On the Use of Agentic Coding Manifests: An Empirical Study of Claude Code

Worawalan Chatlatanagulchai[1], Kundjanasith Thonglek[1], Brittany Reid[2], Yutaro Kashiwa[2], Pattara Leelaprute[1], Arnon Rungsawang[1], Bundit Manaskasemsak[1], and Hajimu Iida[2]

[1] Faculty of Engineering, Kasetsart University, Thailand
[2] Nara Institute of Science and Technology (NAIST), Japan

Abstract. Agentic coding tools receive goals written in natural language as input, break them down into specific tasks, and write/execute the actual code with minimal human intervention. Key to this process are agent manifests, configuration files (such as `Claude.md`) that provide agents with essential project context, identity, and operational rules. However, the lack of comprehensive and accessible documentation for creating these manifests presents a significant challenge for developers. We analyzed `253` `Claude.md` files from `242` repositories to identify structural patterns and common content. Our findings show that manifests typically have shallow hierarchies with one main heading and several subsections, with content dominated by operational commands, technical implementation notes, and high-level architecture.

Keywords: Agentic Coding, Autonomous Programming, Documents

1 Introduction

"The hottest new programming language is English," stated OpenAI founding member Andrej Karpathy,[3] capturing a fundamental shift in how software development is evolving. The rise of Large Language Models (LLMs) has enabled the deployment of Artificial Intelligence (AI) agents capable of facilitating or executing autonomous software engineering tasks through natural language interactions. This novel approach, termed `Agentic Coding`, interpret natural language goals, decompose them into subtasks, and autonomously plan and execute code with minimal human intervention. Unlike vibe coding, which focuses on describing desired feelings or essence, agentic coding provides concrete objectives and lets AI independently determine implementation through multi-step planning, tool usage, and self-correction.

Notable implementations of agentic coding tools include Claude Code, Cursor, Aider, GitHub Copilot, and Devin AI. Most of these tools rely on `Agentic coding manifests` to function effectively, which are specialized configuration files that define AI agent behavior within specific projects. Loaded at the start

[3] `https://x.com/karpathy/status/1617979122625712128`

of each session, they equip AI agents with project-specific knowledge, behavioral guidelines, and operational rules that determine how well the agent can understand codebases, interpret developer intent, and execute tasks autonomously. However, despite `agentic coding manifests` being crucial for the performance, there is little research on how to design them effectively. The lack of comprehensive documentation means developers resort to trial-and-error approaches, resulting in suboptimal agent behavior and missed opportunities to fully leverage these tools' capabilities.

This study aims to reveal common structural patterns and their contents. Our empirical study on `253` `Claude.md` files from `242` repositories revealed (i) they follow a shallow hierarchical structure with a single main heading and moderate subsections; and (ii) the most common content patterns include instructions for `Build and Run`, `Implementation Details`, and `Architecture`, highlighting the critical role of contextual information for AI-assisted software development and how action-oriented focus of `agentic coding manifests`.

2 Agent Coding Manifests

Agentic coding tools can be configured through specialized Markdown files that define how AI coding assistants should operate within specific projects. These configuration files, such as `Claude.md` for Claude Code or `AGENTS.md` for Codex, establish the AI agent's identity, capabilities, and operational workflows.[4] We refer to these configuration files as `Agentic Coding Manifests` (`ACMs`). By documenting project-specific context and conventions, `ACMs` eliminate the need for repetitive explanations and reduces misunderstandings between developers and AI assistants. When stored in version control alongside the codebase, these files ensure that AI agents maintain a consistent understanding of each project's unique requirements and characteristics throughout the development lifecycle.

There are official documents for each agentic coding tool that introduce setup for `ACMs` to pull context into prompts automatically. However, it is not comprehensively written. For example, the official documents of Claude Code indicate only that `ACMs` can be used to share instructions for the project, such as project architecture, coding standards, and common workflows.

This fragmentation and lack of explicit, standardized guidance can delay developers' ability to effectively define, orchestrate, and leverage the Claude agent's behavior, therefore 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 `ACMs` served as a primary motivation for our current research. We aim to address this challenge by systematically investigating existing `Claude.md` files within open-source repositories. Our study is specifically designed to infer common structural patterns, typical instructions, and general practices in how developers configure these crucial `agentic coding manifests` and maintain them.

[4] An example can be seen here: `https://github.com/fschutt/azul/blob/3fe83b9d4c8004ebe96ea0a77660c777fcd05bc8/CLAUDE.md`

3 Data Collection

This study used a systematic data collection methodology using the GitHub API[5] to identify and analyze "`Claude.md`" files in open-source software repositories. The data collection process began by searching for repositories containing files named `Claude.md` (insensitive to case) using the GitHub API. The search focused on files created between February 24, 2025, when Claude MCP[6] was released, and June 16, 2025. This initial search identified 838 `Claude.md` files distributed in 806 different repositories.

To exclude projects that had only recently adopted Claude Code, we applied a filtering requiring repositories to have at least 20 commits after introducing their `Claude.md` file. This threshold corresponds to an average of five commits per month during the four months between the release of Claude Code and the start of data collection. After applying this filter, our dataset comprised `253` `Claude.md` files from `242` repositories. We cloned these repositories and retrieved `1,249` commits in total.

4 Results

RQ_1: How Are Agent Manifest Files Organized?

Motivation. Prior work on software documentation has noted that developer docs often use hierarchical section structures [12]. For example, Treude *et al.* observed that technical documentation "usually follows a hierarchical structure with sections and subsections" [12]. Empirical studies of project documentation (*e.g.,* `README` and `CONTRIBUTING` files) also find that early versions tend to be very minimal and focused on basic usage or contribution procedures [6]. However, `agentic coding manifests` files such as `CLAUDE.md` represent a novel documentation artifact specifically designed for AI-assisted coding. To the best of our knowledge, no prior research has systematically analyzed their structural characteristics (*e.g.,* how many and what levels of instructions they contain). We first investigate common organizational patterns that developers use when structuring instructions for AI coding agents.

Approach. We extracted each `Claude.md` file from the cloned repositories and measured the number of markdown headers that multiply nest. Specifically, we identified each header level, ranging from H1 (*i.e.,* `#`) to H6 (*i.e.,* `######`). A header's section includes all subsequent lines until the next header of the same level is found. We counted every non-empty line within these sections. Lines within code blocks (*i.e.,* demarcated by ```` ``` ````) were not included in our count.

Results. Figure 1 depicts the distribution of header levels in `Claude.md` files. The distributions indicate that most documents begin with a single primary heading (H1), with a median of 1.0. This typically branches into a moderate

[5] `https://docs.github.com/en/rest?apiVersion=2022-11-28`
[6] https://docs.anthropic.com/en/release-notes/claude-code

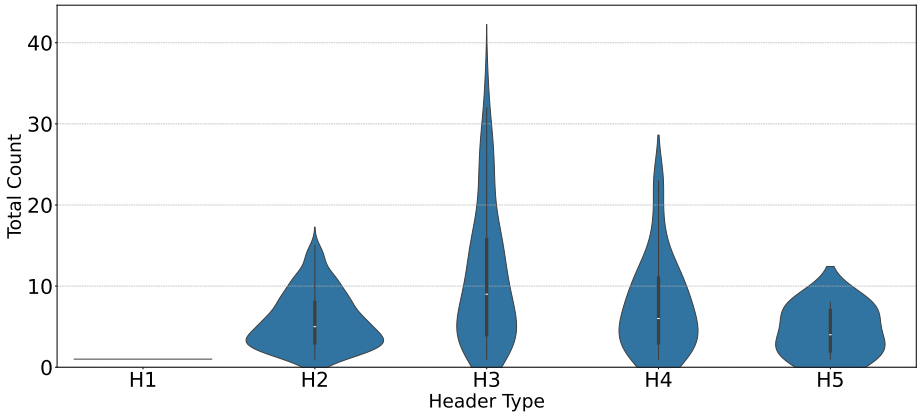

Fig. 1: Distribution of header counts across `Claude.md` files (outliers removed)

number of subsections (H2), showing a median of 5.0, followed by more granular points (H3), which have a median of 9.0.

Header usage declines sharply as depth increases. Deeper levels such as H4 and H5 occur infrequently, with H4 appearing in only 37 documents and H5 in just 5 among the 253 files analyzed. The median counts per document are 6 for H4 and 4 for H5, which indicates their limited use. We observed H6 only once in the entire corpus; given this extremely low frequency, we exclude it from further analysis. Overall, deeply nested structures are rare, indicating a preference for straightforward organization.

This organizational pattern in `agentic coding manifests` is further confirmed by the statistics for nested headers, which show a similar distribution to the overall heading counts. For example, the median for H2 headings directly nested under an H1 heading (Median = 5.0) is nearly identical to the overall statistics for all H2 headings (Median = 5.0). This consistency suggests a predictable structural approach across the dataset, where the main topics are broken down in a similar fashion. We observed distinct purposes for different heading levels. H1 headings typically encapsulate the main content topic of the `agentic coding manifests`. H2 headings commonly describe broader aspects such as coding style, project structure, command-line instructions, or overall testing strategies. As the hierarchy deepens, H3 headings become more granular, detailing specific methods of testing or how to apply those methods. Further details on the content will be addressed in the next research question.

Answer to RQ1: Agent manifest files are typically organized using a shallow hierarchical structure, with most documents starting from a single top-level heading and branching into a moderate number of H2 and H3 subsections.

RQ_2: What Instruction Are Included?

Motivation. Prior work shows that clear, structured instructions, such as stepwise task descriptions or templated formats, significantly improve LLM-generated outputs [15], with performance fundamentally shaped by the contextual information provided [9]. Despite this, there is little empirical research on `agentic coding manifests` 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.

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.md`, 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.md` files. Subsequently, we prompted three popular large language models (LLMs), Claude, Gemini, and ChatGPT, 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 [1], 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 20 core labels.

In the second phase, two inspectors assigned the labels generated in the first phase to each `Claude.md` file, allowing multiple labels per file. Initially, both inspectors independently labeled the content of each file. This process resulted in 1,228 total label assignments across the `253` files, with 113 instances of disagreement. To resolve these conflicts, a third inspector joined the discussion and collaborated with the initial two to reach a consensus on the final labels. During this reconciliation process, a new and more descriptive label (*i.e.,* `Build and Run`) was introduced. This refinement finalized our classification scheme with 15 distinct labels. All three inspectors involved in the labeling process have programming experience ranging from 4 to 17 years.

Results. Table 1 presents the distribution of documentation categories, where percentages indicate the proportion of `Claude.md` files containing instructions for each category. The most prevalent was `Build and Run` (77.1%), containing command-line instructions, scripts, and procedures for compiling and running code. This was followed by `Implementation Details` (71.9%) with development guidance (*e.g.,* code style) and `Architecture` (64.8%) describing high-level system design.

While the most prevalent instructions (`Build and Run`, `Implementation Details`, `Architecture`, `Testing`) address functional aspects, meta-level or non-functional categories like Performance (12.7%), Security (8.7%), and UI/UX

Table 1: Categories, descriptions, and their prevalence.

Category	Label	Description	%
General	`System Overview`	Provides a general overview or describes the key features of the system.	48.2
	`AI Integration`	Contains instructions or notes specifically for integrating with or interacting with the agentic coding tools.	15.4
	`Doc.&Refs`	Lists supplementary documents, links, or references for additional context.	13.8
Impl.	`Architecture`	Describes the high-level structure, design principles, or key components of the system's architecture.	64.8
	`Impl. Details`	Provides specific details for implementing code or system components, including coding style guidelines.	71.9
Build	`Build and Run`	Outlines the process for compiling source code and running the application, often including key commands.	77.1
	`Testing`	Details the procedures and commands for executing automated tests.	60.5
	`Conf.&Env.`	Instructions for configuring the system and setting up the development or production environment.	26.5
	`DevOps`	Covers procedures for software deployment, release, and operations, such as CI/CD pipelines.	9.5
Management	`Development Process`	Defines the development workflow, including guidelines for version control systems like Git.	37.2
	`Project Management`	Information related to the planning, organization, and management of the project.	11.1
Quality	`Maintainance`	Guidelines for system maintenance, including strategies for improving readability, detecting and resolving bugs.	19.8
	`Performance`	Focuses on system performance, quality assurance, and potential optimizations.	12.7
	`Security`	Addresses security considerations, vulnerabilities, or best practices for the system.	8.7
	`UI/UX`	Contains guidelines or details concerning the user interface (UI) and user experience (UX).	8.3

(8.3%) 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.md` files contain system overview explanations (*i.e.,* `System Overview`). Additionally, 15.4% 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.

Answer to RQ2: The most common content categories in `agentic coding manifests` are `Build and Run`, followed by `Implementation Details` and `Architecture` descriptions. These patterns reflect the action-oriented focus and specificity of the files.

5 Future Direction

Our future work will pursue the following two directions as well as involving more different agentic coding tools, such as Codex and Copilot.

Maintenance: The long-term efficacy and relevance of `agentic coding manifests` may be inherently tied to their maintenance. Previous research on documentation maintenance, such as Gaughan *et al.* [6] observed "burst-then-taper" effect in documentation changes. Future work could examine the evolution and decay of `Claude.md` files, revealing update frequencies, change-prone sections, and correlations with project development cycles or agent performance.

Impact: A critical direction for future research is to empirically assess the direct impact of `Claude.md` files on the performance of the Claude agent and the productivity of developers utilizing it. This could involve controlled experiments where agents are tasked with identical coding challenges, but with varying qualities or completeness of `agentic coding manifests`. Metrics such as task completion time, code quality (*e.g.,* bug count, adherence to style guides), number of iterations, and developer satisfaction could be measured. Additionally, qualitative studies, such as developer interviews or surveys, could explore how developers perceive the utility and influence of well-crafted manifests on their workflow, debugging efforts, and overall experience with agentic coding.

6 Related Work

Instructions for AI: A growing body of work investigates how developers formulate instructions for AI tools (prompt engineering) and categorizes AI agent tasks. Schulhoff *et al.* [11] investigates a wide array of prompt techniques and best practices. Kumar *et al.* [8] examine recurring instruction patterns and found that roughly 50% of developer prompts to an AI agent requested code changes, while others sought code explanations, test execution, or reviews. These studies examine typical instruction patterns that directly order AIs, while our study focuses on the contexts written in documents behind the instructions.

Contexts for AI: Many studies [4,5] consistently highlight the critical role of contextual information such as codebase [3], documentation [14], and dependencies [7] for effective AI software development assistance. Akhoroz *et al.* [2] observed that programmers often note "*inaccuracies [and] lack of contextual awareness*" in AI outputs. Tufano *et al.* [13] points out that existing assistants like Copilot "*exhibit limited functionalities and lack contextual awareness*". Our work directly addresses this need, as `agentic coding manifests` are designed to encapsulate this crucial context.

Documents for Human: Previous studies provide a strong foundation for understanding the structure and evolution of documentation. Gaughan *et al.* [6] found that initial README files are typically concise and function-focused, often expanding over time. Similarly, Prana *et al.* [10] observed that over 90% of GitHub READMEs they studied mentioned basic information like project name, description, and usage instructions. Our work extends this by quantifying the structure of the documents for AIs, like `Claude.md`.

7 Conclusion

This study analyzed `253 Claude.md` files and found that developers prefer a shallow hierarchical structure, typically using a single primary heading with a moderate number of subsections. Additionally, manual analysis revealed that manifests prioritize operational commands, followed by technical implementation notes and architecture descriptions. Many manifests also include AI role definitions to guide agent behavior. These patterns demonstrate how manifests serve dual purposes in agentic coding workflows: as execution guides for technical tasks and as frameworks for human-AI collaboration.

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).

References

1. Ahmed, T., Devanbu, P.T., Treude, C., Pradel, M.: Can llms replace manual annotation of software engineering artifacts? In: Proc. of MSR'25. pp. 526–538 (2025)
2. Akhoroz, M., Yildirim, C.: Conversational ai as a coding assistant. CoRR **abs/2503.16508** (2025)
3. Athale, M., Vaddina, V.: Knowledge graph based repository-level code generation. In: Proc. of LLM4Code'25. pp. 169–176 (2025)
4. Brézillon, P.: Context in artificial intelligence ii. key elements of contexts. Computers and Artificial Intelligence **18** (1999)
5. Calegario, F., et al.: Exploring the intersection of generative ai and software development. CoRR **abs/2312.14262** (2023)
6. Gaughan, M., Champion, K., Hwang, S., Shaw, A.: The introduction of README and CONTRIBUTING files in open source software development. In: Proc. of CHASE'25. pp. 191–202 (2025)
7. Hai, N.L., Nguyen, D.M., Bui, N.D.Q.: On the impacts of contexts on repository-level code generation. In: Proc. of NAACL'25. pp. 1496–1524 (2025)
8. Kumar, A., Bajpai, Y., Gulwani, S., Soares, G., Murphy-Hill, E.R.: Sharp tools: How developers wield agentic AI in real software engineering tasks. CoRR **abs/2506.12347** (2025)
9. Min, S., Lyu, X., Holtzman, A., Artetxe, M., Lewis, M., Hajishirzi, H., Zettlemoyer, L.: Rethinking the role of demonstrations: What makes in-context learning work? In: Proc. of EMNLP'22. pp. 11048–11064 (2022)
10. Prana, G.A.A., Treude, C., Thung, F., Lo, D., Jiang, L.: Categorizing the content of github readme files. Empirical Software Engineering **24**(3), 1296–1327 (2019)
11. Schulhoff, S., et al.: The prompt report: A systematic survey of prompt engineering techniques. CoRR **abs/2406.06608** (2024)
12. Treude, C., Robillard, M.P., Dagenais, B.: Extracting development tasks to navigate software documentation. IEEE TSE **41**(6), 565–581 (2015)
13. Tufano, M., Agarwal, A., Jang, J., Moghaddam, R.Z., Sundaresan, N.: Autodev: Automated ai-driven development. CoRR **abs/2403.08299** (2024)
14. Wang, Y., et al.: Towards an understanding of context utilization in code intelligence. CoRR **abs/2504.08734** (2025)

15. Zamfirescu-Pereira, J.D., Wong, R.Y., Hartmann, B., Yang, Q.: Why johnny can't prompt: How non-ai experts try (and fail) to design llm prompts. In: Proc. of CHI'23. pp. 437:1–437:21 (2023)