

Decoding the Configuration of AI Coding Agents: Insights from Claude Code Projects

Hélio Victor F. Santos, Vitor Costa, João Eduardo Montandon and Marco Túlio Valente
 Universidade Federal de Minas Gerais
 Belo Horizonte, Minas Gerais, Brazil
 {helio.santos,vitorcosta,joao,mtov}@dcc.ufmg.br

ABSTRACT

Agentic code assistants are a new generation of AI systems capable of performing end-to-end software engineering tasks. While these systems promise unprecedented productivity gains, their behavior and effectiveness depend heavily on configuration files that define architectural constraints, coding practices, and tool usage policies. However, little is known about the structure and content of these configuration artifacts. This paper presents an empirical study of the configuration ecosystem of Claude Code, one of the most widely used agentic coding systems. We collected and analyzed 328 configuration files from public Claude Code projects to identify (i) the software engineering concerns and practices they specify and (ii) how these concerns co-occur within individual files. The results highlight the importance of defining a wide range of concerns and practices in agent configuration files, with particular emphasis on specifying the architecture the agent should follow.

ACM Reference Format:

Hélio Victor F. Santos, Vitor Costa, João Eduardo Montandon and Marco Túlio Valente. 2025. Decoding the Configuration of AI Coding Agents: Insights from Claude Code Projects. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/nnnnnnnnnnnnnn>

1 INTRODUCTION

Agentic code assistants are autonomous AI systems that independently execute complex software development workflows. Emerging in 2024 with tools like Claude Code (Anthropic), Codex (OpenAI), and Jules (Google), these agents mark a fundamental shift from code completion tools—such as the early versions of GitHub Copilot—to systems capable of performing end-to-end software engineering tasks. Rather than requiring constant human guidance, they autonomously plan solutions, generate and refactor code, debug issues, run tests, query documentation, and perform version control operations, including creating commits and submitting pull requests.

Although agentic code assistants are built for autonomy, they should be configured with rules that guide their behavior and help to improve their performance and effectiveness. In software projects,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

© 2025 Association for Computing Machinery.
 ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00
<https://doi.org/10.1145/nnnnnnnnnnnnnn>

such rules may define the system architecture, enforce coding standards, or specify which tools—such as linters, test frameworks, or build systems—the agent can rely on. Essentially, the goal is to allow autonomy while keeping development aligned with project guidelines, tools, and best practices.

However, despite the growing popularity of agentic code assistants, little is known about how their configuration files are structured and what they reveal about agent behavior and capabilities. To address this gap, this paper explores the configuration ecosystem of Claude Code, one of the most advanced agentic coding systems. We collected and analyzed a dataset of 328 configuration files from publicly available and popular Claude Code projects. Using this dataset, we seek to answer two research questions:

- *RQ1: What concerns and practices are specified in agent configuration files?* Our goal is to help developers who are using—or planning to use—agentic systems configure their code agents effectively. To achieve this goal, we believe it is important to identify the software engineering aspects and practices most frequently specified in the configuration files of agentic coding systems. In particular, we focus on the configuration files of a widely used system, Claude Code.
- *RQ2: In addition to textual rules, do configuration files include other elements such as code examples, links, and diagrams?* The goal is to determine whether these elements—which have specific tags in markdown—are also used by developers when writing configuration files for code agents.
- *RQ3: What are the most common patterns of Claude.md files?* With this RQ we intend to provide a list of typical agent configuration files so developers can choose which setup best fits their needs. For this, we analyzed which concerns and practices co-occur in the configuration files analyzed in RQ1, and reveal the most common patterns.

The remainder of this paper is organized as follows. Section 2 introduces the Claude Code tool, and Section 3 describes our study design, including data collection, manual analysis, and pattern analysis. Sections 4, 5, and 6 present the results for each research question. Finally, Section 7 discusses threats to validity, Section 8 reviews related work, and Section 9 concludes the paper.

2 CLAUDE CODE

Claude Code is a command line tool for agentic coding created by Anthropic, and designed to assist developers in their coding workflows.¹ Among its features, the tool can read and understand the codebase of a project, integrate with the local development environment to automate tasks, and manage Git workflows.

¹<https://www.anthropic.com/news/clause-3-7-sonnet>

To make best use of Claude Code, developers can provide specific instructions about their project and how the agent should interact with it. These instructions are defined in a configuration file named `Claude.md` written in markdown format. For each request, Claude first reads the content of this file looking for information that can help it at fulfilling the request. These files contain a wide range of information, including general source code style guides, project's directory structure, and a list of commands to build, test, and run the project. Figure 1 presents a simple `Claude.md` file.² As we can see, the file defines the bash commands Claude Code can execute, the code style used in the project, and general workflow guidelines, such as the recommendation to run single tests whenever possible, rather than the entire test suite, for performance reasons.

```
# CLAUDE.md
This file provides guidance to Claude Code

## Bash commands
- npm run build: Build the project
- npm run typecheck: Run the typechecker

## Code style
- Use ES modules (import/export) syntax, not CommonJS (require)
- Destructure imports when possible (eg. import { foo } from 'bar')

## Workflow
- Be sure to typecheck when you're done making a series of code
  ↵ changes
- Prefer running single tests, and not the whole test suite, for
  ↵ performance
```

Figure 1: Example of `Claude.md` file.

3 STUDY DESIGN

In this section, we present the design and methodology used in the study, including the construction of the dataset and the procedures we used to answer the two RQs proposed in the research.

3.1 Data Collection

This process was carefully planned and then divided into three steps, as detailed below.

Repositories Detection. We first collected an initial list of GitHub repositories which are using Claude Code between August 28, 2025 and August 30, 2025. To this end, we combined distinct queries performed on the GitHub Search API to find repositories containing the file `Claude.md`. This initial search returned 4,724 repositories.

Repositories Selection. We selected repositories relevant to our study, i.e., projects that are popular, active and represent real-world applications. We removed repositories based on the following criteria:

- (1) *Popularity.* We filtered out repositories with less than 100 stars, which is a common proxy to select popular projects in empirical studies [2].
- (2) *Language.* We excluded non-English repositories, by checking its name and description.

²This example was extracted and adapted from Claude Code's official documentation: <https://www.anthropic.com/engineering/clause-code-best-practices>

- (3) *Actual Applications.* We removed repositories that are not real-world applications, e.g., awesome lists, tutorials, or configuration examples; this filtering step was performed manually, by analyzing the repository description and README.

From the remaining ones, we selected the top-100 most popular repositories, i.e., those with the highest number of stars. Overall, the projects are implemented in 23 programming languages, with JavaScript/TypeScript (35 projects), Python (16 projects), and Go (9 projects) being the most common ones. All projects are fairly popular (median of 950 stars) and active (median of 488 commits). They are also long-lived, with a median age of 58 months since their first commit. Some examples of selected projects include: `modelcontextprotocol/python-sdk` (the official Python SDK for implementing MCP servers and clients)³, `oven-sh/bun`⁴ (a popular JavaScript runtime environment), and `callstack/react-native-testing-library` (React Native testing library).⁵ Finally, we fetched the `CLAUDE.md` files from the selected repositories.

Fetching CLAUDE.md Files. Finally, we fetched the `CLAUDE.md` files from the selected repositories. During this process, we detected 45 files functioning as memory banks, i.e., files that only pointed to other markdown files within their respective projects. In such cases, we retrieved the contents of the referenced files instead. At the end of this process, we obtained a total of 328 `CLAUDE.md` files.

3.2 Manual Analysis

We parsed all 328 `Claude.md` files in our dataset to extract their section titles. In markdown, section titles are represented by hashtags (#), where the number of hashtags indicates the section level (e.g., # for level 1, ## for level 2, and so on). To answer the proposed RQs, we focused on level 2 sections—those starting with ##—since level 1 sections are typically used for the main title of the file (e.g., `Claude.md`, as presented in Figure 1). Thus, since Claude Code configuration files are written in Markdown, we leveraged their section structure to identify the categories of rules specified in these files, rather than analyzing each line individually.

Table 1 shows the distribution of the number of level-2 headings in the dataset. As we can see, the files have a median of seven level-2 headings. However, there is also one file with 213 headings. Interestingly, we found that 36 files have no level-2 sections at all. After a manual analysis, we concluded that these files describe practices and rules directly at level 1 (# tag). Thus, in such cases, we decided to analyze the content of level-1 headings instead.

Table 1: Size of `Claude.md` files (measured in number of ## sections).

	Min	Q1	Median	Q3	Max
## sections	0	4.0	7.0	10.0	213

After following the previous procedures, we extracted 2,492 section titles. These sections were then manually analyzed by one of the authors, who grouped them into semantically related categories according to the software engineering concerns and practices they

³<https://github.com/modelcontextprotocol/python-sdk>

⁴<https://github.com/oven-sh/bun>

⁵<https://github.com/callstack/react-native-testing-library>

refer to. For example, the sections listed below were grouped into a single practice called **Testing**: *Running Tests, Test Organization, Testing, Testing Approach, Testing Considerations, Testing Guidelines, Testing Patterns, Testing Purpose, Testing Requirements, Testing Strategy, and Testing Structure*. Next, a meeting was held with two other authors, who inspected the proposed classification and confirmed it. The final classification, with the original section titles (#² tags), the proposed classification, and a link to the respective Claude.md file is available in our replication package (link at the end of this manuscript).

3.3 Pattern Analysis

We used the FP-Max algorithm to detect which concerns and practices are mentioned together [3]. FP-Max is a variant of the Apriori algorithm [1] that focuses on finding maximal frequent itemsets, i.e., the largest independent sets of items that frequently co-occur in a dataset. We relied on this algorithm to detect independent combinations of concerns and practices that are frequently described in the same Claude.md files, thus representing possible file patterns. We represent each Claude.md file as a transaction; a tuple (or set) of the concerns and practices it contains. Each tuple is then provided to the FP-Max algorithm to identify frequently co-occurring items. We used the FP-Max implementation provided by the MLxtend library,⁶ configured with 0.15 minimum support.

4 MOST COMMON CONCERN AND PRACTICES (RQ1)

Figure 2 shows the main concerns and practices defined in coding agent configuration files. As we can see, these concerns and practices are related to the architecture of the software to be generated with the support of AI agents (found in 238 Claude.md files from our dataset, 72.6%), followed by general development guidelines (147 Claude.md files, 44.8%), project overview (128 Claude.md files, 39%), and testing guidelines (116 Claude.md files, 35.4%) and commands (109 Claude.md files, 33.2%). Other recurring themes include dependencies (30.8%), general project guidelines (25.6%), integration and usage guidelines, each with 59 Claude.md files (18%), and configuration, each with 57 Claude.md files (17.4%). In the following, we will detail these common concerns and practices and provide real examples of recommendations that can be defined within them.

Software Architecture: Essentially, these sections define the software architecture the coding agent should follow. For example, Figure 3 shows an example of such guidelines in evstack/ev-node.⁷ The guidelines define the main packages of the architecture, key interfaces, and modular design principles (e.g., *each component has an interface in the core package*).

Development Guidelines: This section aims to guide the AI agent on low-level development issues. Figure 4 shows an example from the PrefectHQ/marvin project.⁸ It includes rules for typing, installing dependencies, navigating on the code, etc.

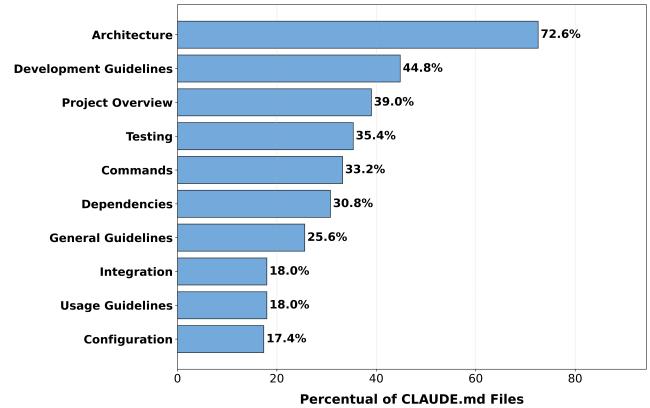


Figure 2: Concerns and practices addressed in agent config. files

Code Architecture

Core Package Structure

- The project uses a zero-dependency core package pattern:
- **core** - Contains only interfaces and types, no external dependencies
 - **block** - Block management, creation, validation, and synchronization
 - **p2p** - Networking layer built on libp2p
 - **sequencing** - Modular sequencer implementations
 - **testapp** - Reference implementation for testing

Key Interfaces

- **Executor** (core/executor.go) - Handles state transitions
- **Sequencer** (core/sequencer.go) - Orders transactions
- **DAG** (core/da.go) - Data availability layer abstraction

Modular Design

- Each component has an interface in the core package
- Implementations are in separate packages
- Components are wired together via dependency injection
- Multiple go.mod files enable modular builds

Figure 3: Architecture guidelines (evstack/ev-node)

Development Guidelines

Type Hints

- Use `X | Y` instead of `Union[X, Y]`
- Use builtins like `list`, `dict` instead of `typing.List`
- Use `typing.Dict`
- Use `T | None` instead of `Optional`

Dependencies & Running

- Use `uv` for dependency management and script execution
- Install deps: `uv sync` or `uv sync --extra foo`
- Run scripts: `uv run some/script.py` or `uv run --with pandas`
- Use `script.py`
- Testing: `uv run pytest` or `uv run pytest -n3` for parallel

Finding Things

- Use `rg` for searching, not grep
- Use `ls` and `tree` for navigation

Linter Philosophy

- Empirically understand by running code
- Linter tells basic truths but may be orthogonal to goals

Figure 4: Development guidelines (PrefectHQ/marvin)

⁶<https://rasbt.github.io/mlxtend/>

⁷<https://github.com/evstack/ev-node/blob/main/CLAUDE.md>

⁸<https://github.com/PrefectHQ/marvin/blob/main/CLAUDE.md>

Commands: This section documents commands that Claude Code can use in specific scenarios. For example, for executing tests, formatting code, and performing type checking.

Project Overview: This section is commonly used to provide Claude Code with context about the system through a general description. This helps the AI agent understand the objectives and the problem domain, resulting in more accurate code aligned with the system's needs. An example is shown in Figure 5, which clarifies the purpose and goals of the React Native Testing Library (RNTL).

Project Overview

```
This is the **React Native Testing Library (RNTL)** - a comprehensive
→ testing solution for React Native applications that provides
→ React Native runtime simulation on top of `react-test-renderer`.
→ The library encourages better testing practices by focusing on
→ testing behavior rather than implementation details.
```

Figure 5: Project overview (callstack/react-native-testing-library)

Testing: This section defines testing concerns and practices, such as in the example in Figure 6, where the configuration specifies the types of tests that the agents should create (unit, integration tests, and mocks) and the test commands.

```
## Testing
**Test Structure:**
- Unit tests for core utilities
- Integration tests for dbt operations
- Mock data for warehouse connections

**Test Commands:**
```bash
jest # Run all tests
jest --watch # Watch mode
jest src/dbt/models.test.ts # Specific test file
```

```

Figure 6: Example of testing guidelines (lightdash/lightdash)

Summary: Our results highlight the importance of defining a wide range of concerns and practices in agent configuration files, with emphasis on specifying the architecture the agent should follow when generating the application (as found in 72.6% of the analyzed Claude.md files).

5 USAGE OF CODE EXAMPLES AND LINKS (RQ2)

Table 2 presents the occurrence of source code examples and links in the analyzed Claude.md files. As shown, the highest percentage of code examples appears in the Development Guidelines category (17.68% of the instances classified in this category).

An example is provided in Figure 7, which includes a code snippet demonstrating how to configure a custom database table using Supabase.

Conversely, links were most frequently found in the Architecture category, though in only 1.83% of the files. Figure 8 illustrates an

Table 2: Usage of Code Example and Links

| | Code Examples | Links |
|------------------------|---------------|-------|
| Architecture | 10.98% | 1.83% |
| Development Guidelines | 17.68% | 0.61% |
| Testing | 15.24% | 0.0% |
| Commands | 15.55% | 0.3% |

Common Patterns

```
### Custom Table Configuration
```typescript
const vectorStore = new SupabaseVectorStore({
 supabaseUrl: process.env.SUPABASE_URL,
 supabaseKey: process.env.SUPABASE_KEY,
 table: "custom_table_name",
});
```

```

Figure 7: Config. file with code example (run-llama/LlamaIndexTS)

example containing a link to a file documenting design patterns and code conventions. Finally, we also searched for diagrams in the files—specifically UML diagrams written in the syntax of the Mermaid⁹ tool, but found only two instances.

Design patterns and Code conventions

```
Please see the dedicated "[Design patterns and Code
→ conventions](design-patterns-and-conventions.md)" page.
```

Figure 8: Config. file with a link to a project's page (grafana/mimir)

Summary: In the analyzed configuration files, there is a non-negligible presence of code examples, especially in sections that describe Development Guidelines (17.68%).

6 PATTERNS OF CONFIGURATION FILES (RQ3)

Table 3 presents the top-5 most common Claude.md patterns detected by the FP-Max algorithm. These patterns encompass combinations of two to three concerns and practices. The most common one includes practices on *Architecture*, *Dependencies*, and *Project Overview*, appearing in 21.6% of the files; the fifth pattern appears in 17.7% of the files, and contains *Architecture* and *Integration* rules. *Architecture* stands out as being the unique practice described in all top-5 patterns. *Development Guidelines* and *Project Overview* show up twice, each. The remaining ones—*Dependencies*, *General Guidelines*, *Testing*, and *Integration*—appear only once.

⁹<https://mermaid.js.org>

Table 3: Top-5 Most Popular Claude.md Patterns.

| Practices & Concerns | % Files |
|---|---------|
| Architecture, Dependencies, Project Overview
<i>Example: Repository Structure, Key Dependencies, Project Overview</i> | 21.6 |
| Architecture, General Guidelines
<i>Example: Repository Structure, Specialized Agent Usage Guidelines</i> | 20.1 |
| Architecture, Development Guidelines, Project Overview
<i>Example: Package Structure, Component Development Guidelines, Overview</i> | 19.8 |
| Architecture, Development Guidelines, Testing
<i>Example: Code Architecture, Development Workflow, Testing</i> | 18.9 |
| Architecture, Integration
<i>Example: Architecture Overview, External Integrations</i> | 17.7 |

Summary: The most common pattern of Claude.md file includes rules about Architecture, Dependencies, and Project Overview (21.6%). Moreover, Architecture is a key element in the identified patterns, appearing in the top-5 ones.

7 THREATS TO VALIDITY

Even though this is an initial study, we highlight two threats to validity. First, the classification of concerns and practices specified in the Claude.md files was performed by only one author, although it was reviewed and confirmed by two additional authors. Second, the field of AI agents is highly dynamic, meaning that our results may evolve with advances in language models and agent-based tools. However, we believe such changes will not be drastic, since the current results reflect classical Software Engineering concerns and practices, such as architecture and testing.

8 RELATED WORK

Large Language Models (LLMs) are transforming software engineering practices. Recent research describes this new era as “Software Engineering 3.0”, where AI plays a key role in increasing productivity and reducing the cognitive load of developers [4]. As developers are increasingly relying on such agents to perform software engineering tasks [6, 7, 13], investigating good practices for configuring and working with these tools becomes essential.

Research in this area has primarily focused on evaluating the effectiveness of these agents in solving software problems, using benchmarks like SWE-bench to measure performance [5]. The Agentless approach, for example proposed a three-step workflow to solve bugs and other issues using LLMs [12]. Other studies investigated the practical use of these tools and their impact on software projects. Tufano et al. [9] and Watanabe et al. [10] have investigated how developers use LLM-based chatbots, such as ChatGPT, for various software engineering tasks. Silva et al. [8] analyzed the effectiveness of ChatGPT for detecting code smells in Java Projects. Watanabe et al. [11] analyzed 567 pull requests generated by Claude Code on GitHub and highlighted the importance of incorporating project-specific rules into the agent’s instructions. Our

work proposes to investigate the guidelines that humans provide to configure software agents, thus offering a new lens on the need of collaboration between developers and AI agents.

9 CONCLUSION

This paper analyzed 328 Claude Code configuration files to understand how developers configure AI coding agents. The results show that these files capture key software engineering practices, especially those related to architectural concerns. As future work, we plan to analyze discussions about coding agent configurations in pull requests; study the evolution and modifications made to such files; and implement a tool to recommend best practices for writing agent configuration files.

Replication Data: The data and results of this research are available at: <https://doi.org/10.5281/zenodo.17388127>

ACKNOWLEDGMENTS

This research was supported by grants from CNPq and FAPEMIG.

REFERENCES

- [1] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. 1996. Fast discovery of association rules. *Advances in Knowledge Discovery and Data Mining* 12, 1 (1996), 307–328.
- [2] Hudson Borges and Marco Tulio Valente. 2018. What’s in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129.
- [3] Gösta Grahne and Jianfei Zhu. 2003. Efficiently using prefix-trees in mining frequent itemsets. In *Workshop on Frequent Itemset Mining Implementations (FIMI)*, Vol. 90. 65.
- [4] Ahmed E. Hassan, Gustavo A. Oliva, Dayi Lin, Boyuan Chen, Zhen Ming, and Jiang. 2024. Towards AI-Native Software Engineering (SE 3.0): A Vision and a Challenge Roadmap.
- [5] 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 *12th International Conference on Learning Representations (ICLR)*.
- [6] Stefano Lambiase, Gemma Catolino, Fabio Palomba, and Filomena Ferrucci. 2024. Motivations, Challenges, Best Practices, and Benefits for Bots and Conversational Agents in Software Engineering: A Multivocal Literature Review. *Comput. Surveys* 57, Article 93 (Dec. 2024), 37 pages.
- [7] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software* 203 (2023), 111734.
- [8] Luciana Lourdes Silva, Jania Rosa Da Silva, Joao Eduardo Montandon, Marcus Andrade, and Marco Tulio Valente. 2024. Detecting Code Smells Using ChatGPT: Initial Insights. In *18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 400–406.
- [9] Rosalia Tufano, Antonia Mastropaoletti, Federica Pepe, Ozren Dabic, Massimiliano Di Penta, and Gabriele Bavota. 2024. Unveiling ChatGPT’s Usage in Open Source Projects: A Mining-based Study. In *21st International Conference on Mining Software Repositories (MSR)*. 571–583.
- [10] Miku Watanabe, Yutaro Kashiwa, Bin Lin, Toshiki Hirao, Ken’ichi Yamaguchi, and Hajimu Iida. 2024. On the Use of ChatGPT for Code Review: Do Developers Like Reviews By ChatGPT? In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. 375–380.
- [11] Miku Watanabe, Hao Li, Yutaro Kashiwa, Brittany Reid, Hajimu Iida, and Ahmed E. Hassan. 2025. On the Use of Agentic Coding: An Empirical Study of Pull Requests on GitHub.
- [12] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based Software Engineering Agents.
- [13] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering.