System Prompts as Specifications: Assessing the Verifiability of Claude Code Sub-Agent Behavioral Claims

Débora Souza
Federal University of Campina Grande
Campina Grande, Paraíba, Brasil
debora.souza@ccc.ufcg.edu.br

Pedro Lima
Federal University of Campina Grande
Campina Grande, Paraíba, Brazil
pedro.lima@copin.ufcg.edu.br

Abstract

The integration of AI-powered coding assistants into software development has become a main topic in Software Engineering. Recent tools such as Claude Code increasingly support developers by automating software engineering tasks. Claude Code introduces a sub-agent mechanism that allows developers to define specialized assistants through natural-language specifications. Each sub-agent encapsulates a particular role and is described by a prompt specifying its expected behavior, execution constraints, and available tools. Despite the growing adoption of such tools, little is known about the quality of the behavioral specifications that define these sub-agents. In particular, it remains unclear whether their prescribed behaviors can be objectively verified and whether the implemented agents actually conform to these specifications during execution. To address this gap, we analyze three Claude Code sub-agent specifications from a Requirements Engineering perspective. We characterize their structure, extract behavioral claims, operationalize the notion of verifiability, and evaluate whether the observed executions conform to the specified behaviors. An analysis of three representative sub-agents identified a common specification structure composed of nine recurring sections and extracted 60 behavioral claims, of which only 11 (18.3%) were objectively verifiable. Execution-based evaluation further showed that 87% of the applicable verifiable claims were satisfied, indicating high conformance for behaviors that can be objectively assessed while highlighting substantial limitations in the verifiability of current specifications.

Keywords

AI Agents; Behavioral Specifications; Requirements Engineering; Verifiability; Claude Code

1 Introduction

The integration of Artificial Intelligence (AI) based agents into the software development process has established itself as one of the central topics of recent research in Software Engineering [7]. As Large Language Models (LLMs) enable increasingly autonomous software production, agents emerge as mechanisms for orchestrating these models and constraining their behavior, guiding them through the execution of workflows and the completion of complex tasks [5]. In agentic coding tools, this behavior is frequently defined through natural language specifications, such as Markdown files that establish roles, responsibilities, and constraints, directing the expected behavioral flow for these agents [3].

In practice, these specifications have become first-class development artifacts, and are shared in open source repositories, with the GitHub platform serving as the main hub for this dissemination [3]. Because these specifications shape the behavior of increasingly autonomous agents, understanding what they prescribe has become a practical concern. Recent studies have begun to empirically characterize this type of artifact, showing that agent manifests exhibit recurring patterns of organization and content [3].

Despite these advances, it remains unclear to what extent agents follow these specifications at runtime. Although widely shared and reused, these files are rarely evaluated against actual agent behavior, leaving little evidence that agents deliver on their prescribed behaviors. Two questions remain open: whether these behaviors are objectively verifiable from execution traces and whether agents actually conform to them at runtime. Thus, a gap persists between the widespread adoption of these specifications and evidence that their behavioral claims hold in practice.

Against this backdrop, this study focuses on the GitHub ecosystem and investigates the awesome-claude-code-subagents [1] repository, which catalogs more than 150 sub-agents organized into different categories and which, as of July 2026, had nearly 23,000 stars, 2,700 forks, and steady activity, indicating that the phenomenon is real and reflects the growing adoption of this type of artifact in practice. Drawing on this corpus, this paper studies whether the behavior these specifications prescribe holds when the agents are actually executed.

This paper makes three contributions. First, it characterizes the structure and recurring elements of Claude Code sub-agent specifications. Second, it proposes operational criteria for assessing the verifiability of their behavioral claims. Third, it empirically evaluates conformance between prescribed and observed behavior during execution. Together, these contributions assess not only how agent specifications are structured, but also whether agents adhere to them at runtime.

2 Related Work

LLM use in Software Engineering is well documented, with systematic reviews covering Software Engineering tasks [6] and surveys examining LLM-based agents [7]. Multi-agent frameworks such as MetaGPT [5] and industry systems that coordinate specialized sub-agents through a main agent [1] show that structured roles and workflows improve coordination and reduce errors in complex tasks. However, they treat role definition as part of system design rather than as a shareable artifact for empirical analysis.

Closer to this work, recent research has begun to empirically characterize these configuration files. Studies show that manifests follow shallow hierarchies dominated by operational commands and implementation notes [3], context files predominate and AGENTS.md is emerging as an interoperable standard, while

[1] https://github.com/VoltAgent/awesome-claude-code-subagents

Skills and Sub-agents remain in early adoption [4], and behavioral specifications vary widely without a uniform standard [9].

A complementary perspective understands these files as natural-language requirements artifacts [8], which makes it possible to transpose to them classical quality attributes, in particular verifiability: a requirement is verifiable only when there exists an objective procedure capable of determining whether it has been satisfied [11]. Applied to agents, this implies that an instruction whose fulfillment cannot be observed during execution cannot be reliably evaluated. LLM adherence to instructions has been measured by benchmarks such as IFEval [13], IHEval [12], and AgentIF [10], which reveal that even advanced models fail when faced with conflicting instructions or complex constraints. These studies, however, evaluate instruction-following over controlled prompts, and not over real agent specifications executed in use.

Taken together, the literature characterizes the structure and adoption of these specifications and measures models' adherence to instructions in controlled environments, but does not connect the two fronts: it remains open to what extent the behavior prescribed by real agent specifications is followed during execution. This work addresses this gap by focusing on the sub-agents cataloged in the awesome-claude-code-subagents repository, evaluating the conformance between prescribed and observed behavior and shifting the focus from characterizing the artifact to verifying its adherence at runtime.

3 Methodology

This section describes the research methodology adopted to investigate the behavioral specifications of Claude Code sub-agents.

Study Objects. This study investigates Claude Code sub-agents[2], which are specified as Markdown files containing YAML metadata followed by natural-language behavioral instructions. These specifications constitute the primary artifacts analyzed in this study. Figure 2 presents an excerpt from a representative sub-agent specification. We randomly selected three sub-agents from the Awesome Claude Code Sub-agents repository[3]—*prompt-engineer*, *rails-expert*, and *test-automator*—for the analyses conducted in RQ2 and RQ3 (Table 1), while all repository specifications were analyzed for the structural characterization performed in RQ1.

Table 1: Selected Claude Code sub-agents.

Sub-agent	Primary responsibility
prompt-engineer	Design, optimize, test, and evaluate prompts for production LLM applications.
rails-expert	Develop and modernize Ruby on Rails applications, including APIs, background jobs, real-time features, and deployment automation.
test-automator	Build automated test frameworks, implement test suites, and integrate testing into CI/CD pipelines.

Experimental Design. Figure 1 summarizes the study workflow. First, three randomly selected sub-agent specifications were manually analyzed to identify recurring structural elements and derive an initial specification template. An automated analysis script then processed all 153 Markdown specifications in the repository, using regular expressions to identify these structural elements despite minor variations in section names (e.g., Workflow and Development Workflow), and computed their prevalence across the repository. Next, behavioral claims were extracted from the selected specifications and classified according to the proposed verifiability criteria. Finally, the selected sub-agents were executed on controlled tasks, and the resulting execution artifacts were compared against the extracted behavioral claims to assess conformance with their behavioral specifications.

Excerpt of the *prompt-engineer* Sub-agent Specification

YAML Metadata

```
name: prompt-engineer
description: Use this agent when you need to design, optimize,
             test, or evaluate prompts for LLMs.
tools: Read, Write, Edit, Bash, Glob, Grep
model: sonnet
```

Agent Role Description: *You are a senior prompt engineer with expertise in crafting and optimizing prompts for maximum effectiveness...*

Agent Checklist: Accuracy > 90%; Token usage optimized; Latency < 2s; Safety filters enabled

When Invoked

(1) Query context manager for use cases and LLM requirements.
(2) Review existing prompts and constraints.
(3) Analyze improvement opportunities.
(4) Implement optimized prompt engineering solutions.

Delivery Notification: *"Prompt optimization completed. Tested 47 variations achieving 93.2% accuracy with 38% token reduction..."*

Figure 2: Prompt-engineer sub-agent specification excerpt

Operationalizing Verifiability. We analyze Claude Code sub-agent prompts as behavioral specifications and adopt the concept of *verifiability* from Requirements Engineering. According to Wiegers and Beatty [11], a requirement is verifiable if an objective verification procedure can determine whether it has been correctly implemented. In the context of Claude Code sub-agents, we operationalize this concept by treating execution artifacts as the evidence available to the verifier. Accordingly, a behavioral claim is considered verifiable only if its satisfaction can be objectively determined from these artifacts.

To operationalize this definition, we define four criteria adapted to agent behavioral claims (Table 2). A behavioral claim is considered *verifiable* only if all four criteria are satisfied:

$$\text{Verifiable} = \text{Trigger} \wedge \text{ObservableEvidence} \wedge \text{ResolvableTarget} \wedge \text{Falsifiability} \quad (1)$$

Behavioral claims were extracted from each specification using Claude Sonnet 5 with the prompt shown in Figure 3. The extracted claims were independently reviewed by two authors according to the proposed criteria. Six disagreements were resolved through discussion, producing the final classification.

Experimental Setup. The study uses the awesome-claude-code-subagents repository, which contains more than 150 publicly available Claude Code sub-agent specifications written as Markdown

[2]https://code.claude.com/docs/en/sub-agents
[3]https://github.com/VoltAgent/awesome-claude-code-subagents

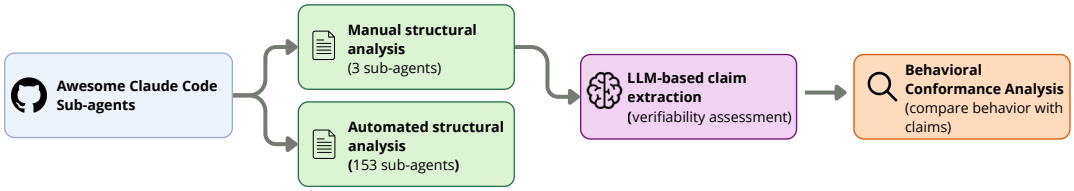

Figure 1: Workflow of agent structure analysis, claim extraction and agent execution steps.

Table 2: Criteria for behavioral claim verifiability.

Criterion	Operational Definition
Trigger	The claim specifies a clear activation condition indicating when the behavior is expected to occur.
Observable Evidence	The claim produces an observable artifact (e.g., tool call, JSON object, generated file, log entry, or response message) that provides objective evidence of execution.
Resolvable Target	All entities required to perform the behavior (e.g., tools, agents, services, or resources) are identifiable and available in the execution environment.
Falsifiability	It is possible to objectively determine that the behavior did not occur if the expected evidence is absent.

Behavioral Claim Extraction Prompt

Model: Claude Sonnet 5

"Extract every claim the agent explicitly makes about actions it will perform. Exclude domain knowledge, capability traits, and performance targets or goals. Include only statements where the agent declares it will take a concrete action, such as sending a message, calling a tool, producing output, or interacting with another agent or system." The output was required to be a CSV containing the behavioral claim, its trigger condition, and the specification section where the claim appears.

Figure 3: Prompt for behavioral claim extraction.

files with YAML metadata and natural-language behavioral instructions. Three sub-agents (prompt-engineer, rails-expert, and test-automator) were randomly selected for the qualitative analyses (RQ2 and RQ3), while all repository specifications were analyzed for the structural characterization (RQ1). Behavioral claims were extracted using Claude Sonnet 5 and subsequently reviewed by two authors according to the proposed verifiability criteria. The execution study was conducted using Claude Code v2.1.129.

3.1 Research Questions

Following the Goal-Question-Metric (GQM) approach [2], this study is guided by the following research questions:

- **RQ1: How can the behavioral specifications of Claude Code sub-agents be characterized?** Aims to identify the common structural elements and behavioral specification patterns found in Claude Code sub-agents. To answer it, we first manually analyzed sub-agent specifications to identify recurring structural elements and derive an initial specification template. We then developed a script to analyze the complete Awesome Claude Code Sub-agents repository and measure the prevalence of each identified structural element across all available specifications.
- **RQ2: To what extent are the behavioral specifications of Claude Code sub-agents verifiable?** Investigates whether the behavioral specifications can be objectively verified through observation of agent execution. Behavioral claims were automatically extracted using a large language model and subsequently analyzed according to predefined verifiability criteria.
- **RQ3: To what extent do Claude Code sub-agents conform to their behavioral specifications during execution?** Evaluates whether the observed behavior of each sub-agent conforms to its behavioral specification. Each selected sub-agent was executed manually using Claude Code version 2.1.129 under controlled scenarios, and the observed behaviors were compared against the extracted behavioral claims.

4 Results

4.1 RQ_1: Characterizing Sub-agents

To understand how Claude Code sub-agents specify behavior, we first inspected the structure of an individual specification. The specification combines YAML metadata with a series of natural-language sections that describe the agent's role description, invocation conditions, execution workflow, quality requirements, and expected outputs. Although written in natural language, the document follows a well-defined organization rather than an unstructured prompt.

After examining the three selected sub-agents, we observed that this organization is remarkably consistent across specifications. Despite targeting different domains, all analyzed agents share the same high-level structure, with recurring sections that define complementary aspects of the agent's behavior. Table 3 summarizes these common sections and their respective roles.

The analyzed specifications exhibit a layered organization that separates agent configuration, role definition, behavioral knowledge, and execution procedures. Each specification begins with YAML metadata describing the agent configuration, followed by a role description that establishes the agent's expertise and responsibilities. The core of the specification consists of behavioral sections describing domain knowledge, recommended practices, and expected capabilities, while the final sections define the agent's execution process, including invocation conditions, communication protocols, development workflow, progress reporting, quality criteria, and interactions with other agents.

After manually identifying the recurring structural elements in the three selected sub-agents, we analyzed all 153 specifications

Table 3: Common sections in Claude Code sub-agents.

Section	Description
Agent Role Description	Describes the agent's expertise, responsibilities, objectives, and overall expected behavior.
When Invoked	Specifies the conditions and initial actions that determine when the sub-agent should be delegated and how execution begins.
Agent Checklist	Lists mandatory requirements or quality criteria that should be satisfied while performing the assigned task.
Communication Protocol	Defines how the sub-agent exchanges information with the parent agent or external components, including structured messages and communication formats.
Query Context	Specifies the contextual information that should be obtained before starting the task.
Development Workflow	Organizes the execution process into sequential phases, describing the activities to be performed throughout task completion.
Progress Tracking	Defines how execution progress, intermediate results, or performance indicators should be represented and reported.
Excellence Checklist	Specifies the expected quality attributes that characterize a successful execution before task completion.
Agents Integration	Describes collaboration and delegation relationships between the sub-agent and other specialized agents.

Table 4: Prevalence of elements across 153 sub-agents.

Structural element	Agents	%
Agent Role Description	151	98.7
Agents Integration	150	98.0
Communication Protocol	132	86.3
Development Workflow	132	86.3
Progress Tracking	131	85.6
When Invoked	130	85.0
Agent Checklist	129	84.3
Query Context	121	79.1
Excellence Checklist	101	66.0
Complete structural template	95	62.1

in the repository using the automated analysis script. The results (Table 4) indicate that these structural elements are consistently adopted across the repository. Agent Role Description and Agents Integration were present in 98.7% and 98.0% of the specifications, respectively, followed by Communication Protocol (86.3%), Development Workflow (86.3%), Progress Tracking (85.6%), When Invoked (85.0%), Agent Checklist (84.3%), Query Context (79.1%), and Excellence Checklist (66.0%).

Moreover, 95 of the 153 specifications (62.1%) contained all structural elements identified during the manual analysis, suggesting that a substantial portion of the repository follows a common specification organization despite the absence of a formally defined template.

Overall, the analyzed sub-agents can be characterized as semi-structured behavioral specifications rather than conventional free-form prompts. They combine declarative elements, such as role descriptions and quality expectations, with procedural elements, including invocation conditions, execution workflows, communication guidelines, and coordination with other agents. The high prevalence of these structural elements indicates that the repository has converged toward an implicit specification template, providing a suitable foundation for the systematic extraction of behavioral claims and their subsequent verifiability and conformance analyses.

4.2 RQ_2: Sub-Agent Specification Verifiability

Table 5 summarizes the behavioral claims extracted from the three analyzed Claude Code sub-agent specifications and their corresponding verifiability classification. In total, 60 behavioral claims were extracted, of which only 11 (18.3%) were classified as verifiable according to the operational criteria defined in this study.

The distribution of claims varied across sub-agents. The *rails-expert* specification contained the largest number of behavioral claims (30), followed by *prompt-engineer* and *test-automator*, with 15 claims each. The *rails-expert* agent also exhibited the highest number of verifiable claims (9, 30.0%), whereas both *prompt-engineer* and *test-automator* contained only one verifiable claim each (6.7%). This difference is likely influenced by the behavior of the claim extraction model. The *rails-expert* specification contains many procedural statements expressed as explicit actions (e.g., Generate resources with scaffolding as a starting point"), which were consistently extracted as behavioral claims. Conversely, the *prompt-engineer* and *test-automator* specifications include several high-level activity descriptions (e.g., Design prompts") that were generally not identified by the model as behavioral claims, leading to a smaller number of extracted claims.

Behavioral Claim Verifiability Examples: rails-expert

Verifiable claim: *"Define models with associations and validations"*

✓Trigger: When the sub-agent enters the implementation phase.
✓Observable evidence: Model source files containing association and validation declarations.
✓Resolvable target: The app/models directory of the target Rails project.
✓Falsifiability: Model files defined without these declarations falsify the claim.

Non-verifiable claim: *"Read Gemfile.lock to determine Rails and Ruby versions"*

✓Trigger: When the sub-agent is assigned a Rails-related task.
× Observable evidence: No observable artifact
✓Resolvable target: Gemfile.lock file.
× Falsifiability: File reading cannot be objectively verified.

Figure 4: Example of behavioral claim verifiability

Figure 4 illustrates the application of the proposed verifiability criteria using two behavioral claims extracted from the *rails-expert* sub-agent specification. The first claim is classified as verifiable because it satisfies all four operational criteria: it specifies a clear trigger (the implementation phase), produces observable evidence in the form of association and validation declarations in model source files, refers to a resolvable target (the app/models directory of the target project), and is falsifiable, since any model defined without these declarations contradicts the claim. In contrast, the second claim is classified as non-verifiable. Although it defines a clear trigger and refers to an identifiable target (`Gemfile.lock`), it does not produce observable evidence that the file was actually read, making it impossible to objectively determine whether the behavior occurred. Since all four criteria must be satisfied for a

Table 5: Behavioral and verifiable claims by sub-agents.

Sub-agent	Claims	Verifiable	Verifiable (%)
prompt-engineer	15	1	6.7
rails-expert	30	9	30.0
test-automator	15	1	6.7
Total	**60**	**11**	**18.3**

claim to be considered verifiable, the absence of observable evidence also prevents objective falsification, resulting in a non-verifiable classification.

Overall, the results indicate that the majority of behavioral claims found in Claude Code sub-agent specifications cannot be objectively verified from execution traces alone, suggesting that these specifications frequently describe behaviors or capabilities that lack observable evidence during execution. Although Claude Code specifications contain numerous behavioral claims, only a small fraction (18.3%) are objectively verifiable under the proposed criteria. Most claims describe cognitive or qualitative behaviors that lack observable execution evidence, limiting their suitability for automated conformance assessment.

4.3 RQ_3: Execution of Sub-Agents Specifications

Initially, each selected sub-agent was manually recreated in Claude Code using its interactive agent management interface. Specifically, we accessed the /agents command, selected the Create New Agent option, and copied the agent's name, description, and system prompt from the original specification into the corresponding fields. This procedure ensured that the experimental agents faithfully reproduced the behavioral specifications available in the source repository before executing the evaluation scenarios.

After configuring the sub-agents, each agent was executed non-interactively using the Claude Code command-line interface. To enable reproducibility and subsequent analysis, Claude Code was invoked from the terminal and its standard output was captured using the tee command, which simultaneously displayed the execution in the terminal and stored the complete execution log in a file.

Each sub-agent received a single task designed to match its intended specialization. For example, the *prompt-engineer* agent was instructed to create an optimized prompt for summarizing scientific papers and save the resulting prompt to a file in the current working directory. Analogous domain-specific tasks were defined for the remaining sub-agents. The complete set of execution tasks is presented in Figure 5. The following command illustrates the execution procedure used throughout the experiments: `claude –agent agent-name "prompt" | tee -a file.log`

After executing the selected sub-agents, we manually assessed whether each behavioral claim previously classified as verifiable could indeed be objectively verified from the execution artifacts. For both *prompt-engineer* and *test-automator*, only a single behavioral claim satisfied the proposed verifiability criteria. In both cases, this claim corresponded to the primary responsibility of the agent (e.g., "Implement robust test automation solutions").

The *test-automator* sub-agent was evaluated using a simple banking application consisting of a 17-line implementation of a bank transfer operation. During execution, the agent generated a test suite containing 47 unit tests. All generated tests passed successfully and covered both nominal and edge-case scenarios, including invalid transfer amounts, insufficient funds, and self-transfer attempts.

Although the claim was considered verifiable because the agent produced an observable testing artifact, the experiment also revealed an important limitation. The specification requires the agent to implement *robust* test automation solutions, yet neither the specification nor the execution provides an objective procedure for determining whether the generated tests are, in fact, robust. Consequently, only the production of the test suite can be objectively verified, whereas the qualitative attribute "robust" remains subjective and cannot be directly assessed from the execution artifacts.

To exercise the agent *rails-expert* , we created a simple Ruby on Rails application for library management and requested the implementation of a new magazine sales feature, including a complete CRUD implementation and its corresponding automated tests. The agent successfully implemented the requested functionality and summarized the performed changes in its final response. However, verifying the behavioral claims required manually inspecting the generated and modified project files, as the execution trace itself did not provide sufficient evidence to determine whether each claim had been satisfied. Consequently, although these claims were classified as verifiable, their verification depended on post-execution inspection of the project artifacts rather than on the execution log alone.

Three behavioral claims were marked as *Not Applicable* because they were not relevant to the experimental scenario: Generate resources with scaffolding as a starting point," Create views with Hotwire or API serializers," and "Add real-time features with Turbo Streams." Since the implemented feature did not require scaffolding, Hotwire, API serializers, or real-time functionality, these claims could not be meaningfully evaluated.

Among the remaining applicable claims, one was not satisfied during execution. The claim "Document conventions and patterns" was classified as verifiable because it expects the creation or modification of documentation artifacts. However, no documentation files were created, nor was the project's `README` modified, indicating that this behavioral requirement was not fulfilled.

Overall, among the applicable behavioral claims classified as verifiable, only one was not satisfied during execution, and three are not applicable. All remaining claims could be successfully assessed using the proposed verification procedure based on the four operational criteria (Trigger, Observable Evidence, Resolvable Target, and Falsifiability) and were found to conform to their corresponding behavioral specifications. Consequently, 87% of the applicable verifiable claims were successfully implemented by the evaluated Claude Code sub-agents. These results indicate that, although only a small fraction of the behavioral specifications are objectively verifiable (RQ_2), those that are verifiable generally exhibit a high degree of conformance during execution. In other words, the primary limitation lies in the verifiability of the specifications rather than in the agents' ability to satisfy the verifiable behaviors they prescribe.

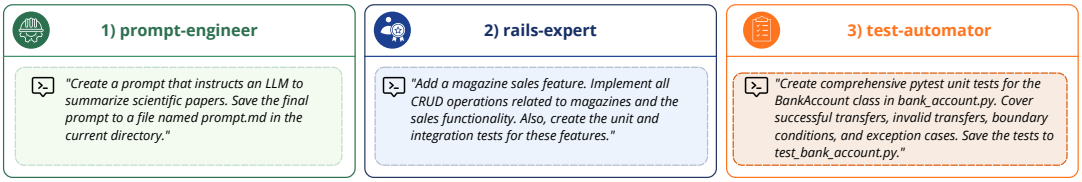

Figure 5: Tasks passed on execution of each agent.

4.4 Threats to Validity

Internal Validity. Claim extraction and verifiability assessment involve human judgment. Although two authors independently reviewed the extracted claims and resolved six disagreements through discussion, evaluator subjectivity and extraction omissions may affect the results. The automated structural analysis may likewise be affected by variations not captured by the regular expressions. **Construct Validity.** Verifiability is defined relative to the evidence exposed by our execution environment. Thus, a claim classified as non-verifiable may become verifiable under richer instrumentation, such as logging file accesses or tool invocations. Our results should therefore be interpreted as verifiability under the adopted instrumentation rather than as an intrinsic property of the claims. **External Validity.** The execution study covers three sub-agents, one task per agent, and one execution configuration. LLM-based agents are nondeterministic, and different tasks, prompts, interaction strategies, or execution settings may produce different behaviors. Therefore, the observed conformance should not be generalized beyond the evaluated scenarios.

5 Conclusion

This paper investigated the behavioral specifications of Claude Code sub-agents through the lens of Requirements Engineering by characterizing their specification structure, assessing the verifiability of their behavioral claims, and evaluating their conformance during execution. Manual analysis of three sub-agents (*prompt-engineer*, *rails-expert*, and *test-automator*) identified a common specification structure composed of nine sections. This structure was subsequently validated across 153 specifications from the Awesome Claude Code Sub-agents repository. From the specifications of the three selected sub-agents, 60 behavioral claims were extracted, of which only 11 (18.3%) were classified as verifiable according to the proposed operational criteria based on trigger, observable evidence, resolvable target, and falsifiability. The selected sub-agents were then executed, revealing that only one applicable behavioral claim was not satisfied during execution, while three claims were not applicable to the experimental scenarios, resulting in an overall conformance rate of 87%. These findings indicate that Claude Code sub-agents generally conform to the behavioral claims that can be objectively verified, but that the majority of their specifications describe behaviors that cannot be assessed objectively from execution artifacts. This suggests that improving the verifiability of specifications is an important step toward enabling systematic conformance assessment and automated evaluation of AI agents. **Future Work.** Evaluating multiple tasks and repeated executions per sub-agent would help characterize behavioral variability across different execution settings and interaction strategies. Richer instrumentation could also provide additional execution evidence and distinguish inherently non-verifiable claims from claims that are merely unobservable in the current environment. Finally, the claim extraction and structural analysis should be validated through alternative extraction methods and manual validation, and the study should be extended to more sub-agents and agentic coding platforms.

Artifact Availability

All data produced are available online [4]. All experiments were conducted on July 1, 2026.

Acknowledgements

This work was supported by Kunumi Institute. The authors thank the institution for its financial support and commitment to advancing scientific research.

References

[1] Anthropic. 2025. How We Built Our Multi-Agent Research System. https://www.anthropic.com/engineering/multi-agent-research-system. Anthropic Engineering. Acesso em: 6 jul. 2026.
[2] Victor R Basili1 Gianluigi Caldiera and H Dieter Rombach. 1994. The goal question metric approach. *Encyclopedia of software engineering* (1994), 528–532.
[3] Worawalan Chatlatanagulchai et al. 2025. On the Use of Agentic Coding Manifests: An Empirical Study of Claude Code. In *Product-Focused Software Process Improvement: 26th International Conference, PROFES 2025, Salerno, Italy, December 1–3, 2025, Proceedings* (Salerno, Italy). Springer-Verlag, Berlin, Heidelberg, 543–551. doi:10.1007/978-3-032-12089-2_40
[4] Matthias Galster et al. 2026. Harness Engineering for Agentic AI Coding Tools: An Exploratory Study. arXiv:2602.14690 [cs.SE] https://arxiv.org/abs/2602.14690
[5] Sirui Hong et al. 2024. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VtmBAGCN7o
[6] Xinyi Hou et al. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8, Article 220 (Dec. 2024), 79 pages. doi:10.1145/3695988
[7] Junwei Liu et al. 2026. Large Language Model-Based Agents for Software Engineering: A Survey. *ACM Trans. Softw. Eng. Methodol.* (March 2026). doi:10.1145/3796507 Just Accepted.
[8] Qianou Ma et al. 2025. What Should We Engineer in Prompts? Training Humans in Requirement-Driven LLM Use. *ACM Trans. Comput.-Hum. Interact.* 32, 4, Article 41 (Aug. 2025), 27 pages. doi:10.1145/3731756
[9] Seyedmoein Mohsenimofidi et al. 2026. Context Engineering for AI Agents in Open-Source Software. arXiv:2510.21413 [cs.SE] https://arxiv.org/abs/2510.21413
[10] Yunjia Qi et al. 2025. AGENTIF: Benchmarking Instruction Following of Large Language Models in Agentic Scenarios. arXiv:2505.16944 [cs.AI] https://arxiv.org/abs/2505.16944
[11] Karl Wiegers and Joy Beatty. 2013. *Software requirements.* Pearson Education.
[12] Zhihan Zhang et al. 2025. IHEval: Evaluating Language Models on Following the Instruction Hierarchy. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 8374–8398. doi:10.18653/v1/2025.naacl-long.425
[13] Jeffrey Zhou et al. 2023. Instruction-Following Evaluation for Large Language Models. arXiv:2311.07911 [cs.CL] https://arxiv.org/abs/2311.07911

[4] https://github.com/Agents4Good/claude-subagents-analysis