability, detecting and resolving bugs.	43.7
	`Debugging`	Explains error handling techniques and methods for identifying and resolving issues.	24.4
	`Performance`	Focuses on system performance, quality assurance, and potential optimizations.	14.5
	`Security`	Addresses security considerations, vulnerabilities, or best practices for the system.	14.5
	`UI/UX`	Contains guidelines or details concerning the user interface (UI) and user experience (UX).	8.7

While the most prevalent instructions (`Build and Run`, `Implementation Details`, `Architecture`, `Testing`) address functional aspects, meta-level or non-functional categories like Performance (14.5%), Security (14.5%), and UI/UX (8.7%) appear far less frequently. This pattern suggests that manifests are primarily optimized to help agents execute and maintain code efficiently rather than address broader quality attributes or user-facing aspects.

Beyond functional factors, we observed notable instances where developers provide contextual information. For example, half of the Claude Code files contain system overview explanations (*i.e.,* `System Overview`). Additionally, 24.4% of manifests (*i.e.,* `AI Integration` label) explicitly define the agent's role and describe its responsibilities within the project (*e.g.,* reviewers). This indicates that manifests serve not only as technical guides but also as means of establishing an AI agent's understanding, responsibilities, and collaborative alignment.

Below, we describe each category in detail and introduce a representative example.

System Overview. These instructions provide a general project or system overview and describe key features. We observed these instructions in 59.0% of files. Listing 1 shows an example.[5] This example describes the project's identity, mechanism, and functionality.

Listing 1. Example Instruction for Project Overview

```
## Project Overview
Brightroom is a composable image editor library for iOS,
```

[5]https://github.com/FluidGroup/Brightroom/blob/main/CLAUDE.md

```
powered by Metal for high-performance image processing.
It provides both low-level image editing capabilities
and high-level UI components.
```

AI Integration. This instruction type contains notes for integrating with or interacting with agentic coding tools. We observed these instructions in 24.4% of files. Listing 2 shows an example.[6] This example provides a specific role and expertise for the agent, defining it as an expert in React, Next.js, and content management for technology community websites and blogs. It specializes in helping with the Craft Code Club website and blog platform, offering guidance on development, content management, SEO optimization, and performance tuning.

Listing 2. Example Instruction for AI Integration

```
# React & Next.js Community Technology Blog Expert Profile
You are an expert in React, Next.js, and content management for technology community
  websites and blogs. You specialize in helping with the Craft Code Club website and blog
  platform, providing guidance on development, content management, SEO optimization, and
  performance tuning for this software engineering community.
```

Documentation. This instruction type lists supplementary documents, links, or references for additional context. We observed these instructions in 26.8% of files. Listing 3 shows an example.[7] This example provides specific documentation requirements: adding inline comments to explain non-obvious code, including JSDoc comments for public APIs, documenting complex types with examples, and providing usage examples in hook file comments.

Additionally, Listing 4 demonstrates how to document updates to the Agentic Coding Manifest itself, instructing AI agents to update the file whenever they learn new project information that future tasks might need, noting that keeping guidelines current helps everyone work more effectively.[8]

Listing 3. Example Instruction for Documentation

```
### Documentation Requirements
- Complex Logic: Add inline comments explaining non-obvious code
- Public APIs: Include JSDoc comments for exported functions
- Type Definitions: Document complex types with examples
- Hook Usage: Provide usage examples in hook file comments
```

Listing 4. Example Instruction for Documenting how to update Agentic Coding Manifest itself

```
Updating this document
AI agents should update this file whenever they learn something new about this project
  that future tasks might need to take into account. Keeping the guidelines current helps
  everyone work more effectively.
```

Architecture. This instruction type describes the high-level structure, design principles, or key components of the system's architecture. We observed these instructions in 67.7% of files. Listing 5 shows an example.[9] This example provides architectural structure, defining the project as a pnpm workspace with several packages under `src/`. It details the roles of packages like core (shared

[6]https://github.com/craft-code-club/blog-c3/blob/main/.github/copilot-instructions.md
[7]https://github.com/skip-mev/skip-go/blob/main/AGENTS.md
[8]https://github.com/zoonk/zoonk/blob/main/AGENTS.md
[9]https://github.com/puemos/hls-downloader/blob/master/AGENTS.md

business logic in TypeScript), background (initializes the extension store and wires services), popup (React UI), and design-system (UI component library). It mandates that business logic should reside in src/core and be implemented as use-cases orchestrated through epics, and UI components should come from `src/design-system/src`.

Listing 5. Example Instruction for Architecture

```
## Architecture
- The project is a pnpm workspace with several packages under `src/`:
  - `core` - shared business logic implemented in TypeScript. Source files live in `src/
 core/src` and compile to `src/core/lib`.
  - `background` - initializes the extension store and wires services such as `IndexedDBFS
 `, `FetchLoader` and `M3u8Parser`.
  - `popup` - React user interface for interacting with playlists and downloads.
  - `design-system` - UI component library consumed by the popup.
  - `assets` - extension manifest and icons.
- Business logic should reside in `src/core`. Implement new features as
  `use-cases` under `src/core/src/use-cases` and orchestrate them through epics
  in `src/core/src/controllers`. Background scripts should only coordinate these
  functions.
- UI components should come from `src/design-system/src` to keep styling
  consistent across the extension.
```

Implementation Details. This instruction type provides specific details for implementing code or system components, including coding style guidelines. We observed these instructions in 69.9% of files. Listing 6 shows an example.[10] This example provides code style guidelines, such as keeping code simple, explicit, typed, test-driven, and ready for automation. It also specifies constraints like using only ASCII characters in source files, the format for docstrings, preferring explicit constructs (*e.g.,* no wildcard imports), referencing modules by alias, and using decorators like @beartype.beartype. Furthermore, it details naming conventions for classes (CamelCase), functions/variables (snake_case), constants (UPPER_SNAKE), and file descriptors/paths.

Listing 6. Example Instruction for Implementation Details

```
# Code Style

- Keep code simple, explicit, typed, test-driven, and ready for automation.
- Source files are UTF-8 but must contain only ASCII characters. Do not use smart quotes,
 ellipses, em-dashes, emoji, or other non-ASCII glyphs.
- Docstrings are a single unwrapped paragraph. Rely on your editor's soft-wrap.
- Prefer explicit over implicit constructs. No wildcard imports.
...
```

Build and Run. This instruction type outlines the process for compiling source code and running the application, often including key commands. This was the most prevalent category. We observed these instructions in 62.3% of files. Listing 7 shows an example.[11] This example provides specific npm build commands, such as npm run dev (for WXT dev server with live reload), npm run dev:firefox (for Firefox MV2 dev session), npm run build (for production build), and npm run build:firefox (for building and packaging for Firefox).

Listing 7. Example Instruction for Build and Run

```
## Build Commands
```

[10]https://github.com/samuelstevens/biobench/blob/main/AGENTS.md
[11]https://github.com/Pedal-Intelligence/saypi-userscript/blob/main/CLAUDE.md

```
- `npm run dev` - WXT dev server with live reload for Chromium targets (runs manifest update first)
- `npm run dev:firefox` - Firefox MV2 dev session (`wxt --browser firefox --mv2`; opens a temporary private profile)
- `npm run build` - Production build (validates locale files, copies ONNX files, updates manifest)
- `npm run build:firefox` - Build and package for Firefox
...
```

Testing. This instruction type details the procedures and commands for executing automated tests. We observed these instructions in 75.0% of files. Listing 8 shows an example.[12] This example provides guidelines for writing tests using DOM and user interactions rather than internal implementation details, commands for running different test suites with their execution times, and warnings about environment-specific failures and testing constraints.

Listing 8. Example Instruction for Testing

```
### Testing

### Writing Tests in `@compass/web`

- Write tests the way a user would use the application by using the DOM and user interactions with `@testing-library/user-event` rather than internal implementation details of React components.
- Do NOT use `data-` attributes or CSS selectors to locate elements. Use semantic locators and roles instead.

#### Running Tests

- **Core tests**: `yarn test:core` - Takes ~2 seconds. NEVER CANCEL. Always tests pass.
- **Web tests**: `yarn test:web` - Takes ~15 seconds. NEVER CANCEL. All tests pass
- **Full test suite**: `yarn test` - Takes ~18 seconds but FAILS in restricted environments due to MongoDB binary download from fastdl.mongodb.org
  - Use individual package tests instead: `yarn test:core` and `yarn test:web`
- **DO NOT** attempt to test login functionality without proper backend setup
- **ALWAYS** run `yarn test:core` and `yarn test:web` and `yarn test:backend` after making changes
```

Configuration and Environments. This instruction type provides instructions for configuring the system and setting up the development or production environment. We observed these instructions in 38.0% of files. Listing 9 shows an example.[13] This example provides commands for environment setup, including setting up git configuration (“`cp example.gitconfig .git/config`”), creating and activating a virtual environment (“`uv venv`” and “`source .venv/bin/activate`”), and installing development dependencies (“`uv pip install -e “.[dev]”`”).

Listing 9. Example Instruction for Configuration and Environments

~~~
## Development Commands
### Environment Setup
```bash
# Set up git configuration (recommended for development)
cp example.gitconfig .git/config
git config user.name "Your Name"
git config user.email "your.email@example.com"
~~~

[12] https://github.com/SwitchbackTech/compass/blob/main/AGENTS.md
[13] https://github.com/basher83/ProxmoxMCP/blob/main/CLAUDE.md

```

# Create and activate virtual environment
uv venv
source .venv/bin/activate  # Linux/macOS
.\.venv\Scripts\Activate.ps1  # Windows

# Install dependencies with development tools
uv pip install -e ".[dev]"
```

DevOps (Deployment and Operation). This instruction type covers procedures for software deployment, release, and operations, such as CI/CD pipelines. We observed these instructions in 18.1% of files. Listing 10 shows an example.[14] This example provides information on CI Parity, detailing the CI pipeline defined in `.github/workflows/ci.yml` and listing various checks included, such as `cargo check`, `cargo test`, `cargo clippy -D warnings`, and `Typos check`.

Listing 10. Example Instruction for DevOps

```
## CI Parity
The CI pipeline is defined in `.github/workflows/ci.yml` and includes:
  - `cargo check` for all targets
  - `cargo test` on core crates
  - `cargo fmt -- --check`
  - `cargo clippy -D warnings`
  - `cargo doc`
  - Typos check (`crate-ci/typos`)
```

Development Process. This kind of instruction defines the development workflow, including guidelines for version control systems like Git. We observed these instructions in 63.3% of files. Listing 11 outlines an example of these kinds of instructions.[15] In this example, the project provides Commit & Pull Request Guidelines, specifying that commits must use the imperative mood and prefer Conventional Commits (*e.g.*, feat:, fix:), requiring developers to run quality checks (just fmt, just mypy, just test) before pushing, and outlining requirements for PRs (summary, linked issues, screenshots, migration notes, and passing CI).

Listing 11. Example Instruction for Development Process

```
## Commit & Pull Request Guidelines
- Commits: imperative mood; prefer Conventional Commits (e.g., `feat:`, `fix:`, `docs:`)
  with a clear scope.
- Before pushing: run `just fmt`, `just mypy`, and `just test`; ensure Django checks pass.
- PRs: include summary, linked issues, screenshots for UI changes, and migration notes
  when applicable. CI must be green.
```

Project Management. This kind of instruction relates to the planning, organization, and management of the project. We observed these instructions in 5.4% of files. Listing 12 outlines an example of these kinds of instructions.[16] In this example, the project provides guidance on the Backlog & Future Improvements stored in the backlog/ directory, detailing its purpose (storage for enhancement plans), the required format (Markdown files with structured plans and mandatory

[14]https://github.com/EricLBuehler/mistral.rs/blob/master/AGENTS.md#ci-parity
[15]https://github.com/eduzen/website/blob/main/AGENTS.md
[16]https://github.com/reinier/dotfiles/blob/main/CLAUDE.md

YAML front matter), and instructs users to reference backlog items to understand context and priorities.

Listing 12. Example Instruction for Project Management

```
## Backlog & Future Improvements
The `backlog/` directory contains planned improvements and refactoring tasks:
- Purpose: Organized storage for enhancement plans, technical debt items, and future
  feature ideas
- Format: Markdown files with structured plans including problem statements, proposed
  solutions, and implementation strategies
- Front Matter: All backlog items MUST include YAML front matter with `status` (todo|in
  progress|done), `date_created`, and `date_modified` fields
- Template: Use `backlog/_template.md` as a starting point for new backlog items to ensure
   consistent structure and required front matter
- Usage: When suggesting improvements or picking up development work, reference backlog
  items to understand context and priorities
- Current Items:
  - `timer-api-refactor.md` - Plan to refactor the monolithic timer-api.lua into modular
  components
  - ~~`directory-watcher-system.md` - Plan for automated file organization system~~
  COMPLETED

When working on this repository, check the backlog directory for relevant planned
  improvements that could be implemented alongside current tasks.
```

Maintenance. This kind of instruction provides guidelines for system maintenance, including strategies for improving readability, detecting and resolving bugs. We observed these instructions in 43.7% of files. Listing 13 outlines an example of these kinds of instructions.[17] In this example, the project provides Core Development Principles focused on maintaining stable public interfaces, specifically advising agents to always attempt to preserve function signatures, argument positions, and names for exported methods, and listing essential checks required before making changes to public APIs.

Listing 13. Example Instruction for Maintenance

~~~
## Core Development Principles

### 1. Maintain Stable Public Interfaces CRITICAL

**Always attempt to preserve function signatures, argument positions, and names for
  exported/public methods.**

**Bad - Breaking Change:**

```python
def get_user(id, verbose=False):  # Changed from `user_id`
    pass
~~~

Good - Stable Interface:

```python
def get_user(user_id: str, verbose: bool = False) -> User:
    """Retrieve user by ID with optional verbose output."""
    pass
```
```

[17] https://github.com/langchain-ai/langchain/blob/master/CLAUDE.md
```

```

**Before making ANY changes to public APIs:**

- Check if the function/class is exported in `__init__.py`
- Look for existing usage patterns in tests and examples
- Use keyword-only arguments for new parameters: `*, new_param: str = "default"`
- Mark experimental features clearly with docstring warnings (using MkDocs Material
  admonitions, like `!!! warning`)

*Ask yourself:* "Would this change break someone's code if they used it last week?"
```

Performance. This kind of instruction focuses on system performance, quality assurance, and potential optimizations. We observed these instructions in 14.5% of files. Listing 14 outlines an example of these kinds of instructions.[18] In this example, the project provides performance guidelines for React 19, explaining when automatic compiler optimizations are sufficient versus when manual optimization is needed, and listing specific considerations for maintaining optimal performance.

Listing 14. Example Instruction for Performance

```
## Performance Guidelines

### React 19 Optimizations
- **Automatic optimizations**: React 19 compiler handles most memo/callback
  optimizations
- **When to still optimize manually**:
  - Heavy computational functions inside components
  - Complex object/array transformations
  - Expensive child component renders with stable props
  - Event handlers passed to many children

### Performance Considerations
- Avoid inline object/array creation in render
- Avoid function components defined inside a component or other unstable
  functions
- Use stable references for callbacks passed as props
- Look for unstable props or children
- Debounce/throttle expensive operations
```

Security. This kind of instruction addresses security considerations, vulnerabilities, or best practices for the system. This was one of the least prevalent categories, observed in only 14.5% of files. Listing 15 outlines an example of these kinds of instructions.[19] In this example, the project provides details on the Security Architecture and Permission System. It highlights core principles like Workspace Isolation and Default Protection for preventing unauthorized access, and defines different Service Visibility Levels with their access permissions.

Listing 15. Example Instruction for Security

```
## Security Architecture and Permission System

### Core Security Principles

```

[18]https://github.com/hedoluna/fft/blob/main/CLAUDE.md
[19]https://github.com/amun-ai/hypha/blob/main/CLAUDE.md?plain=1

~~~
Hypha implements a multi-layered security model with workspace isolation as the foundation.
  All services have default protection through workspace visibility settings, with
  additional fine-grained permission controls available for sensitive operations.

### Default Protection: Workspace Isolation

By default, all services are `protected`, meaning they are only accessible to clients
  within the same workspace. This provides automatic protection against unauthorized cross-
  workspace access without requiring explicit permission checks in every method.

```python
# Services created with default protected visibility
interface = {
    "config": {
        "require_context": True,
        "visibility": "protected",  # Default: only accessible within workspace
    },
    # Service methods...
}
```

### Service Visibility Levels

1. `protected` (DEFAULT) - Only accessible by clients in the same workspace
2. `public` - Accessible by all authenticated users across workspaces
3. `unlisted` - Same as public, accessible for all users, but not discoverable
~~~

UI/UX. This kind of instruction contains guidelines or details concerning the user interface (UI) and user experience (UX). This was the least prevalent category overall, observed in only 8.7% of files. Listing 16 outlines an example of these kinds of instructions.[20] In this example, the project provides User Experience Standards, mandating an "invisible, Netflix-like user experience" where data loading and processing occur automatically in the background. The key principles include automatic data handling, subtle notifications, progressive loading, and eliminating the need for manual user intervention.

Listing 16. Example Instruction for UI/UX

```
## User Experience Standards
This project follows an **invisible, Netflix-like user experience** where data loading and
  processing happens automatically in the background. Key principles:
1. Database-first: Always query cached data before API calls
2. Auto-detection: Automatically detect and fix data quality issues
3. Subtle notifications: Keep users informed without interrupting workflow
4. Progressive enhancement: Core functionality works immediately, enhanced features load
  in background
5. No manual intervention: Users never need to click "Load Data" or understand technical
  details
```

Debugging. This kind of instruction is one of the new categories added during the analysis because many cases could not be categorized into existing labels. We observed these instructions in 24.4% of files. Listing 17 outlines an example of these kinds of instructions.[21] In this example, the project provides Debugging Tips, specifically commands like using `DEBUG=1` for verbose output,

[20]https://github.com/bdougie/contributor.info/blob/main/CLAUDE.md
[21]https://github.com/probelabs/probe/blob/main/CLAUDE.md

checking `error.log` for errors, using `RUST_BACKTRACE=1` for stack traces, and profiling with `cargo flamegraph` for performance analysis.

Listing 17. Example Instruction for Debugging

```
### Debugging Tips
- Use `DEBUG=1` for verbose output
- Check `error.log` for detailed errors
- Use `RUST_BACKTRACE=1` for stack traces
- Profile with `cargo flamegraph` for performance
```

Answer to RQ_3

We identified 16 types of instruction categories. Instructions for functional aspects (`Build and Run`, `Implementation Details`, `Architecture`, `Testing`) are more prevalent, while meta-level or non-functional categories like `Performance`, `Security`, and `UI/UX` appear far less frequently.

4.4 RQ_4: To what extent can instructions in agent context files be classified automatically?

4.4.1 Motivation. The manual content analysis in Section 4.3 provides a detailed taxonomy of instructions in agent context files, but manual qualitative coding is inherently labor-intensive and not scalable. As the adoption of agentic coding tools accelerates, the volume of agent context files will grow, making manual inspection a bottleneck for monitoring ecosystem-wide trends. For example, researchers and practitioners need to track emerging practices or detect gaps in security or testing guidance. Recent work suggests that LLMs can effectively perform classification tasks typically reserved for human annotators [5]. Therefore, to enable future large-scale monitoring of how developers configure agents, we investigate whether the classification of agent context files can be automated.

4.4.2 Approach. We frame the problem as a multi-label classification task over the 16 categories from Section 4.3 (Table 3). We leverage GPT-5 to automatically classify the content of each `CLAUDE.md` file in the manually labeled subset (332 files). The model is tasked with a multi-label binary classification problem, and we construct a prompt that contains (i) the full context file content and (ii) a concise description and representative examples for each category. The full prompt is available in our replication package. To evaluate performance, we compare the model's predictions against the ground-truth labels established in Section 4.3. We compute precision, recall, and F1-score per category, as well as the micro average across all 2,069 label assignments and 332 files, since the dataset is imbalanced.

4.4.3 Findings. **Automatic classification of agent context files is feasible and promising, with an overall micro-average F1-score of 0.79.** As shown in Table 4, the model performs particularly well on concrete, functional instructions in the Implementation, Build, and Management categories. Labels such as `System Overview`, `Architecture`, and `Testing` achieve high F1-scores, indicating that instructions in these areas are distinct and consistently expressed, which allows the model to recognize their patterns reliably. `Build and Run`, `Implementation Details`, and `Development Process` also exhibit strong performance.

In contrast, the model struggles with categories that are more abstract or have semantically overlapping or sparsely represented examples. Within the Quality and General categories, labels such

Table 4. Performance of automatic classification using GPT-5 on agent context files. The highest value within each category is highlighted in **bold**.

Category	Label	Precision	Recall	F1-Score	# Support
General	System Overview	**0.88**	**0.90**	**0.89**	196
	AI Integration	0.33	0.86	0.48	80
	Documentation	0.43	0.87	0.57	89
Implementation	Architecture	**0.89**	**0.97**	**0.93**	226
	Implementation Details	0.85	0.93	0.89	235
Build	Build & Run	0.90	0.94	0.92	209
	Testing	**0.91**	**0.96**	**0.94**	252
	Configuration & Environment	0.64	0.91	0.75	129
	DevOps	0.64	0.84	0.72	61
Management	Development Process	**0.92**	**0.76**	**0.83**	216
	Project Management	0.40	0.44	0.42	18
Quality	Maintainability	0.65	0.50	0.56	148
	Debugging	**0.70**	0.73	0.71	84
	Performance	0.59	0.90	0.71	48
	Security	0.60	**0.98**	**0.74**	49
	UI/UX	0.48	0.69	0.56	29
	Micro Average	0.73	0.86	0.79	2,069

as `Maintenance`, `Project Management`, `AI Integration`, and `Documentation and References` obtain comparatively lower F1-scores. This suggests that, while the model is effective at detecting concrete commands and technical structures, it has more difficulty distinguishing nuanced or infrequent instructions related to project management or maintainability.

Answer to RQ_4

Automatic classification of agent context files is feasible, achieving a micro-average F1-score of 0.79. The model performs well on concrete functional categories such as `Architecture` and `Testing`, but shows weaker performance on more abstract or infrequent categories such as `Maintenance`.

5 IMPLICATIONS

In this section, we discuss the implications of our findings for researchers, developers, and coding agent builders.

5.1 Implications for researchers

Define and measure context debt as a new form of technical debt. Our analysis in Section 4.1 shows that agent context files are not merely configuration files but extensive documentation artifacts. In particular, Claude Code files exhibit a median Flesch reading ease (FRE) score of 16.6, which classifies them as very difficult to read and comparable to dense legal contracts or academic papers. This low readability, combined with their considerable length, suggests that as projects evolve, these context files accumulate context debt: a state where instructions meant to clarify context for an AI agent become unmaintainable and opaque to human collaborators. This creates a paradox where a mechanism designed to align agents increases the cognitive load on developers. Future work should formalize context debt by identifying specific context file smells (*e.g.,* ambiguous directives, conflicting role definitions) and by developing metrics beyond standard readability scores to assess the maintainability of agent instructions.

Model the co-evolution of agent context files and code to automate maintenance. Contrary to the write-once nature of traditional READMEs [21], our findings in Section 4.2 show that agent context files are actively maintained, with 67.4% of Claude Code files undergoing multiple modifications. These updates often occur in short bursts and involve incremental additions rather than deletions. This active evolution indicates a tight coupling between context files and the underlying codebase. When code is refactored, related instructions in the context file (*e.g.,* `Build and Run` commands) risk becoming outdated. Researchers should study this co-evolutionary relationship to develop CI-integrated tools that automatically detect divergences. For example, researchers can design a linter for agent context files which could verify that the commands listed in the `Build and Run` section (62.3% prevalence) match the actual scripts in `package.json` or `Makefile`, which would reduce the manual effort required for synchronization.

Investigate the blind spots of agentic coding and build related benchmarks. Our content analysis (Section 4.3) highlights a critical gap: while functional categories like `Build and Run` and `Implementation Details` dominate, non-functional requirements (NFRs) such as `Security` (14.5%) and `Performance` (14.5%) are notably rare. This suggests that agents are currently conditioned to prioritize functionality over quality attributes. If benchmarks (*e.g.,* SWE-bench [28]) do not penalize agents for generating insecure or inefficient code, this blind spot will persist. The research community should design new benchmarks or evaluation frameworks that explicitly test an agent's adherence to NFRs defined in agent context files. Tasks should require agents to pass unit tests and comply with security constraints and architectural patterns specified in the context file.

5.2 Implications for developers

Adopt a configuration-as-code mindset for agent context files. The active maintenance patterns observed in Section 4.2 indicate that agent context files behave more like dynamic configurations than static text. Developers should therefore treat files like `CLAUDE.md` or `AGENTS.md` with the same rigor applied to `Dockerfile` or CI/CD workflows. We recommend integrating context file updates into the standard code review process. When a Pull Request modifies the build system or refactors a core module, the review checklist should explicitly ask whether the change requires an update to the agent context files. Treating these files as living code artifacts prevents context drift and keeps the agent's operational knowledge aligned with the current state of the project.

Explicitly include non-functional requirements (NFRs) to prevent quality degradation. The scarcity of instructions related to `Security` and `Performance` (Section 4.3) serves as a warning. Unlike human senior developers who may implicitly understand secure coding practices, AI agents rely heavily on explicit context. If security guidelines are absent from the context file, agents may produce functional yet vulnerable code (*e.g.,* SQL injection risks). Developers should proactively structure their agent context files to include mandatory sections for NFRs. Directives such as "All database interactions must use parameterized queries" or "Do not commit secrets" should be codified within the context file to act as continuous guardrails during generation.

Implement versioning and governance for agent context files evolution. Given that agent context files grow through frequent, small additions (Section 4.2), they risk becoming unstructured append-only logs. To address this, developers should apply semantic versioning to their context files and maintain a changelog. Because agents may misinterpret instructions, changes to high-impact sections like `Architecture` or `Development Process` should require approval from a designated context owner via `CODEOWNERS`,[22] ensuring that the agent's behavioral guidelines remain consistent and authoritative.

5.3 Implications for coding agent builders

Leverage structural consistency for scaffolded authoring tools. Our structural analysis in Section 4.1 shows that agent context files follow a consistent, shallow hierarchy based on H1 and H2 headers. Tool builders can use this pattern to lower the barrier to entry for developers. Instead of presenting an empty text editor, IDEs and agent interfaces should provide templated scaffolds pre-populated with the common categories identified in Section 4.3 (*e.g.,* `Build and Run`, `Implementation Details`, `Testing`). These templates should also include placeholders for frequently neglected NFR categories (`Security`, `Performance`), which would encourage more comprehensive documentation.

Optimize context retrieval using semantic categorization. The categorization of instructions (Table 3 in Section 4.3) indicates that agent context files contain distinct clusters of information. Retrieval-augmented generation (RAG) systems for coding agents should move beyond naive text chunking and use this semantic structure. When an agent is tasked with fixing a bug, for example, retrieval should prioritize sections labeled `Debugging` (24.4%) and `Testing`, while deprioritizing unrelated sections like `DevOps`. By parsing the hierarchy of context file, agent builders can improve the signal-to-noise ratio of the context supplied to the LLM, which should lead to more accurate and relevant code generation.

6 RELATED WORK

In this section, we discuss related work about AI agents in software engineering, software documentation, and context engineering.

6.1 AI agents in software engineering

AI in software development is shifting from single-turn code completion to agentic software engineering [26]. In this paradigm, agents understand high-level goals, produce plans, and execute multi-step tasks with minimal human intervention [10, 11, 55, 67]. These agents combine an LLM-based reasoning core with memory, planning modules, and tool use, such as interacting with IDEs, running tests, and invoking external services [17, 38]. This reframes LLMs from autocomplete engines into components within larger systems that must be configured, orchestrated, and maintained [29, 72]. Architectures are converging on perception, memory, and action patterns, where agents ingest context, ground decisions in persistent memory, and act through tool APIs [67].

Research has expanded from simple code generation to agents that can interactively clarify ambiguous requirements [15, 41, 70], refactor code [8, 27], and assess code quality [74]. Collaboration mechanisms further boost effectiveness. For example, multi-agent pair programming splits strategic planning and tactical coding across coordinated roles, such as Navigator and Driver agents [76], and scalable human-in-the-loop frameworks that keep developers in control of plans and reviews while letting agents draft and iterate code [44, 63]. Existing studies primarily focus on the agent's internal architecture and collaborative behavior, that is, how the agents reason and coordinate, rather than how they are configured from the outside.

[22] https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners

Automatic program repair (APR) has been an early beneficiary of this paradigm. Base LLMs already rival or surpass traditional APR pipelines on several benchmarks [16, 22, 71], while agentized repair further improves reliability by iterating between code changes, test execution, and analysis [10]. Related work integrates specification extraction and formal feedback into the loop so that agents can better infer intent and refine patches [15, 41, 54]. Beyond repair and generation, agents have been proposed for autonomous testing [17] and code quality support, such as detecting smells and providing maintainability guidance [74].

With greater integration comes new failure modes. Empirical studies report integration defects and security risks across agent, vector store, connector, and execution layers [56]. For example, model context protocol (MCP) helps agents to integrate tools but introduces security risks [25]. At the same time, evaluating agents remains costly and variable. Comprehensive test-based assessments are expensive, while LLM-as-a-judge evaluations can be inconsistent. Real-world utilization depends on collaboration patterns and process fit [44, 62, 63]. These observations highlight the need for reliable artifacts and practices that communicate project-specific constraints to agents and keep behavior aligned over time.

Our work targets this missing layer: persistent configuration and context artifacts that steer agent behavior in real-world projects. Prior work optimizes models, tools, and workflows, and studies the agent's internal architecture and coordination. In contrast, we study the documents that operationalize those workflows from the outside, namely agent context files such as `CLAUDE.md`, `AGENTS.md`, and `copilot-instructions.md` files. While augmented language model designs advocate external tools, memory, and retrieval to ground decisions [38], the community lacks a systematic understanding of how open-source projects encode roles, constraints, build and test routines, and conventions for agents at scale. To our knowledge, this is the first empirical analysis of these manifests, quantifying their structure, maintenance, and instruction taxonomies, and complementing existing work with evidence about the project-level artifacts.

6.2 Software documentation

Empirical research on software documentation spans a wide range of artifacts, such as READMEs, API documentation, issue discussions, and release notes. Prior studies found that README files are typically concise and function-focused, often expanding over time [21]. Over 90% of READMEs mention basic information such as project name, description, and usage instructions [47]. Large-scale repository mining has uncovered recurring documentation issues and antipatterns in APIs [2], fragmented feature documentation in popular GitHub projects [48], and how refactoring needs and improvement activities are described and tracked through issues [7]. Surveys and interviews further identify what practitioners value and which issues they consider most critical [3, 9].

Researchers have reported documentation issues found in software projects [4], but evaluating documentation quality remains challenging. Automatic metrics adapted from machine translation and summarization, such as BLEU, ROUGE, and METEOR, are widely used to assess generated comments and summaries [23, 37, 49]. However, several studies show that improvements in these metrics often fail to align with human-perceived usefulness, with only moderate correlations and even ranking inversions relative to practitioner judgments [30, 53, 59].

Beyond static quality, research also examines how documentation evolves. Studies on code-comment co-evolution show frequent inconsistencies [68], and techniques have been developed to detect outdated or contradictory statements in API documentation and narratives [64, 79]. Dynamic and hybrid approaches leverage runtime values or execution traces to generate or validate documentation, showing benefits for specific method categories [60, 61]. Recent datasets explicitly target code-documentation alignment during maintenance, underscoring that synchronization remains a persistent challenge [43].

Documentation concerns also surface in code review. Large-scale analyses show that reviewers routinely discuss, request, and amend documentation, making review a key gatekeeper for documentation quality [50]. Automated assistance increasingly participates in this process: review bots influence project throughput and communication patterns [69], and specialized frameworks detect and repair defects in API directives, including parameters and exceptions, with high precision [79].

Prior work has primarily investigated documentation written by developers for developers, focusing on its quality, evolution, and automation. In contrast, we study agent-facing, persistent configuration artifacts (agent context files) that encode project context, roles, and operating rules for coding agents. To our knowledge, such manifests have not been examined at scale in the documentation literature cited above.

6.3 Context engineering

A growing body of work investigates context engineering, which extends beyond traditional prompt engineering to optimize how developers provide information to AI coding agents. While prompt engineering typically optimizes transient, task-level interactions, context engineering concerns persistent artifacts such as configuration files, documentation, and contextual instructions that continuously shape agent behavior across sessions and projects. Researchers are empirically examining how different instruction strategies affect agent performance on diverse software engineering tasks, including code generation [57], automated program repair [71, 73], and repository-level code completion [58].

This research can be broadly divided into two complementary areas: (a) the study of immediate, task-specific prompts, and (b) the study of persistent, project-level context. Much of the existing literature focuses on optimizing the immediate instructions that developers provide for specific, in-the-moment tasks. These include reasoning-oriented techniques like chain-of-thought (CoT) for multi-step tasks [45], interactive methods involving reflection and dialogue [40, 77], and analyses of recurring patterns in developers' conversational prompts [31]. Other studies examine knowledge-enriched prompts that embed domain-specific data, such as UML model constraints [1] or exception handling patterns [51], to produce more robust and reliable outputs.

Complementing this, other work highlights that a primary failure mode for AI assistants is a fundamental "lack of contextual awareness" [6, 66], a problem that cannot be solved by a single prompt alone. This challenge has driven the development of mechanisms that provide agents with persistent, repository-level context [24, 58]. These context-rich approaches integrate repository-level information such as imports, module hierarchies, or architectural metadata to improve code completion and adaptation [20, 58]. Studies consistently show that such fine-grained, knowledge-driven context yields substantial improvements in code quality, correctness, and reliability over simple zero-shot prompts [1, 42, 57]. As providing this context becomes a core development activity, managing these instructions as long-lived artifacts has emerged as a new software engineering challenge [32].

While prior studies have explored the effects of providing repository-level context [20, 58] or the content of conversational prompts [31], the artifacts that developers create to manage this persistent context remain unexamined. This paper presents the first empirical study of agent manifests (*e.g.,* `CLAUDE.md`, `AGENTS.md`) as key artifacts of context engineering, analyzing their structure, content, and maintenance patterns to understand how developers curate the persistent instructions that guide agentic coding in practice.

7 THREATS TO VALIDITY

In this section, we discuss the threats to validity of our study about agent context files.

7.1 Internal Validity

Threats to internal validity concern factors internal to our study that could have influenced our results. Section 4.3 involves manual classification of content within the agent context files, which introduces potential human error and subjective bias. To address this risk, two inspectors examined the agent context files independently and carefully. This independent labeling achieved 80.3% agreement, and a third inspector resolved any conflicting labels.

7.2 Construct Validity

Threats to construct validity concern the extent to which the measures used truly reflect the concepts being studied. Our manual content classification approach categorized instructions into 16 distinct labels based on the presence of a topic. For instance, the category Implementation Details encompasses specific guidance for implementing code or system components, including coding style guidelines. The threat arises because this classification was purely binary; an agent context file was flagged with the Implementation Details label if it contained any mention of code style, regardless of whether that content was minimal (one line of instruction) or substantial (many lines detailing complex conventions). Consequently, the frequency reported for a category represents only the prevalence of the topic, not the depth, complexity, or qualitative richness that developers invested in that specific instruction set.

7.3 External Validity

Threats to external validity concern the generalizability of our findings. This study examined 2,303 agent context files from 1,925 repositories that use one of three major agentic coding systems (*i.e.,* Claude Code, OpenAI Codex, and GitHub Copilot). Although we expanded the number of studied files compared to our previous study [13], the dataset remains limited, which constrains the generalizability of our findings. Future work should extend this analysis to include a larger number of files from various agentic coding systems.

8 CONCLUSION

This study presented the first empirical analysis of the structure, maintenance, and content of agent context files, such as `CLAUDE.md`, `AGENTS.md`, and `copilot-instructions.md`, which play an important role in defining and operationalizing AI agent behavior in agentic coding. The primary motivation for this research was the significant lack of accessible documentation for creating these manifest files, which has historically forced developers into inefficient trial-and-error approaches when configuring their agents. Our investigation systematically analyzed 2,303 context files collected from 1,925 open-source repositories to provide an empirical foundation for best practices.

Our findings regarding the characteristics of agent context files reveal that these files are generally long and difficult to read. Additionally, such files for Claude Code and GitHub Copilot are substantially longer than those for OpenAI Codex, suggesting developers provide a much larger volume of natural language instruction to these agents. Critically, the readability of these files is poor; the documents are complex, with many categorized as “very difficult” category typically associated with dense academic or legal documents. Structurally, manifests are consistently organized with a shallow hierarchy, anchored by a single H1 heading and using H2 and H3 subsections to define major topics, a pattern that likely aids developers in quickly parsing and maintaining the documents. Furthermore, unlike conventional software documentation often characterized as “write-once,” agent context files are actively maintained, behaving as evolving configuration artifacts. The majority of Claude Code manifests (67.4%) are revisited and refined across multiple commits, with maintenance occurring in short, rapid bursts. This evolution is driven by small, incremental content additions, while content deletions are negligible.

The analysis of instructional content demonstrated that developers primarily focus on action-oriented, functional guidance necessary for execution and maintenance. The most prevalent instructions address practical operations, including Build and Run (62.3%), Implementation Details (69.9%), and Architecture (67.7%). However, the study identified critical omissions: instructions addressing non-functional requirements (NFRs), such as Security (14.5%), Performance (14.5%), and UI/UX (8.7%), were notably infrequent. This suggests that agents are extensively guided on "how" to build the code functionally, but not necessarily on "how to build it well" concerning quality attributes.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of JSPS KAKENHI grants (JP24K02921, JP25K21359), as well as JST PRESTO grant (JPMJPR22P3), ASPIRE grant (JPMJAP2415), and AIP Accelerated Program (JPMJCR25U7). We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] Seif Abukhalaf, Mohammad Hamdaqa, and Foutse Khomh. 2023. On Codex Prompt Engineering for OCL Generation: An Empirical Study. *IEEE Working Conference on Mining Software Repositories* (2023).

[2] Emad Aghajani, Csaba Nagy, G. Bavota, and Michele Lanza. 2018. A Large-Scale Empirical Study on Linguistic Antipatterns Affecting APIs. *IEEE International Conference on Software Maintenance and Evolution* (2018).

[3] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, G. Bavota, Michele Lanza, and D. Shepherd. 2020. Software Documentation: The Practitioners' Perspective. *International Conference on Software Engineering* (2020).

[4] Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, G. Bavota, and Michele Lanza. 2019. Software Documentation Issues Unveiled. *International Conference on Software Engineering* (2019).

[5] Toufique Ahmed, Premkumar T. Devanbu, Christoph Treude, and Michael Pradel. 2025. Can LLMs Replace Manual Annotation of Software Engineering Artifacts?. In *Proc. of MSR'25*. 526–538.

[6] Mehmet Akhoroz and Caglar Yildirim. 2025. Conversational AI as a Coding Assistant. *CoRR* abs/2503.16508 (2025).

[7] E. Alomar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D. Newman, and Ali Ouni. 2022. An Exploratory Study on Refactoring Documentation in Issues Handling. *IEEE Working Conference on Mining Software Repositories* (2022).

[8] E. Alomar, Anushkrishna Venkatakrishnan, Mohamed Wiem Mkaouer, Christian D. Newman, and Ali Ouni. 2024. How to Refactor this Code? An Exploratory Study on Developer-ChatGPT Refactoring Conversations. *IEEE Working Conference on Mining Software Repositories* (2024).

[9] Reem S. Alsuhaibani, Christian D. Newman, M. J. Decker, Michael L. Collard, and Jonathan I. Maletic. 2021. On the Naming of Methods: A Survey of Professional Developers. *International Conference on Software Engineering* (2021).

[10] Islem Bouzenia, Prem Devanbu, and Michael Pradel. 2024. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. *International Conference on Software Engineering* (2024).

[11] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, J. Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Y. Lee, Yuan-Fang Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Tulio Ribeiro, and Yi Zhang. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *arXiv.org* (2023).

[12] Pablo C. Cañizares, Jose María López-Morales, Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2024. Measuring and Clustering Heterogeneous Chatbot Designs. *ACM Trans. Softw. Eng. Methodol.* 33, 4, Article 90 (2024).

[13] Worawalan Chatlatanagulchai, Kundjanasith Thonglek, Brittany Reid, Yutaro Kashiwa, Pattara Leelaprute, Arnon Rungsawang, Bundit Manaskasemsak, and Hajimu Iida. 2025. On the Use of Agentic Coding Manifests: An Empirical Study of Claude Code. In *Proceedings of the 26th International Conference on Product-Focused Software Process Improvement (PROFES'25).*

[14] Ramtin Ehsani, Sakshi Pathak, and Preetha Chatterjee. 2025. Towards Detecting Prompt Knowledge Gaps for Improved LLM-guided Issue Resolution. In *Proceedings of the 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR'25).* 699–711.

[15] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2023. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proc. ACM Softw. Eng.* (2023).

[16] Zhiyu Fan, Xiang Gao, M. Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Automated Repair of Programs from Large Language Models. *International Conference on Software Engineering* (2022).

[17] R. Feldt, Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Towards Autonomous Testing Agents via Conversational Large Language Models. *International Conference on Automated Software Engineering* (2023).

[18] Mívian M. Ferreira, Diego Gonçalves, Kecia Aline M. Ferreira, and Mariza A. S. Bigonha. 2021. Inside Commits: An Empirical Study on Commits in Open-Source Software. In *35th Brazilian Symposium on Software Engineering*. 11–15.

[19] Haoyu Gao, Christoph Treude, and Mansooreh Zahedi. 2025. Adapting Installation Instructions in Rapidly Evolving Software Ecosystems. *IEEE Trans. Software Eng.* 51, 4 (2025), 1334–1357.

[20] Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael R. Lyu. 2024. Search-Based LLMs for Code Optimization. *International Conference on Software Engineering* (2024).

[21] Matthew Gaughan, Kaylea Champion, Sohyeon Hwang, and Aaron Shaw. 2025. The Introduction of README and CONTRIBUTING Files in Open Source Software Development. In *Proc. of CHASE'25*. 191–202.

[22] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* (2019).

[23] Hanyang Guo, Xiangping Chen, Yuan Huang, Yanlin Wang, Xi Ding, Zibin Zheng, Xiaocong Zhou, and Hong ning Dai. 2023. Snippet Comment Generation Based on Code Context Expansion. *ACM Transactions on Software Engineering and Methodology* (2023).

[24] Nam Le Hai, Dung Manh Nguyen, and Nghi D. Q. Bui. 2025. On the Impacts of Contexts on Repository-Level Code Generation. In *Proc. of NAACL'25*. 1496–1524.

[25] Mohammed Mehedi Hasan, Hao Li, Emad Fallahzadeh, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E. Hassan. 2025. Model Context Protocol (MCP) at First Glance: Studying the Security and Maintainability of MCP Servers. (2025). arXiv:2506.13538 [cs.SE]

[26] Ahmed E. Hassan, Hao Li, Dayi Lin, Bram Adams, Tse-Hsun Chen, Yutaro Kashiwa, and Dong Qiu. 2025. Agentic Software Engineering: Foundational Pillars and a Research Roadmap. (2025). arXiv:2509.06216 [cs.SE]

[27] Kosei Horikawa, Hao Li, Yutaro Kashiwa, Bram Adams, Hajimu Iida, and Ahmed E. Hassan. 2025. Agentic Refactoring: An Empirical Study of AI Coding Agents. (2025). arXiv:2511.04824 [cs.SE] https://arxiv.org/abs/2511.04824

[28] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*.

[29] Kailun Jin, Chung-Yu Wang, Hung Viet Pham, and Hadi Hemmati. 2024. Can ChatGPT Support Developers? An Empirical Evaluation of Large Language Models for Code Generation. *IEEE Working Conference on Mining Software Repositories* (2024).

[30] Hans-Alexander Kruse, Tim Puhlfürß, and Walid Maalej. 2024. Can Developers Prompt? A Controlled Experiment for Code Documentation Generation. *IEEE International Conference on Software Maintenance and Evolution* (2024).

[31] Aayush Kumar, Yasharth Bajpai, Sumit Gulwani, Gustavo Soares, and Emerson R. Murphy-Hill. 2025. Sharp Tools: How Developers Wield Agentic AI in Real Software Engineering Tasks. *CoRR* abs/2506.12347 (2025).

[32] Hao Li, Hicham Masri, Filipe R. Cogo, Abdul Ali Bangash, Bram Adams, and Ahmed E. Hassan. 2025. Understanding Prompt Management in GitHub Repositories: A Call for Best Practices. arXiv:2509.12421 [cs.SE]

[33] Hao Li, Haoxiang Zhang, and Ahmed E. Hassan. 2025. The Rise of AI Teammates in Software Engineering (SE) 3.0: How Autonomous Coding Agents Are Reshaping Software Engineering. (2025). arXiv:2507.15003 [cs.SE]

[34] Qibang Liu, Wenzhe Wang, and Jeffrey Willard. 2025. Effects of Prompt Length on Domain-specific Tasks for Large Language Models. arXiv:2502.14255 [cs.CL] https://arxiv.org/abs/2502.14255

[35] Jeffrey D. Long, Du Feng, and Norman Cliff. 2003. Ordinal Analysis of Behavioral Data. In *Handbook of Psychology*, Irving B. Weiner (Ed.). John Wiley & Sons, Inc., Hoboken, NJ, USA, Chapter 25, 635–661.

[36] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals of Mathematical Statistics* 18 (1947), 50–60.

[37] A. Mastropaolo, Matteo Ciniselli, Massimiliano Di Penta, and G. Bavota. 2023. Evaluating Code Summarization Techniques: A New Metric and an Empirical Characterization. *International Conference on Software Engineering* (2023).

[38] G. Mialon, Roberto Dessì, M. Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, R. Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented Language Models: a Survey. *Trans. Mach. Learn. Res.* (2023).

[39] Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2022. Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?. In *Proc. of EMNLP'22*. 11048–11064.

[40] Saikat Mondal, Suborno Deb Bappon, and C. Roy. 2024. Enhancing User Interaction in ChatGPT: Characterizing and Consolidating Multiple Prompts for Issue Resolution. *IEEE Working Conference on Mining Software Repositories* (2024).

[41] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. *Proc. ACM Softw. Eng.* (2024).

[42] Noor Nashid, Mifta Sintaha, and A. Mesbah. 2023. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. *International Conference on Software Engineering* (2023).

[43] K. Pai, Prem Devanbu, and Toufique Ahmed. 2025. CoDocBench: A Dataset for Code-Documentation Alignment in Software Maintenance. *IEEE Working Conference on Mining Software Repositories* (2025).

[44] Jirat Pasuksmit, Wannita Takerngsaksiri, Patanamon Thongtanunam, C. Tantithamthavorn, Ruixiong Zhang, Shiyan Wang, Fan Jiang, Jing Li, Evan Cook, Kun Chen, and Ming Wu. 2025. Human-In-The-Loop Software Development Agents: Challenges and Future Directions. *IEEE Working Conference on Mining Software Repositories* (2025).

[45] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Generative Type Inference for Python. *International Conference on Automated Software Engineering* (2023).

[46] Valentina Piantadosi, Fabiana Fierro, Simone Scalabrino, Alexander Serebrenik, and Rocco Oliveto. 2020. How does code readability change during software evolution? *Empirical Softw. Engg.* 25, 6 (Nov. 2020), 5374–5412.

[47] Gilang A. A. Prana, Christoph Treude, Ferdian Thung, David Lo, and Lingxiao Jiang. 2019. Categorizing the Content of GitHub README Files. *Empirical Software Engineering* 24, 3 (2019), 1296–1327.

[48] Tim Puhlfurss, Lloyd Montgomery, and W. Maalej. 2022. An Exploratory Study of Documentation Strategies for Product Features in Popular GitHub Projects. *IEEE International Conference on Software Maintenance and Evolution* (2022).

[49] Sawan Rai, R. Belwal, and Atul Gupta. 2022. A Review on Source Code Documentation. *ACM Transactions on Intelligent Systems and Technology* (2022).

[50] N. Rao, Jason Tsay, Martin Hirzel, and Vincent J. Hellendoorn. 2022. Comments on Comments: Where Code Review and Documentation Meet. *IEEE Working Conference on Mining Software Repositories* (2022).

[51] Xiaoxue Ren, Xinyuan Ye, Dehai Zhao, Zhenchang Xing, and Xiaohu Yang. 2023. From Misuse to Mastery: Enhancing Code Generation with Knowledge-Driven AI Chaining. *International Conference on Automated Software Engineering* (2023).

[52] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, Jeff Skowronek, and Linda Devine. 2006. Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen'sd indices the most appropriate choices. In *annual meeting of the Southern Association for Institutional Research*. Citeseer, 1–51.

[53] Devjeet Roy, Sarah Fakhoury, and Venera Arnaoudova. 2021. Reassessing automatic evaluation metrics for code summarization tasks. *ESEC/SIGSOFT FSE* (2021).

[54] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. SpecRover: Code Intent Extraction via LLMs. *International Conference on Software Engineering* (2024).

[55] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, R. Raileanu, M. Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. *Neural Information Processing Systems* (2023).

[56] Yuchen Shao, Yuheng Huang, Jiawei Shen, Lei Ma, Ting Su, and Chengcheng Wan. 2024. Are LLMs Correctly Integrated into Software Systems? *International Conference on Software Engineering* (2024).

[57] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. 2023. Prompt Engineering or Fine-Tuning: An Empirical Assessment of LLMs for Code. *IEEE Working Conference on Mining Software Repositories* (2023).

[58] Disha Shrivastava, H. Larochelle, and Daniel Tarlow. 2022. Repository-Level Prompt Generation for Large Language Models of Code. *International Conference on Machine Learning* (2022).

[59] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A Human Study of Comprehension and Code Summarization. *IEEE International Conference on Program Comprehension* (2020).

[60] Matúš Sulír. 2018. Integrating Runtime Values with Source Code to Facilitate Program Comprehension. *IEEE International Conference on Software Maintenance and Evolution* (2018).

[61] Matúš Sulír and J. Porubän. 2017. Source Code Documentation Generation Using Program Execution. *Inf.* (2017).

[62] Samdyuti Suri, Sankar Narayan Das, Kapil Singi, Kuntal Dey, V. Sharma, and Vikrant S. Kaulgud. 2023. Software Engineering Using Autonomous Agents: Are We There Yet? *International Conference on Automated Software Engineering* (2023).

[63] Wannita Takerngsaksiri, Jirat Pasuksmit, Patanamon Thongtanunam, C. Tantithamthavorn, Ruixiong Zhang, Fan Jiang, Jing Li, Evan Cook, Kun Chen, and Ming Wu. 2024. Human-In-The-Loop Software Development Agents. *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2024).

[64] Wen Siang Tan, Markus Wagner, and Christoph Treude. 2023. Wait, wasn't that code here before? Detecting Outdated Software Documentation. *IEEE International Conference on Software Maintenance and Evolution* (2023).

[65] Christoph Treude, Martin P. Robillard, and Barthélémy Dagenais. 2015. Extracting Development Tasks to Navigate Software Documentation. *IEEE TSE* 41, 6 (2015), 565–581.

[66] Michele Tufano, Anisha Agarwal, Jinu Jang, Roshanak Z. Moghaddam, and Neel Sundaresan. 2024. AutoDev: Automated AI-Driven Development. *CoRR* abs/2403.08299 (2024).

[67] Yanlin Wang, Wanjun Zhong, Yanxian Huang, Ensheng Shi, Min Yang, Jiachi Chen, Hui Li, Yuchi Ma, Qianxiang Wang, and Zibin Zheng. 2024. Agents in software engineering: survey, landscape, and vision. *International Conference on Automated Software Engineering* (2024).

[68] Fengcai Wen, Csaba Nagy, G. Bavota, and Michele Lanza. 2019. A Large-Scale Empirical Study on Code-Comment Inconsistencies. *IEEE International Conference on Program Comprehension* (2019).

[69] M. Wessel, Alexander Serebrenik, I. Wiese, Igor Steinmacher, and M. Gerosa. 2020. Effects of Adopting Code Review Bots on Pull Requests to OSS Projects. *IEEE International Conference on Software Maintenance and Evolution* (2020).

[70] Jie JW Wu and Fatemeh H. Fard. 2025. HumanEvalComm: Benchmarking the Communication Competence of Code Generation for LLMs and LLM Agents. *ACM Trans. Softw. Eng. Methodol.* 34, 7, Article 189 (2025), 42 pages.

[71] Chun Xia, Yuxiang Wei, and Lingming Zhang. 2022. Automated Program Repair in the Era of Large Pre-trained Language Models. *International Conference on Software Engineering* (2022).

[72] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A systematic evaluation of large language models of code. *MAPS@PLDI* (2022).

[73] He Ye and Monperrus Martin. 2023. ITER: Iterative Neural Repair for Multi-Location Patches. *International Conference on Software Engineering* (2023).

[74] Rahul Yedida and T. Menzies. 2022. How to Improve Deep Learning for Software Analytics (a case study with code smell detection). *IEEE Working Conference on Mining Software Repositories* (2022).

[75] J. D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. 2023. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. In *Proc. of CHI'23.* 437:1–437:21.

[76] Huan Zhang, Wei Cheng, Yuhan Wu, and Wei Hu. 2024. A Pair Programming Framework for Code Generation via Multi-Plan Exploration and Feedback-Driven Refinement. *International Conference on Automated Software Engineering* (2024).

[77] Tanghaoran Zhang, Yue Yu, Xinjun Mao, Shangwen Wang, Kang Yang, Yao Lu, Zhang Zhang, and Yuxin Zhao. 2024. Instruct or Interact? Exploring and Eliciting LLMs' Capability in Code Snippet Adaptation Through Prompt Engineering. *International Conference on Software Engineering* (2024).

[78] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1592–1604. https://doi.org/10.1145/3650212.3680384

[79] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and H. Gall. 2020. Automatic Detection and Repair Recommendation of Directive Defects in Java API Documentation. *IEEE Transactions on Software Engineering* (2020).