UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Hélio Victor Flexa dos Santos

Understanding the Configuration of AI Coding Agents: Concerns, Patterns, and Smells

Belo Horizonte
2026

Hélio Victor Flexa dos Santos

Understanding the Configuration of AI Coding Agents: Concerns, Patterns, and Smells

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Marco Túlio de Oliveira Valente
Co-Advisor: João Eduardo Montandon de Araujo Filho

Belo Horizonte
2026

2026, Hélio Victor Flexa dos Santos

Santos, Hélio Victor Flexa dos.

S237u Understanding the configuration of AI coding Agents: [recurso eletrônico] concerns, patterns, and smells / Hélio Victor Flexa dos Santos – 2026.
1 recurso online (73 f. il., color.) : pdf.

Orientador: Marco Túlio de Oliveira Valente.
Coorientador: João Eduardo Montandon de Araújo Filho.

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 68-73

1. Computação – Teses. 2. Engenharia de Software – Teses. 3. Agentes inteligentes (Software) – Teses. 4. Mineração de dados (Computação) – Teses. I. Valente, Marco Túlio de Oliveira II. Araújo Filho, João Eduardo Montandon de. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV. Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg Lucas Cruz - CRB 6/819 - Universidade Federal de Minas Gerais ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS

UNDERSTANDING THE CONFIGURATION OF AI CODING AGENTS: CONCERNS, PATTERNS, AND SMELLS

HELIO VICTOR FLEXA DOS SANTOS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

Prof. João Eduardo Montandon de Araújo Filho - Coorientador
Departamento de Ciência da Computação - UFMG

Prof. André Cavalcante Hora
Departamento de Ciência da Computação - UFMG

Profa. Lucas Francisco da Matta Vegi
Departamento de Informática - UFV

Belo Horizonte, 2 de outubro de 2026

Documento assinado eletronicamente por **Marco Tulio de Oliveira Valente**, **Professor do Magistério Superior**, em 05/10/2026, às 16:29, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Andre Cavalcante Hora**, **Professor do Magistério Superior**, em 06/10/2026, às 17:04, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **João Eduardo Montandon de Araujo Filho**, **Professor do Magistério Superior**, em 07/10/2026, às 12:54, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Lucas Francisco da Matta Vegi**, **Usuário Externo**, em 08/10/2026, às 08:30, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **5621858** e o código CRC **35AEF833**.

Referência: Processo nº 23072.260339/2026-62 SEI nº 5621858

Acknowledgments

A conclusão desta dissertação representa mais um grande passo da minha trajetória acadêmica, que se iniciou em 2019, sendo eu o primeiro da família a ingressar no ensino superior. Entretanto, o resultado disso não foi exclusivamente meu, mas contou com o apoio de outras pessoas. Em especial, agradeço:

A **Deus**, por nunca ter me deixado perder a fé e a esperança, estando comigo nos momentos bons e ruins, inclusive naqueles em que eu mesmo já estava me perdendo.

A **mim**, por ter enfrentado todos os desafios da vida e vencido, mesmo quando achava que tudo estava ruindo.

À minha mãe, **Maria do Carmo Santos**, meu exemplo de vida, que sempre lutou para me criar sozinha e acreditou intensamente em todos os meus objetivos e conquistas por meio da educação.

Aos meus **orientadores**, Prof. Dr. Marco Túlio e Prof. Dr. João Eduardo, por me guiarem intensamente nesta jornada. A atenção e o cuidado durante todo o processo foram decisivos para o resultado desta dissertação.

Aos **amigos e familiares**, pelo apoio incondicional durante essa trajetória, pela valorização das minhas conquistas.

Aos **membros da banca**, Prof. Dr. André Hora e Prof. Dr. Lucas Vegi, pelo reconhecimento e pela valorização do meu trabalho.

Ao **PPGCC**, pelo suporte e pelo apoio financeiro e educacional, que me possibilitaram novas vivências. Em especial, agradeço à secretária Sônia Borges e aos coordenadores do programa.

Ao **DCC/UFMG**, por todo o apoio durante esses anos, pelo suporte para a realização das minhas atividades e pela oportunidade de me tornar o profissional e acadêmico que sou hoje.

"*Somos quem podemos ser*"
(Engenheiros do Hawaii)

Resumo

Grandes Modelos de Linguagem (LLMs) deslocaram o desenvolvimento de software dos assistentes de código interativos para agentes de codificação autônomos, capazes de planejar, invocar ferramentas e modificar repositórios. Para delimitar essa autonomia, desenvolvedores escrevem arquivos de configuração, como `AGENTS.md` e `CLAUDE.md`, carregados automaticamente no contexto do agente a cada sessão e que declaram convenções do projeto, restrições arquiteturais e comandos de construção e teste. Diferentemente de *Dockerfiles* ou *pipelines* de CI/CD, esses arquivos são escritos em linguagem natural e consumidos por LLMs. Portanto, podem ser redundantes, obsoletos ou contraditórios e, ainda assim, permanecer válidos para todas as ferramentas do processo de desenvolvimento. Nesta dissertação, investigamos esses artefatos por meio de dois estudos empíricos. No primeiro, analisamos 328 arquivos `CLAUDE.md` dos projetos Claude Code mais populares do GitHub. Constatamos que preocupações arquiteturais predominam, presentes em 72,6% dos arquivos, que conteúdo não textual como links e diagramas é escasso, e que um pequeno conjunto de padrões recorrentes explica como esses arquivos são organizados, com preocupações sobre arquitetura aparecendo entre os cinco mais frequentes. No segundo estudo, dedicado à qualidade desses arquivos, introduzimos, a partir de uma revisão de literatura cinzenta, o primeiro catálogo de *configuration smells* para agentes de codificação, composto por seis problemas recorrentes: *Context Bloat*, *Skill Leakage*, *Lint Leakage*, *Blind Reference*, *Init Fossilization* e *Conflicting Instructions*. A análise de discussões em *pull requests* não revelou problemas além desses, reforçando a abrangência do catálogo. Propomos heurísticas automatizadas de detecção e as avaliamos em 100 repositórios populares de código aberto, constatando que 91 dos 100 arquivos contêm ao menos um *smell*. A mineração de regras de associação revela que os *smells* co-ocorrem com frequência, indicando que arquivos longos frequentemente contêm instruções inconsistentes ou excessivamente específicas. Como os arquivos de configuração são artefatos centrais no desenvolvimento de software agêntico, nossos resultados sugerem que sua qualidade merece esforço e atenção. Esperamos que o catálogo proposto sirva de base para futuras ferramentas e técnicas voltadas a detectar e prevenir *configuration smells* em ecossistemas de agentes de codificação.

Palavras-chave: engenharia de software; agentes de codificação; arquivos de configuração; mineração de repositórios de software.

Abstract

Large Language Models (LLMs) have shifted software development from interactive code assistants toward autonomous coding agents that plan, invoke tools, and modify repositories. To constrain this autonomy, developers write configuration files, such as `AGENTS.md` and `CLAUDE.md`, which are automatically loaded into the agent's context at every session and declare project conventions, architectural constraints, and build and test commands. Unlike *Dockerfiles* or CI/CD pipelines, these files are written in natural language and consumed by LLMs: they can be redundant, obsolete, or contradictory and still remain valid for every tool in the development process. In this dissertation, we investigate these artifacts through two empirical studies. In the first, we analyze 328 `CLAUDE.md` files from the most popular Claude Code projects on GitHub. We find that architectural guidance predominates, appearing in 72.6% of the files, that non-textual content such as links and diagrams is scarce, and that a small set of recurring patterns explains how these files are organized, with concerns about architecture appearing among the five most frequent ones. In the second study, devoted to the quality of these files, we draw on a grey literature review to introduce the first catalog of *configuration smells* for coding agents, comprising six recurring problems: *Context Bloat*, *Skill Leakage*, *Lint Leakage*, *Blind References*, *Init Fossilization*, and *Conflicting Instructions*. An analysis of *pull request* discussions revealed no problems beyond these, reinforcing the comprehensiveness of the catalog. We propose automated detection heuristics and evaluate them on 100 popular open-source repositories, finding that 91 out of 100 files contain at least one *smell*. Association rule mining reveals that smells frequently co-occur, indicating that long files often contain inconsistent or overly specific instructions. Since configuration files are key artifacts in agentic software development, our findings suggest that their quality deserves effort and attention. We expect the proposed catalog to serve as a foundation for future tools and techniques to detect and prevent configuration smells in coding-agent ecosystems.

Keywords: software engineering; coding agents; configuration files; mining software repositories.

List of Figures

2.1 Example of Few-shot Prompting for Code Conversion. 19
2.2 Example of Chain-of-Thought (CoT) for Debugging. 20
2.3 Example of a System Prompt for Code Review. 20
2.4 Example of a Tool Schema and Function Call. 23
2.5 Example of an `AGENTS.md` configuration file. 24

3.1 Concerns and practices addressed in agent config. files 35
3.2 Architecture guidelines (evstack/ev-node). 36
3.3 Development guidelines (PrefectHQ/marvin). 37
3.4 Project overview (callstack/react-native-testing-library). 37
3.5 Example of testing guidelines (lightdash/lightdash). 38
3.6 Config. file with code example (run-llama/LlamaIndexTS). 39
3.7 Config. file with a link to a project's page (grafana/mimir). 39

4.1 Overview of the grey literature steps 43
4.2 Search string used in the literature review. 43
4.3 Prompt to detect Skill Leakage, Lint Leakage, Blind References, and Conflicting Instructions. 50
4.4 Example added to the prompt to detect Blind References 50
4.5 Example of Skill Leakage (quickemu-project/quickemu). 53
4.6 Example of Skill Leakage (promptfoo/promptfoo). 54
4.7 Example of Skill Leakage (TryGhost/Ghost). 54
4.8 Example of Lint Leakage (google/adk-python) 55
4.9 External reference described with context (browser-use/browser-use) 56
4.10 Blind Reference (SuperClaude-Org/SuperClaude_Framework) 56
4.11 Number of changes in `AGENTS.md` files (*Init Fossilization* corresponds to files with a single commit) 57
4.12 Number of commits in projects exhibiting *Init Fossilization* (counting only commits made after the creation of `AGENTS.md`). 58
4.13 Example of Conflicting Instructions (inkline/inkline) 58
4.14 Smell accumulation in `AGENTS.md` files. 60

List of Tables

3.1 Size of Claude.md files (measured in number of ## sections). 34
3.2 Usage of Code Example and Links 38
3.3 Top-5 Most Popular `Claude.md` Patterns. 40

4.1 Configuration Smells 46
4.2 Smells detected in real projects (FP: False Positives; Prec: Precision) 51
4.3 Most common leaked skills 53
4.4 Co-existing smells detected by Apriori. 59

Contents

1 Introduction **13**

1.1 Motivation . . . 13
1.2 Proposed Work . . . 14
1.3 Contributions . . . 15
1.4 Publications . . . 16
1.5 Outline of the Dissertation . . . 17

2 Background and Related Work **18**

2.1 Large Language Models . . . 18
2.1.1 Prompt Techniques . . . 19
2.1.2 Context Engineering . . . 21
2.2 LLM-based Software Agents . . . 21
2.3 Agent Configuration Files . . . 23
2.4 Related Work . . . 25
2.4.1 LLMs in Software Engineering . . . 26
2.4.2 Autonomous Coding Agents . . . 27
2.4.3 Bad Smells in Source Code and Configuration Artifacts . . . 29
2.5 Final Remarks . . . 30

3 Decoding the Configuration of AI Coding Agents **31**

3.1 Research Questions . . . 31
3.2 Study Design . . . 32
3.2.1 Data Collection . . . 32
3.2.2 Manual Analysis . . . 33
3.2.3 Pattern Analysis . . . 34
3.3 Most Common Concerns and Practices (RQ1) . . . 35
3.4 Usage of Code Examples and Links (RQ2) . . . 38
3.5 Patterns of Configuration Files (RQ3) . . . 39
3.6 Threats to Validity . . . 40
3.7 Final Remarks . . . 41

4 Configuration Smells in AGENTS.md Files **42**

4.1 Methods . . . 42
4.1.1 Literature Review . . . 42

4.1.2 Dataset of Agent Configuration Files . . . 44
4.1.3 Pull Requests Analysis . . . 45
4.2 Configuration Smells . . . 46
4.2.1 Context Bloat . . . 46
4.2.2 Skill Leakage . . . 47
4.2.3 Lint Leakage . . . 47
4.2.4 Blind References . . . 48
4.2.5 Init Fossilization . . . 48
4.2.6 Conflicting Instructions . . . 48
4.3 Detection Heuristics . . . 49
4.3.1 Heuristic based on Lines of Code . . . 49
4.3.2 Heuristic based on Language Models . . . 49
4.3.3 Heuristic based on Number of Commits . . . 51
4.4 Configuration Smells in the Wild . . . 51
4.4.1 Context Bloat . . . 52
4.4.2 Skill Leakage . . . 52
4.4.3 Lint Leakage . . . 55
4.4.4 Blind Reference . . . 55
4.4.5 Init Fossilization . . . 56
4.4.6 Conflicting Instructions . . . 57
4.5 Co-Occurrence Analysis . . . 59
4.6 Threats to Validity . . . 61
4.7 Final Remarks . . . 62

5 Conclusion **63**
5.1 Overview . . . 63
5.2 Summary of Findings . . . 63
5.2.1 Anatomy of Agent Configuration Files . . . 64
5.2.2 Configuration Smells . . . 64
5.3 Guidelines for Writing Agent Configuration Files . . . 65
5.4 Future Work . . . 66

References **68**

Chapter 1

Introduction

1.1 Motivation

The recent advancement of Large Language Models (LLMs) has structurally transformed Software Engineering practices, driving a shift from simple interactive code assistants (such as code-completion plugins and chatbots) to autonomous coding agents [15, 17]. Contemporary agentic tools, such as Claude Code, OpenAI Codex, and Cursor Agent, operate within continuous executive control loops (*agentic loops*), actively interacting with operating systems, command-line interfaces, automated test suites, and version-control repositories to resolve repository-level engineering tasks [53, 17].

To guide and delimit the autonomous behavior of these agents, the software industry and the open-source community have established the use of repository-level configuration files, commonly known as `AGENTS.md` or `CLAUDE.md` [6]. Automatically loaded into the privileged prompt scope upon the initialization of each work session, these artifacts function as declarative context contracts between the development team and the autonomous tool [23]. Their central purpose is to provide persistent instructions regarding system architecture, design patterns, styling conventions, testing procedures, and contribution workflows, ensuring that the agent remains aligned with developer expectations [6].

The adoption of these files has been rapid, and so has the delegation of real engineering work to the agents they configure [32]. This raises a practical question: what happens when the guidance they provide is incomplete, outdated, or contradictory? Recent evidence suggests the cost is concrete. In an empirical study of 567 pull requests authored by Claude Code on GitHub, Watanabe et al. [49] found that a recurring cause of rejection by human maintainers was not broken syntax or failing tests, but subtle violations of unwritten team conventions, local architectural choices, and testing practices. In other words, the agents produced code that was technically correct and contextually wrong. Since project conventions are precisely what configuration files are meant to encode, the quality of these artifacts is not a cosmetic concern: it directly affects whether agent-generated contributions are accepted into a codebase.

The nature of these failures is best appreciated through a concrete case. In one of the projects analyzed in this dissertation, the configuration file instructs the agent, in one section, to place new components in a given directory, and, in a later section, to create them in a different one. The two rules cannot both be satisfied. No compiler will reject this file, no linter will flag it, and no test will fail because of it; the agent simply follows one instruction or the other, and may choose differently across sessions. This illustrates the central difficulty of the artifact: written in natural language and consumed by an LLM, a configuration file can be internally inconsistent, redundant, or obsolete while remaining, from the perspective of every tool in the development pipeline, perfectly valid.

Research on agentic coding has so far concentrated on the capabilities of the underlying models and on the design of the harnesses that wrap them—benchmarks for issue resolution [18], agent-computer interfaces [53], and retrieval strategies that determine which code the model sees [52]—as well as on prompts understood as transient, task-level inputs [31, 36]. The persistent, project-level artifacts through which maintainers actually steer these agents have received considerably less attention.

Despite their growing prominence in the agentic ecosystem, these artifacts remain under-investigated in scientific literature. Because they are a recent practice, there is a distinct lack of empirical studies characterizing the internal structure, recurrent sections, and content patterns authored by developers in these files, as well as investigations assessing the *configuration smells* that affect them. Unlike traditional infrastructure files (such as *Dockerfiles* or CI/CD pipelines), where syntactic errors are immediately flagged by deterministic compilers and interpreters, agent configuration files are consumed by probabilistic inference engines [17]. A contradictory rule therefore raises no error; it silently biases the agent toward one behavior or another. The lack of standardization, verbosity, and the presence of conflicting or redundant instructions can thus degrade an agent's ability to ground essential project rules while unnecessarily inflating operational inference costs, since these files are re-injected into the model's context at the start of every session and billed per input token [17].

1.2 Proposed Work

In this dissertation, we propose a comprehensive empirical investigation into the configuration of coding agents in open-source projects. The primary objective is to systematically understand how developers structure these configuration files and identify the *configuration smells* that impair their maintainability and operational effectiveness.

To achieve these goals, the research is organized into two complementary empirical

studies:

1. **Study 1 — Anatomy and Characterization of Agent Configurations:** An empirical analysis aimed at decoding the structure and content of configuration files (such as `CLAUDE.md`) across real-world GitHub projects. This study investigates what information maintainers consider essential to convey to agents, mapping recurrent sections (such as architecture overviews, build/test commands, and development guidelines). The study also measures the presence of non-textual elements, such as code examples, links, and diagrams, and mines maximal co-occurrence patterns among the identified concerns in order to characterize the most common patterns of configuration files.
2. **Study 2 — Catalog and Detection of Configuration Smells:** An in-depth investigation into recurrent misconfiguration issues within `AGENTS.md` and `CLAUDE.md` files. Grounded in a grey literature review, this study formalizes the first catalog of *configuration smells* for agents, develops automated heuristics for their detection, and empirically analyzes their prevalence and co-occurrence patterns across the open-source ecosystem.

Together, these studies produce a coherent picture. Developers overwhelmingly use configuration files to communicate architectural constraints, development guidelines, and testing procedures, and they organize this content around a small number of recurring patterns in which concerns about architecture are almost always present. At the same time, the quality of these files is uneven: 91% of the files we analyzed exhibit at least one *configuration smell*, the most frequent being the duplication of rules that deterministic tools already enforce. These smells are not independent of one another: long files often contain inconsistent or overly specific instructions. All data, scripts, and annotations used in both studies are available in public replication packages.

1.3 Contributions

The main contributions of this dissertation to the field of Software Engineering are:

- **Structural Characterization of Agent Configuration Files:** We provide the first systematic mapping of the structure, sections, and instruction categories present in agent configuration files across real-world open-source projects, identifying the most prevalent content patterns adopted by developers.

- **Analysis of Structural Richness and Recurring Patterns:** We measure the extent to which developers complement textual rules with non-textual elements—code examples, internal and external links, and diagrams—and we identify the maximal sets of concerns that most frequently co-occur in the same file, providing reference patterns for practitioners.
- **Formalization of a Configuration Smells Catalog:** We introduce an original taxonomy comprising six *configuration smells* in agent configuration files (*Context Bloat*, *Skill Leakage*, *Lint Leakage*, *Blind References*, *Init Fossilization*, and *Conflicting Instructions*), conceptualizing their causes and degradation risks.
- **Detection Heuristics and Empirical Evaluation:** We design and evaluate automated approaches to identify these smells, demonstrating that 91% of analyzed configuration files contain at least one occurrence, while mapping statistical association rules of co-occurrence among them.
- **Guidelines for Writing Agent Configuration Files:** Grounded in the findings from both studies, we derive a set of practical guidelines to assist maintainers in writing concise, consistent, and maintainable configuration files for coding agents.

1.4 Publications

The research conducted throughout this dissertation resulted in the following scientific publications:

- **Hélio Victor Flexa dos Santos**, Vitor Costa, Joao Eduardo Montandon, and Marco Tulio Valente. Decoding the Configuration of AI Coding Agents: Insights from Claude Code Projects. *1st International Workshop on Agentic Engineering (AGENT), co-located with the 48th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 63–67, 2026. https://doi.org/10.1145/3786167.3788412
- **Hélio Victor Flexa dos Santos**, Vitor Costa, Joao Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. Configuration Smells in AGENTS.md Files: Common Mistakes in Configuring Coding Agents. *26th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–11, 2026.

1.5 Outline of the Dissertation

The remainder of this dissertation is organized as follows:

- **Chapter 2** presents the theoretical background on Large Language Models, on the architecture of LLM-based software agents, and on agent configuration files themselves, alongside a review of related work covering the adoption of LLMs in software engineering, autonomous coding agents, and *bad smells* in source code and configuration artifacts.
- **Chapter 3** details the first empirical study, decoding the configuration of AI coding agents through the structural characterization, content taxonomy, and recurring patterns of configuration files in real-world projects.
- **Chapter 4** presents the second empirical study, introducing the catalog of configuration smells in agent configuration files, the automated detection heuristics, their prevalence across open-source repositories, and their co-occurrence patterns.
- **Chapter 5** concludes the dissertation, summarizing the findings of both studies, deriving practical guidelines for writing and maintaining agent configuration files, and outlining directions for future research.

Chapter 2

Background and Related Work

This chapter presents the theoretical foundation and the literature context of this dissertation. Section 2.1 introduces Large Language Models, covering their probabilistic behavior, prompt techniques, and context engineering. Section 2.2 describes the architecture of LLM-based software agents, including control harnesses, tool use, and memory mechanisms. Section 2.3 presents agent configuration files, the central artifact investigated in this dissertation, detailing their format, their loading mechanism, and the tools that adopt them. Section 2.4 reviews the related literature, organized into LLMs in software engineering, autonomous coding agents, and bad smells in source code and configuration artifacts. Finally, Section 2.5 presents the final remarks.

2.1 Large Language Models

The recent advancement of Generative Artificial Intelligence has driven a structural transformation in software engineering practices, a scenario frequently conceptualized in the literature as "Software Engineering 3.0" [15]. At the center of this transformation are Large Language Models (LLMs), which developers invoke to analyze legacy codebases, propose bug fixes, synthesize unit tests, and translate textual requirements into executable implementations.

Unlike traditional software modules, which rely on deterministic logic, LLMs are probabilistic [17]. Due to their stochastic nature, identical inputs can produce variations in generated code across runs, unless the temperature parameter is set to zero, and even then minor infrastructure-level discrepancies can occur. Furthermore, models are prone to hallucinations, such as referencing non-existent third-party libraries, invoking deprecated methods, or inventing file paths. As emphasized by Huyen [17], systems built on LLMs cannot blindly trust model outputs; they require guardrails, deterministic syntactic validators such as linters and compilers, and automated test suites to verify every generated artifact prior to merging it into the repository.

2.1.1 Prompt Techniques

The primary interface for communicating with an LLM is through natural language instructions, known as prompts. From a software engineering perspective, designing these inputs guides code generation and automated analyses without requiring modifications to the underlying model architecture. Prompts can be written by the user, who states the task at hand, or by the system that hosts the model, which establishes the boundaries within which the user prompts are interpreted. We first describe three techniques used in user prompts, and then discuss system prompts.

The first technique concerns the amount of demonstration provided with the instruction. In the *zero-shot* paradigm, the task instruction is provided directly to the model, relying entirely on the prior knowledge acquired during pre-training. Conversely, *few-shot prompting*, also known as in-context learning [5], relies on providing explicit demonstrations containing expected input-output pairs. Figure 2.1 illustrates this technique in a code conversion task: the two demonstration pairs establish, without any explicit rule, that parameters and return values must be annotated with strict TypeScript types. This approach embeds formatting constraints, typing rules, and implementation styles directly within the prompt, serving as a flexible and lightweight alternative to traditional fine-tuning.

Figure 2.1: Example of Few-shot Prompting for Code Conversion.

```
Instruction: Convert the function declaration to strict TypeScript.

Input: function sum(a, b) { return a + b; }
Output: function sum(a: number, b: number): number { return a + b; }

Input: function greet(name) { return "Hello " + name; }
Output: function greet(name: string): string { return `Hello ${name}`; }

Input: function isActive(status) { return status === 1; }
Output:
```

Source: Adapted from Huyen [17].

The second technique concerns the reasoning process itself. *Chain-of-Thought* (CoT) prompting, originally introduced by Wei et al. [50], encourages the model to generate intermediate reasoning steps before delivering the final solution. In analytical software tasks, such as tracing concurrency issues, detecting race conditions, or resolving circular dependencies, step-by-step decomposition reduces hallucinations and improves the logical consistency of the generated output. Figure 2.2 shows this decomposition applied to dead-

lock detection: rather than emitting a verdict directly, the model first isolates the lock acquisition order of each thread, then checks for order inversion, and only then concludes.

Figure 2.2: Example of Chain-of-Thought (CoT) for Debugging.

```
Instruction: Analyze the code snippet below and determine if there is a
deadlock risk. Think step-by-step before stating your conclusion.

Code:
Thread 1: lock(A); lock(B); ... unlock(B); unlock(A);
Thread 2: lock(B); lock(A); ... unlock(A); unlock(B);

Expected Response:
Step 1: Analyze the lock acquisition order in Thread 1 (A -> B).
Step 2: Analyze the lock acquisition order in Thread 2 (B -> A).
Step 3: Check for lock order inversion (Yes, A and B are interleaved).
Step 4: Evaluate the scenario where Thread 1 acquires A and Thread 2 acquires B.
Conclusion: There is an imminent deadlock risk due to the inconsistent
lock acquisition order across threads.
```

Source: Adapted from Huyen [17].

The third technique concerns the scope in which instructions are injected, and moves from user prompts to system prompts. *System prompts* consist of governance guidelines placed in the initial scope of the inference session. This layer defines the role assumed by the model, its level of technical rigor, compliance constraints, and operational boundaries that must be preserved throughout the conversation. Figure 2.3 presents a system prompt that assigns the model the role of a security reviewer. Two elements deserve attention: the explicit prohibition against rewriting business logic, which constrains the model's scope of action, and the mandatory JSON output format, which is what makes the response consumable by an automated pipeline rather than by a human reader.

Figure 2.3: Example of a System Prompt for Code Review.

```
[SYSTEM PROMPT]
You are a senior software engineer acting as an Application Security
(AppSec) code reviewer.
Strict guidelines:
1. Identify potential vulnerabilities according to the OWASP Top 10.
2. Do not rewrite existing business logic; only point out flaws.
3. If the code is secure, respond strictly with: "STATUS: APPROVED".
4. Always return violations in JSON format with keys:
   "line", "vulnerability", and "recommendation".
```

Source: Adapted from Huyen [17].

2.1.2 Context Engineering

Building upon these techniques, Context Engineering has emerged as a distinct discipline [23]. Rather than focusing solely on phrasing individual instructions, context engineering encompasses the systematic, dynamic orchestration of all informational artifacts delivered to the model, integrating codebase slices, data schemas, execution logs, and persistent configuration files. The decision of *how much* context to inject, however, is not free. Despite the continuous expansion of context window capacities in commercial models, indiscriminately packing information into the context introduces severe system engineering trade-offs [17]:

1. **"Lost in the Middle" Phenomenon:** Language models tend to pay closer attention to instructions located at the beginning and end of the prompt, suffering attention degradation when attempting to retrieve constraints or critical details placed in the middle of long contexts.
2. **Financial Cost and Tokenomics:** LLM API requests are billed per processed token, for both input and output. Systematically injecting redundant files or unnecessary rules on every interaction significantly escalates ongoing operational costs.
3. **Inference Latency:** Time to First Token (TTFT) and overall response time scale with the volume of input tokens provided, degrading the developer experience and slowing down automated loops.

2.2 LLM-based Software Agents

The transition from localized coding assistants, such as code completion plugins or question-answering chatbots, to autonomous software agents represents a shift from passive text generation to active automation of end-to-end engineering tasks [17, 51]. Architecturally, a modern coding agent can be decomposed into two core building blocks:

1. **The Base Language Model:** Acts as the engine for reasoning, logical planning, and semantic comprehension.
2. **The Harness (Control Environment):** Functions as the software infrastructure that wraps the model inside a continuous execution loop, the *agentic loop*, grant-

ing operating system access, managing session state, and dispatching the actions requested by the model.

The most widely adopted control pattern in software agents is the *ReAct* (Reasoning + Acting) framework [54]. In this paradigm, the agent iteratively executes three main phases:

- **Thought:** The model assesses the high-level goal (e.g., "fix the authentication failure in the `/login` endpoint") and its recent execution history to formulate the next logical step.
- **Action:** The model decides to invoke a specific external tool (e.g., executing a regex search across the codebase or reading a specific file).
- **Observation:** The harness executes the tool in the actual environment, captures the output (file contents, terminal logs, or test execution results), and feeds it back into the model's context window, triggering the next iteration of the loop.

The capability that gives an LLM agency is support for external tools, commonly referred to as *Tool Use* [17]. The developer or framework provides the model with a structured schema, typically expressed in JSON Schema, defining the available tools, their expected parameters, and descriptions of their behavior. During inference, when the model determines that an action is required, it pauses natural language generation and emits a structured function call. Figure 2.4 shows the schema declared in the system prompt, which describes the `run_test_suite` tool and its parameters, and the call subsequently emitted by the model, which binds concrete values to those parameters.

The standard tool inventory commonly includes code search and navigation utilities (e.g., *grep*, *ripgrep*, or AST-based structural search); file manipulation utilities for reading, creating, and applying patch edits; command-line terminal execution, enabling linters, build systems, and test runners to be invoked; and version-control interfaces (Git) and platform APIs (GitHub, GitLab) to create branches, commit code, and submit pull requests.

Executing complex software engineering tasks also requires agents to maintain consistency across multiple steps over time. Memory in agentic systems is divided into two primary tiers [17]. *Short-term* or session memory retains the sequential record of thoughts, tool calls, and observations of the active session; because this memory is bounded by the context window, summarization and log-pruning strategies are commonly employed by the harness to prevent context exhaustion. *Long-term memory*, in turn, is typically realized through Retrieval-Augmented Generation (RAG) mechanisms, which allow the agent to query external knowledge bases, technical documentation, and repository history

Figure 2.4: Example of a Tool Schema and Function Call.

```
// 1. JSON Schema provided in the System Prompt
{
  "name": "run_test_suite",
  "description": "Executes unit tests at the specified path",
  "parameters": {
    "type": "object",
    "properties": {
      "test_path": { "type": "string", "description": "Test file path" },
      "coverage": { "type": "boolean", "description": "Generate coverage" }
    },
    "required": ["test_path"]
  }
}

// 2. Call emitted by the LLM (Action)
{
  "tool": "run_test_suite",
  "arguments": { "test_path": "tests/test_auth.py", "coverage": true }
}
```

Source: Adapted from Huyen [17].

on demand; the codebase is indexed via vector or hybrid search, so that the agent retrieves only the information relevant to the current task.

Although RAG and tool inventories facilitate project navigation, agents still operate blindly with respect to implicit team conventions, preferred workflows, and project-specific architectural constraints, unless these are explicitly stated. This gap is precisely what agent configuration files are designed to fill, as discussed next.

2.3 Agent Configuration Files

Agent configuration files, such as `AGENTS.md` and `CLAUDE.md`, have emerged as a standard mechanism for context injection and for providing persistent memory to coding agents. According to Xi et al. [51], the effectiveness of an agent lies not only in the power of the base model, but in the precision with which guidelines are defined, stored, and retrieved. These files therefore act as a declarative context contract between the development team and the autonomous tool.

In essence, an `AGENTS.md` file is a markdown document containing rules about the

project. In it, developers can record information regarding system architecture, tool documentation, testing practices, build and execution commands, code style conventions, and other constraints whose absence might lead to agent errors. Figure 2.5 presents a minimal example.[1] As shown, the file declares the commands required to set up and exercise the project, the code style rules adopted by the team, and workflow guidelines, such as the recommendation to run individual tests rather than the entire suite for performance reasons.

Figure 2.5: Example of an `AGENTS.md` configuration file.

```
# AGENTS.md

## Setup commands
- Install deps: `pnpm install`
- Start dev server: `pnpm dev`
- Run tests: `pnpm test`

## Code style
- TypeScript strict mode
- Use functional patterns where possible
- Use ES modules (import/export) syntax, not CommonJS (require)

## 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
```

Source: Adapted from https://agents.md.

Upon starting a session, the agent detects the presence of configuration files in the project directory and incorporates their content into its prompt, persisting this context throughout the entire agentic loop [23].

The first version of a configuration file can be created automatically by the agent itself. In this case, a specific command, generally `/init`, relies on an internal prompt instructing the agent to inspect the repository and derive an initial set of guidelines. For example, the prompt used by OpenAI Codex, which is publicly available,[2] asks the agent to summarize the project structure and module organization, build and testing commands, coding and naming conventions, testing guidelines, and pull request requirements, while keeping the resulting document concise, instructional, and tailored to the analyzed repository. This automatic bootstrapping is convenient, but it also means that a project may adopt a configuration file that no human has ever reviewed.

Among the tools that adopt this mechanism, Claude Code deserves particular attention, as it is the subject of the first empirical study of this dissertation. Claude

[1] This example was extracted and adapted from https://agents.md.

[2] https://github.com/openai/codex/blob/main/codex-rs/tui/prompt_for_init_command.md

Code is a command line tool for agentic coding created by Anthropic and designed to assist developers in their coding workflows.[3] 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. To make best use of Claude Code, developers provide project-specific instructions in a configuration file named `CLAUDE.md`, also written in markdown. For each request, Claude Code first reads the content of this file looking for information that can help it fulfill the request.

It is also possible to use both files in the same repository. In this case, one file simply points to the other; for example, the `CLAUDE.md` file may exist but contain only a single line instructing the agent to read `AGENTS.md`. A related practice, which we observe in Chapter 3, is the use of configuration files as *memory banks*, that is, files whose content consists exclusively of pointers to other markdown documents within the project.

Note: Most agent-based systems use the name `AGENTS.md` for their configuration files. The notable exception is Claude Code, which uses the name `CLAUDE.md`. Therefore, throughout this dissertation, we analyze both `AGENTS.md` and `CLAUDE.md` files, since the only difference between them, in terms of purpose and role within an agent-based system, is their name.

Finally, it is worth stressing that these files are authored and maintained manually by human developers, in an artifact that has no compiler, no type checker, and no test suite to validate it. As a consequence, they are susceptible to quality degradation, obsolete instructions, redundancies, and internal contradictions.

2.4 Related Work

LLMs are transforming software engineering practices, playing a central role in increasing developer productivity and automating cognitive tasks [15, 10]. As developers increasingly rely on LLMs and autonomous agents for software development tasks [53, 19, 25], investigating the artifacts, context-injection strategies, and configuration quality that govern these tools becomes essential. We organize the related literature into three groups: LLMs in software engineering (Section 2.4.1), autonomous coding agents (Section 2.4.2), and bad smells in source code and configuration artifacts (Section 2.4.3). In each group, we conclude by positioning the two empirical studies of this dissertation with respect to the reviewed works.

[3] https://www.anthropic.com/news/claude-3-7-sonnet

2.4.1 LLMs in Software Engineering

The application of LLMs to software engineering encompasses a wide spectrum of tasks, including code generation [7, 37], bug fixing [22], test synthesis [38, 3], code review [21], documentation writing [13, 16], and code comprehension [27, 30].

Early empirical investigations primarily examined conversational LLM interfaces interacting with isolated snippets, methods, or classes. Tufano et al. [43] conducted a mining-based study of how ChatGPT is used in open-source projects, analyzing shared conversation links referenced in commits, issues, and pull requests, and characterizing the developer intentions behind them. The study shows that developers resort to conversational assistants for a broad range of purposes beyond code generation, including explanation and design discussion, but also that a substantial portion of these interactions is not reflected in the final merged artifact. In a complementary direction, Watanabe et al. [48] investigated the acceptance of ChatGPT-generated code reviews by developers, finding that automatically produced review comments are frequently perceived as generic and disconnected from project-specific concerns. Both results point to the same underlying limitation: without access to the conventions and constraints of the target project, an LLM produces output that is technically plausible but contextually misaligned. This is precisely the gap that configuration files attempt to close, and it motivates our interest in what developers actually record in them (Chapter 3).

Another line of work investigated how the formulation of the prompt itself affects outcomes. Pister et al. [31] constructed a dataset of prompts extracted from open-source Python projects, showing that prompts are increasingly treated as first-class artifacts embedded in source code repositories rather than as ephemeral inputs. Sasaki et al. [36] systematized this space into a taxonomy of prompt engineering patterns for software engineering tasks. More specifically related to quality analysis, Silva et al. [39] evaluated the ability of ChatGPT to detect traditional code smells, reporting that LLM-based detection is viable but sensitive to prompt formulation and prone to false positives. This last finding is methodologically relevant to Chapter 4, where we also employ LLM-based heuristics for smell detection and, for that reason, manually validate every reported instance and report precision explicitly.

However, evaluating LLMs on localized tasks revealed clear limitations when applied to real-world development. Through the SWE-bench benchmark, Jimenez et al. [18] assembled a set of 2,294 task instances derived from real issues and their corresponding pull requests in popular Python repositories, evaluating candidate patches by executing the project's own test suite. Their results demonstrated that resolving real-world GitHub issues requires multi-file navigation, contextual reasoning across dependencies, and iterative test execution, a setting in which models that perform well on isolated function

synthesis achieve markedly lower success rates. Consequently, the research focus shifted toward repository-level context engineering.

The effectiveness of foundation models in software engineering thus depends heavily on the quality and volume of the context provided. Naively concatenating entire source files into the prompt leads to severe economic and computational bottlenecks, a domain known as tokenomics. Excessive input tokens inflate API inference costs, introduce high latency, and trigger the attention degradation patterns discussed in Section 2.1 [17]. To address these constraints, Xia et al. [52] proposed Agentless, an approach that deliberately avoids autonomous agent loops in favor of a fixed three-phase pipeline of hierarchical fault localization, patch generation, and patch validation. Their central argument is that much of the cost and unpredictability of agentic approaches stems from letting the model decide which context to retrieve, and that a deterministic retrieval procedure can achieve competitive results at a fraction of the token budget. Mohsenimofidi et al. [23] further characterize context engineering as an emerging concern in open-source projects, documenting the artifacts that maintainers create specifically to supply agents with project context. Configuration files are one such artifact, and they complement retrieval pipelines by supplying pre-computed architectural constraints and project conventions that would be expensive, or impossible, to recover through search alone.

2.4.2 Autonomous Coding Agents

As the field transitioned from passive conversational assistants to autonomous agents, LLMs were embedded within execution harnesses capable of planning, invoking tools, and evaluating environment feedback in iterative loops. Yang et al. [53] introduced SWE-agent and, with it, the notion of an agent-computer interface: the observation that agent performance depends not only on the underlying model but on the design of the commands and feedback formats through which the model perceives and manipulates the repository. Poorly designed interfaces, they show, cause the model to waste turns recovering from malformed operations. This finding is conceptually adjacent to our own premise, with one difference in scope: SWE-agent concerns the interface designed by the *tool builder*, whereas configuration files constitute an interface authored by the *project maintainer*.

The adoption of these tools in practice has itself become an object of study. Robbes et al. [32] examined the diffusion of coding agents across GitHub, characterizing which projects adopt them and how agent-authored contributions are distributed across the ecosystem. Almeida et al. [2] evaluated Copilot agent mode on a concrete, well-delimited

maintenance task, library migration, providing a quantitative assessment of where autonomous agents succeed and where human intervention remains necessary.

The deployment of autonomous agents has also introduced new socio-technical challenges in collaborative development. Watanabe et al. [49] conducted an empirical study on 567 pull requests generated by Claude Code on GitHub, examining their acceptance rates and the reasons behind rejections. Their analysis reveals that a lack of project-specific alignment is a recurring cause of rejection by human maintainers. Crucially, these rejections stem not from broken syntax or failing tests, but from subtle violations of unwritten team conventions, local architectural choices, or testing practices. This result provides the strongest empirical motivation for the present dissertation: it demonstrates that the information typically recorded in configuration files, when absent or inaccurate, has measurable consequences on the acceptance of agent-generated contributions.

Closest to our work, Chatlatanagulchai et al. [6] conducted an empirical study of agentic coding manifests, motivated by the observation that developers lack comprehensive guidance on how to author these files. The authors analyzed 253 `CLAUDE.md` files collected from 242 open-source repositories, extracting their heading structure in order to characterize both how manifests are organized and what they contain. Their results are twofold. Structurally, manifests follow a shallow hierarchy, typically comprising a single main heading and a moderate number of subsections, rather than deeply nested documents. In terms of content, three patterns dominate: instructions for building and running the project, implementation details, and architectural descriptions. The authors interpret this distribution as evidence that manifests are fundamentally action-oriented artifacts, written to tell an agent how to operate within a project rather than to document it for human readers.

Our first study (Chapter 3) investigates the same class of artifact, but differs in three respects. First, our dataset is drawn from a smaller but more strictly filtered set of repositories—the 100 most popular projects that satisfy explicit popularity, language, and real-world application criteria—and it resolves *memory bank* files, that is, manifests whose content consists only of pointers to other markdown documents, by retrieving the referenced files instead. Second, we go beyond the identification of individual concerns and mine *maximal co-occurrence patterns* among them, in order to characterize the most common patterns of configuration files rather than isolated section frequencies. Third, we explicitly measure the presence of non-textual elements, namely embedded code examples, internal and external links, and diagrams, an aspect not addressed by either study. More importantly, none of the works in this group addresses the *quality* of these manifests, which is the subject of our second study (Chapter 4).

2.4.3 Bad Smells in Source Code and Configuration Artifacts

The concept of *bad smells* was originally formalized in the context of source code, to capture design and implementation flaws that degrade maintainability and readability. Over the years, empirical research has investigated the phenomenon from several angles. Tufano et al. [42] analyzed the change history of open-source projects to determine when and why smells are introduced, showing that a substantial fraction of smell instances are present from the very first version of the affected artifact, rather than emerging gradually through maintenance. This is a direct antecedent of the *Init Fossilization* smell we describe in Chapter 4, where the initial version generated by an `/init` command is never subsequently revised. Palomba et al. [28] complemented this perspective by surveying developers on their perception of smell severity, finding that not all smells are perceived as equally harmful and that perception depends heavily on the developer's familiarity with the codebase. Subsequent work systematized this body of knowledge into broader taxonomies [8] and extended it to architectural styles such as microservices [41].

A related line of research investigated how smells relate to one another rather than in isolation. Palomba et al. [29] and Muse et al. [26] applied association rule mining and correlation analysis to show that code smells rarely occur alone, and that certain combinations are far more frequent than chance would predict. Hamdi et al. [14] extended this analysis to the temporal dimension, studying how smell co-occurrences evolve across releases. Association rule mining has, more broadly, an established tradition in software engineering research, having been applied to detect usage patterns in APIs [20], to associate source code files that change together [40, 47], and to relate bug types [9, 24]. We adopt the same methodological family in both studies of this dissertation: FP-Max in Chapter 3, to identify maximal sets of co-occurring concerns, and Apriori in Chapter 4, to identify co-occurring smells.

More recently, the software engineering community extended smell taxonomies beyond imperative source code to declarative and infrastructure artifacts. In the context of Infrastructure as Code, Rosa et al. [33] investigated Dockerfile smells, characterizing violations that impair build performance, image size, and reproducibility, and going further to study whether and how developers actually fix them once they are reported. Their findings indicate that although automated detection is feasible, adoption of the proposed fixes depends on whether maintainers perceive the violation as consequential, an observation that also applies to the catalog we propose. Similarly, Urdih et al. [44] cataloged cache-related smells in GitLab CI/CD pipelines, reporting that 89% of the analyzed projects exhibited at least one pipeline configuration issue, and proposing automated detection for each cataloged violation.

These works establish the methodological template we follow in Chapter 4: catalog

the violations, propose detection heuristics, and measure prevalence in real projects. There is, however, a fundamental difference in the nature of the artifact. Dockerfiles and CI/CD pipeline definitions are consumed by deterministic engines, where a violation produces a reproducible and diagnosable effect: a larger image, a slower build, an invalidated cache. Agent configuration files, in contrast, are consumed by probabilistic inference engines. A contradictory instruction does not raise an error; it silently biases the model toward one behavior or another, and may do so inconsistently across sessions. Similarly, a redundant rule does not fail; it consumes context budget and competes for attention with instructions that matter. This distinction means that neither the existing catalogs nor their detection techniques transfer directly, and it is what justifies a dedicated catalog for this class of artifact.

Taken together, the three groups reviewed above delimit the gap addressed by this dissertation. Research on agent configuration artifacts has so far characterized what these files contain, without examining whether their content is adequate; research on smells in declarative artifacts has produced mature catalogs and detection techniques, but exclusively for artifacts consumed by deterministic engines. The two studies presented in the following chapters address this intersection: Chapter 3 characterizes how developers populate and structure these manifests in practice, and Chapter 4 investigates the quality problems that affect them.

2.5 Final Remarks

This chapter reviewed the foundational concepts and the empirical literature supporting this dissertation. We examined how Large Language Models behave probabilistically, the systemic constraints that make context engineering necessary, and the architectural evolution toward autonomous coding agents built on harnesses and tool inventories. We then introduced agent configuration files as the artifact that bridges the gap between what an agent can discover by inspecting a repository and what it must be told explicitly, and we discussed how these files are loaded, bootstrapped, and maintained. Finally, we reviewed the related literature and positioned this dissertation at the intersection of two research lines: the characterization of agent configuration artifacts, which so far has not addressed their quality, and the study of smells in declarative configuration artifacts, which so far has targeted only deterministic tools.

Chapter 3

Decoding the Configuration of AI Coding Agents

In this chapter, we present the first empirical study of this dissertation, investigating the configuration ecosystem of Claude Code projects to understand how developers structure, specify, and evolve governance files for autonomous coding agents. The Claude Code tool and its `CLAUDE.md` configuration mechanism were introduced in Section 2.3. Section 3.1 states the research questions addressed here. Section 3.2 describes the study design, detailing the dataset construction, manual categorization, and frequent pattern mining methodology. Sections 3.3, 3.4, and 3.5 present the empirical results corresponding to each research question. Section 3.6 outlines the threats to validity, and Section 3.7 concludes with final remarks.

3.1 Research Questions

To understand how coding agents are configured in practice, we collected and analyzed a curated dataset of 328 configuration files from popular Claude Code projects, investigating three research questions:

- **RQ1: What concerns and practices are specified in agent configuration files?** We aim to identify the software engineering aspects and operational rules most frequently established by developers when configuring coding agents.
- **RQ2: In addition to textual rules, do configuration files include other elements such as code examples, links, and diagrams?** We evaluate the structural richness of configuration manifests by measuring the presence of non-textual elements, such as source code snippets, internal/external links, and architectural diagrams.

- **RQ3: What are the most common patterns of `CLAUDE.md` files?** We analyze the co-occurrence of concerns and practices to identify recurring archetypes of configuration manifests, providing reference models for developers.

3.2 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 three RQs proposed in the research. Figure 2.5, in Section 2.3, shows a representative example of the artifact analyzed in this study.

3.2.1 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 [4].

2. *Language.* We excluded non-English repositories, by checking its name and description.

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)[1], `oven-sh/bun`[2] (a popular JavaScript runtime environment), and `callstack/react-native-testing-library` (React Native testing library).[3] 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.

3.2.2 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 2.5). 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 3.1 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.

After following the previous procedures, we extracted 2,492 section titles. These

[1] https://github.com/modelcontextprotocol/python-sdk
[2] https://github.com/oven-sh/bun
[3] https://github.com/callstack/react-native-testing-library

Table 3.1: Size of Claude.md files (measured in number of ## sections).

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

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 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 (## 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).

3.2.3 Pattern Analysis

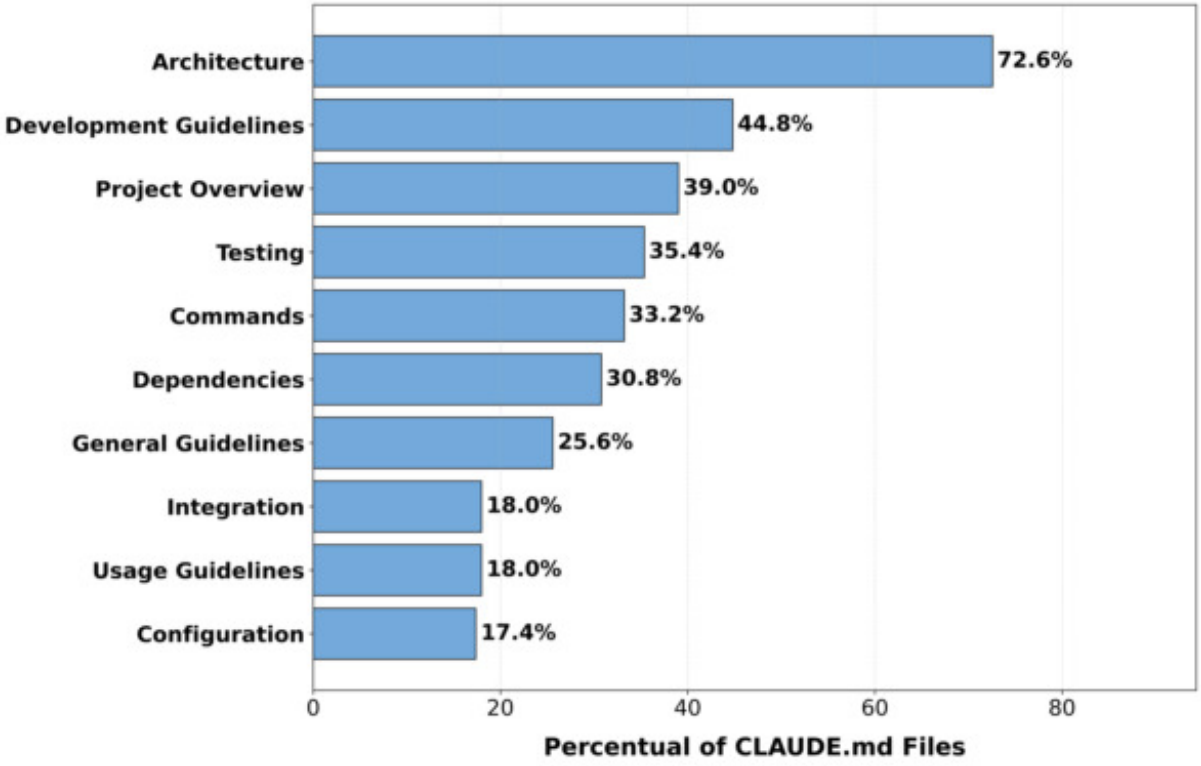
We used the FP-Max algorithm to detect which concerns and practices are mentioned together [12]. FP-Max is a variant of the Apriori algorithm [1] that focuses on finding maximal frequent itemsets, i.e., the largest independent sets of items that frequently co-occur in a dataset. We relied on this algorithm 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,[4] configured with 0.15 minimum support.

[4] https://rasbt.github.io/mlxtend/

3.3 Most Common Concerns and Practices (RQ1)

Figure 3.1 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.

Figure 3.1: Concerns and practices addressed in agent config. files



Software Architecture: Essentially, these sections define the software architecture the coding agent should follow. For example, Figure 3.2 shows an example of such guidelines in `evstack/ev-node`.[5] 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*).

[5] https://github.com/evstack/ev-node/blob/main/CLAUDE.md

Figure 3.2: Architecture guidelines (evstack/ev-node).

```
## 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
- **DA** (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
```

Source: Adapted from evstack/ev-node.

Development Guidelines: This section aims to guide the AI agent on low-level development issues. Figure 3.3 shows an example from the `PrefectHQ/marvin` project.[6] It includes rules for typing, installing dependencies, navigating on the code, etc.

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 3.4, which clarifies the purpose and goals of the React Native Testing Library (RNTL).

Testing: This section defines testing concerns and practices, such as in the example in Figure 3.5, where the configuration specifies the types of tests that the agents should create (unit, integration tests, and mocks) and the test commands.

[6] https://github.com/PrefectHQ/marvin/blob/main/CLAUDE.md

Figure 3.3: Development guidelines (PrefectHQ/marvin).

```
## Development Guidelines

### Type Hints
- Use `X | Y` instead of `Union[X, Y]`
- Use builtins like `list`, `dict` instead of `typing.List`, `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 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
```

Source: Adapted from PrefectHQ/marvin.

Figure 3.4: Project overview (callstack/react-native-testing-library).

```
## 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.
```

Source: Adapted from callstack/react-native-testing-library.

Figure 3.5: Example of testing guidelines (lightdash/lightdash).

````
## 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
````

Source: Adapted from 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).

3.4 Usage of Code Examples and Links (RQ2)

The goal of this research question is to investigate the richness of the analyzed documentation by identifying the prevalence of non-textual content, such as source code and external references. Table 3.2 presents the occurrence of source code examples and links in the analyzed `CLAUDE.md` files. The values represent the percentage of files within each specific category that contain at least one instance of these elements. As shown, the highest percentage of code examples appears in the Development Guidelines category (17.68% of the instances classified in this category).

Table 3.2: 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%
````

An example is provided in Figure 3.6, which includes a code snippet demonstrating how to configure a custom database table using Supabase.

Figure 3.6: Config. file with code example (run-llama/LlamaIndexTS).

````
## Common Patterns

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

Source: Adapted from run-llama/LlamaIndexTS.

Conversely, links were most frequently found in the Architecture category, though in only 1.83% of the files. Figure 3.7 illustrates an 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 Mermaid[7] tool, but found only two instances.

Figure 3.7: Config. file with a link to a project's page (grafana/mimir).

```
## Design patterns and Code conventions

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

Source: Adapted from 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%).

3.5 Patterns of Configuration Files (RQ3)

Table 3.3 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

[7] https://mermaid.js.org
```

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.

Table 3.3: Top-5 Most Popular `Claude.md` Patterns.

Practices & Concerns	**% Files**
Architecture, Dependencies, Project Overview *Example: Repository Structure, Key Dependencies, Project Overview*	21.6
Architecture, General Guidelines *Example: Repository Structure, Specialized Agent Usage Guidelines*	20.1
Architecture, Development Guidelines, Project Overview *Example: Package Structure, Component Development Guidelines, Overview*	19.8
Architecture, Development Guidelines, Testing *Example: Code Architecture, Development Workflow, Testing*	18.9
Architecture, Integration *Example: Architecture Overview, External Integrations*	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.

3.6 Threats to Validity

Even though this is an initial study, we highlight three 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.

3.7 Final Remarks

In this chapter, we presented an empirical investigation into the structure and content of 328 configuration files from popular Claude Code repositories. Our analysis revealed that developers primarily use these manifests to convey architectural constraints, development guidelines, and testing procedures. Furthermore, architecture emerged as a foundational element present across all top-5 maximal co-occurrence patterns.

While these files provide valuable declarative guidance for coding agents, their manual construction and maintenance by human developers can introduce qualitative deviations, ambiguities, and redundancies. In Chapter 4, we extend our exploration of agent governance by investigating recurrent misconfiguration issues, formalizing a catalog of *configuration smells*, and developing automated heuristics for their detection.

Chapter 4

Configuration Smells in AGENTS.md Files

In this chapter, we present the second empirical study of this dissertation, focusing on the quality and maintainability of agent governance files by introducing the first formal catalog of configuration smells for coding agents. The role and the loading mechanism of `AGENTS.md` and `CLAUDE.md` files were introduced in Section 2.3. Section 4.1 details our study design and methodology, combining a grey literature review, repository dataset curation, and pull request mining. Section 4.2 formally defines the six cataloged configuration smells. Section 4.3 introduces the automated detection heuristics, followed by Section 4.4, which reports their empirical prevalence across 100 open-source repositories. Section 4.5 analyzes the co-occurrence patterns of these smells using association rule mining. Section 4.6 discusses threats to validity, and Section 4.7 presents final remarks.

4.1 Methods

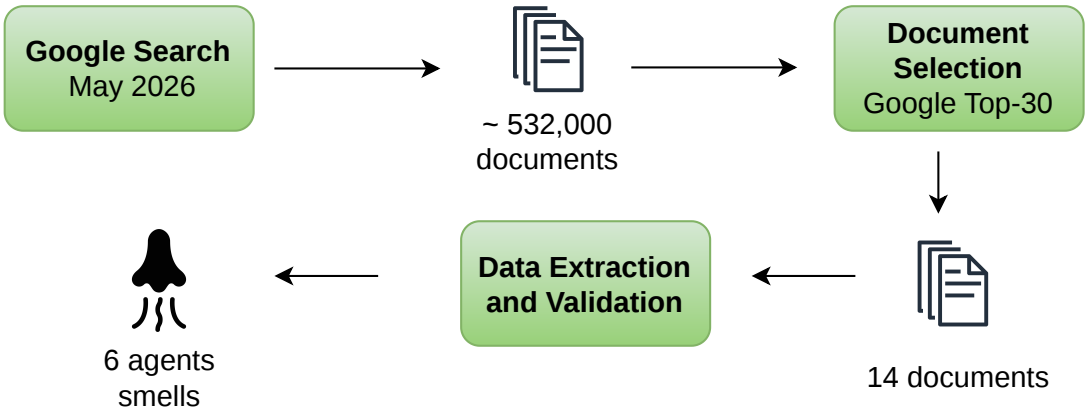
To research the smells that affect agent configuration files, we employed three methods: a grey literature review (Section 4.1.1), the creation of a dataset of real-world agent configuration files (Section 4.1.2), and an analysis of discussions conducted in pull requests (Section 4.1.3).

4.1.1 Literature Review

Grey literature review is commonly used in software engineering research to gather information for emerging topics, since it emphasizes the inclusion of non-academic sources,

such as blog posts, technical reports, and documentation [11]. Our procedure is depicted in Figure 4.1 and consists of three main steps: (a) Google search, (b) Document selection, and (c) Data extraction and validation.

Figure 4.1: Overview of the grey literature steps



Source: Santos et al. [35].

Document Search: We started by performing a Google search to identify documents that describe bad practices or smells in `AGENTS.md` files. Similar to other grey literature reviews, we restricted our search to Google due to its wide coverage and relevance for retrieving non-academic sources [45, 46, 34]. Specifically, the search query, as presented in Figure 4.2, looked for documents containing the terms *agents.md* or *claude.md* together with terms related to *bad smells*, *anti-patterns*, and *best practices*.

Figure 4.2: Search string used in the literature review.

```
("agents.md" OR "AGENTS.md" OR "claude.md" OR "CLAUDE.md")
AND
("bad smell" OR "bad smells" OR "anti-pattern" OR "anti-patterns"
 OR "antipattern" OR "antipatterns" OR "bad-practice"
 OR "bad-practices" OR "bad practice" OR "bad practices"
 OR "best-practices" OR "best practice" OR "best practices")
```

Source: Santos et al. [35].

Document Selection: Our search returned 532,000 documents. From these results, we manually analyzed the first 30 documents returned by the search; i.e., the ones returned in the first three pages. We stopped the analysis at the third page of results following a saturation criterion [11]: the documents on the third page did not introduce any new smell, only mentioning smells already identified in the first two pages. This cutoff is also consistent with previous grey literature reviews [45, 46]. The first author discarded 16

documents according to the following criteria: (a) five documents were accessible only through subscription or paywall; (b) four documents were out of scope, i.e., focused on specific tools or other topics; (c) three represented discussion threads on forums; (d) three documents did not explain any bad smell or best practice on `AGENTS.md` files, they only listed these issues pointing to the official documentation of the tools; and (e) one document was written in a non-English language. As a sanity check, the third author analyzed each discarded document to confirm the reason for exclusion; no disagreements were found in this process. The remaining 14 documents were selected for the next step. We also assessed the quality and reputation of the authors of all selected documents. Six documents were published by well-known companies, such as Anthropic and GitHub. The remaining eight documents were written by authors with relevant expertise in the field (seven authors have at least six years of experience, and one author has two years). Furthermore, three of the articles were extensively discussed on Hacker News (a well-known forum for technology and startup discussions), receiving at least 140 upvotes each. Throughout this chapter, we refer to these documents using the identifier D_n, where n is a numeric identifier. The list of selected documents is available in our replication package.

Smells Extraction and Validation: For each selected document, the first author thoroughly read its content and marked any sentence that described practice to avoid when creating `AGENTS.md` files. Following this initial extraction, the second author reviewed the annotations as a sanity check to confirm whether they actually describe a bad smell. This process revealed four disagreements, which both authors discussed to reach a consensus. After this annotation process, the first and last authors manually grouped sentences describing similar smells and proposed a name and a short description for each group. They also decided to discard one smell that is not specific to coding agents: exposing secret keys in configuration files. At the end, we obtained a list of six smells.

4.1.2 Dataset of Agent Configuration Files

To investigate whether the smells actually occur in real projects, we created a dataset with 100 agent configuration files, including 39 `AGENTS.md` and 61 `CLAUDE.md` files. We selected these files using the following steps. In January 2026, we used the GitHub Search API to find repositories containing either `AGENTS.md` or `CLAUDE.md` in their root folder. We then applied the same selection criteria described in Section 3.2.1, discarding repositories that are not real-world applications, e.g., awesome lists and tutorials, and selecting the top-100 repositories with the highest number of stars. Two differences with

respect to the dataset of Chapter 3 are worth noting: this dataset was collected five months later, and it includes `AGENTS.md` files in addition to `CLAUDE.md` ones. For example, our dataset of selected GitHub projects includes `n8n-io/n8n` (a workflow automation platform)[1], `langchain-ai/langchain` (an agent engineering platform)[2], and `vercel/ai` (an AI toolkit for TypeScript).[3]

4.1.3 Pull Requests Analysis

After creating the dataset of files described in the previous section, we decided to inspect the Pull Requests (PRs) of the corresponding projects to determine whether they contained discussions about potential issues in agent configuration files. Such issues could help expand the list of six smells identified through our grey literature review. Next, we explain how the pull requests were selected and analyzed.

Pull Requests Selection: We collected all commits associated with each of the 100 configuration files in our dataset, including modifications, additions, and deletions. This process yielded an initial dataset composed of 760 commits. From this initial dataset, we selected only the ones linked to pull requests, resulting in a total of 383 pull requests.

A manual inspection was then performed on the conversation history of each filtered pull request to check whether the ongoing discussions about the target files, `CLAUDE.md` and `AGENTS.md`, were in fact of substantive relevance. We discarded all PRs that met any of the following exclusion criteria: (a) conversations consisting entirely of bot-generated messages; (b) threads where human participants interacted exclusively with bots; (c) discussions that failed to actively engage with or reference the `AGENTS.md` or `CLAUDE.md` files; (d) PRs containing only the initial opening description without any subsequent discussion or peer review. The application of these sequential filters yielded a final corpus of 17 relevant pull requests.

Pull Requests Analysis: After analyzing the 17 pull requests, we found discussions about configuration-related problems in only two of them. These pull requests were labeled PR_1 and PR_2 and are also included in our replication package. However, the discussions concerned smells that had already been identified in the grey literature review. Therefore, our list remained unchanged, containing six smells. Although no new smells were discovered, the analysis of PRs was still valuable because it provided concrete examples of

[1] https://github.com/n8n-io/n8n
[2] https://github.com/langchain-ai/langchain
[3] https://github.com/vercel/ai

discussions about smells in agent configuration files, which we use to illustrate the content of Section 4.2.[4]

4.2 Configuration Smells

Table 4.1 presents a list of the smells identified in our study. The table also shows the identifiers of the articles in which these smells were found. In the remainder of this section, we provide a description of the smells listed in this table.

Table 4.1: Configuration Smells

Smells	**Articles and PRs**
Context Bloat	D_1, D_3, D_4, D_5, D_6, D_9, D_{11}, D_{12}, D_{13}, D_{14}, PR_1
Skill Leakage	D_2, D_3, D_4, D_5, D_6, D_{11}, D_{12}, D_{13}, PR_2
Lint Leakage	D_1, D_4, D_5, D_6, D_9, D_{13}
Blind Reference	D_6, D_7
Init Fossilization	D_4, D_6, D_9, D_{10}, D_{13}
Conflicting Instructions	D_1, D_3, D_6

4.2.1 Context Bloat

This was the most frequently cited smell in our documents, being mentioned in 10 out of the 14 reviewed articles and in one PR. It occurs when an `AGENTS.md` file becomes excessively large and overloaded with rules, examples, or low-priority details. Bloated configuration files increase token consumption, raise costs, and reduce the visibility of important instructions. Therefore, `AGENTS.md` files should remain concise and focused on essential project-specific guidance. For example, Anthropic's documentation explicitly recommends the following: *target under 200 lines per* `CLAUDE.md` *file. Longer files consume more context and reduce adherence.*[5]

To provide another example, in one of the pull requests we analyzed (PR_1), the authors proposed restructuring the `AGENTS.md` file to reduce its size. The PR explicitly

[4] As an additional note to this analysis, in six PRs, the goal was to consolidate agent-related documentation into a single file. For example, one PR was described as a pure rename of `CLAUDE.md` to `AGENTS.md`.

[5] https://code.claude.com/docs/en/memory

states that the project's configuration file was reduced from 598 to 149 lines, because *modern LLMs tend to perform better with configuration files containing approximately 150 to 200 lines.*

4.2.2 Skill Leakage

This smell occurs when specific, rarely used, or highly context-dependent instructions are placed in the `AGENTS.md` file instead of being specified in dedicated skill files (e.g., `skills.md`) and loaded on demand. In practice, this means that specialized knowledge "leaks" into every agent session, even when it is not needed. As a result, the agent's context becomes larger, more expensive, and harder to maintain. Furthermore, such rules may compete for attention with the rules that are actually critical for the project. For example, one of the articles explicitly recommends the following: *Instead of including all your different instructions about building your project, running tests, code conventions, or other important context in your CLAUDE.md file, we recommend keeping task-specific instructions in separate markdown files with self-descriptive names somewhere in your project.* (D_{13})

4.2.3 Lint Leakage

This smell occurs when an `AGENTS.md` file includes rules that are already checked by linters, formatters, or other static analysis tools. Typical examples include naming conventions (such as camelCase or PascalCase), formatting rules, import ordering, maximum line length, or generic style-guide recommendations. Because these constraints are automatically checked by local tools, repeating them in `AGENTS.md` adds limited value while unnecessarily increasing the agent's context size [17]. Moreover, emphasizing such coding rules can divert the model from focusing on more important project-specific concerns, such as architectural constraints, domain rules, or safety policies. For example, one of the articles explicitly recommends the following: *Code style enforcement is the biggest trap. Formatting, indentation, import ordering: these are deterministic problems with deterministic solutions. Linters and formatters like Biome, ESLint, or Ruff handle them faster, cheaper, and with 100% consistency. Spending instruction budget on style rules is dead weight: the same work a pre-commit hook does for free.* (D_6)

4.2.4 Blind References

This smell occurs when an `AGENTS.md` file contains references to external documents, files, or directories without explaining their purpose or scope. As a consequence, the agent may unnecessarily load large documents into context, ignore important references, or fail to prioritize the correct source of information for a given task. Thus, a better practice is to complement references with concise descriptions explaining the role of the document, the type of information it contains, and the context in which it should be used. For example, one document explicitly recommends the following: *If you just mention the path [of an external document], Claude will often ignore it. You have to pitch the agent on why and when to read the file.* (D_7)

4.2.5 Init Fossilization

This smell occurs when the configuration file is generated by an initialization command such as `/init` but not reviewed or updated afterwards. Thus, the file generated by the coding agent becomes the permanent configuration, often carrying instructions that are not relevant anymore to the project. As a result, the configuration tends to accumulate noise, increase context consumption, and reduce the overall effectiveness of the agent over time. For example, Anthropic's documentation explicitly recommends that the configuration file should be continuously updated, such as when: *Claude makes the same mistake a second time; a code review reveals something Claude should already have known about the codebase; you find yourself typing the same correction or clarification in chat that you already provided in a previous session; and a new team member would need the same context in order to be productive.*[6]

4.2.6 Conflicting Instructions

This smell occurs when an `AGENTS.md` file contains instructions that contradict each other, creating ambiguity about the expected behavior of the agent. Such inconsistencies

[6] https://code.claude.com/docs/en/memory

can confuse the model and lead to unstable results. For example, small variations in the prompt can result in different agent behavior, because they are enough to make the agent follow a different inference path than the one followed previously. Anthropic's documentation explicitly recommends the following: *if two rules contradict each other, Claude may pick one arbitrarily. Review your CLAUDE.md files periodically to remove outdated or conflicting instructions.*[7]

4.3 Detection Heuristics

We also propose a set of heuristics to detect the smells described in the previous section.

4.3.1 Heuristic based on Lines of Code

This heuristic is used exclusively to detect the *Context Bloat* smell. Specifically, we decided to use a threshold of 200 lines of code to identify this smell, as also suggested in the Anthropic document mentioned in Section 4.2.1. In other words, `AGENTS.md` files with 200 or more lines of code are classified as presenting the *Context Bloat* smell.

4.3.2 Heuristic based on Language Models

To detect the *Skill Leakage*, *Lint Leakage*, *Blind References*, and *Conflicting Instructions* smells, we decided to use a large language model. We designed a dedicated prompt for each smell, thereby requesting a more specific and objective task from the model. Figure 4.3 illustrates the baseline template of these prompts. In the case of *Blind References* and *Conflicting Instructions*, we also enriched the basic prompt with examples of true/false positives. For example, in Figure 4.4 we show the example (true positive) added to the prompt to detect *Blind References*.

[7] https://code.claude.com/docs/en/memory

Figure 4.3: Prompt to detect Skill Leakage, Lint Leakage, Blind References, and Conflicting Instructions.

```
# Context

You are a senior software engineer. I will provide
an agent configuration file (AGENTS.md). Your task
is to detect whether the file contains the
following configuration smell.

# Smell: [name]

Description: [as in Section IV; essentially the
first paragraph of the smell description]

[Optional] Operational Guidelines
- True & False Positives Examples
- Edge cases specific to the smell to minimize false positives.

# Output

- If detected: [Return exact lines /
  JSON object with contradiction context]
- If not detected, return only: NO SMELL

# AGENTS.md file

[file content]
```

Figure 4.4: Example added to the prompt to detect Blind References

```
...
## Releases & Environment

For releases or environment issues, see
`web/book/src/project/contributing/development.md`.
```

4.3.3 Heuristic based on Number of Commits

This heuristic is used exclusively to detect the *Init Fossilization* smell. Conservatively, we consider that an `AGENTS.md` file with only a single commit exhibits this smell; that is, the file has never been modified since its creation.

4.4 Configuration Smells in the Wild

To detect the smells in our dataset of 100 agent configuration files, we applied the heuristics proposed in Section 4.3. For the heuristics based on LLMs, we used `gemini-3.1-flash-lite`, with a temperature of 0. Additionally, each smell instance identified by these heuristics was carefully reviewed by the first author to verify whether it is a true occurrence of the smell. The results are summarized in Table 4.2 and are discussed in the following subsections. In this table, we also show the number of false positives and the precision, for the smells whose identification relies on LLM-based heuristics. In other words, we did not compute precision for *Context Bloat* and *Init Fossilization* because, in these cases, detection is based on pre-established thresholds.

Table 4.2: Smells detected in real projects
(FP: False Positives; Prec: Precision)

	Instances	**FP**	**Prec. (%)**
Context Bloat	42	-	-
Skill Leakage	35	6	82
Lint Leakage	62	4	93
Blind Reference	16	2	87
Init Fossilization	24	-	-
Conflicting Instructions	28	12	57

Summary: We confirmed at least one smell in 91 agent configuration files. Thus, only nine files were found to be smell-free. These results suggest that developers could benefit from catalogs and tools designed to spot configuration issues in agent configuration files.

4.4.1 Context Bloat

The proposed heuristic detected 42 cases of *Context Bloat*. The smallest file contains 216 lines of code, whereas the largest file contains 1,477 lines of code. Due to space constraints, we will not present a complete example of *Context Bloat* here. However, just to provide a high-level illustration, the `CLAUDE.md` file of the `javascript-obfuscator` project—a very popular and powerful obfuscator of JavaScript and Node.js source code—has 1,477 lines.[8] This file is organized into 27 sections, including Project Overview, Architecture Overview, Core Workflow, CLI/API Usage, etc. As a result, the file is very large for repeated context loading, which can lead language models to discard important instructions. Much of these instructions would be better maintained in separate documentation or loaded on demand through skills.

When analyzing the file, we noticed, for example, that the second section is called *Key Features*, and has 22 lines. This section describes the obfuscation techniques used by the project, such as variable and function renaming, string extraction and encryption, dead code injection, and control-flow flattening. This information is mostly product documentation and provides limited value as persistent context for agents. Therefore, it could be removed from the agent configuration and described in the project's README, for example.

It is also important to note that *Context Bloat* is a more visible smell, which makes it easier to detect. The root cause of this smell is the presence of other smells in the configuration file, such as *Skill Leakage*, which we will discuss next.

4.4.2 Skill Leakage

The proposed heuristic detected 35 cases of *Skill Leakage*. After a manual analysis conducted by the first author, 29 cases were confirmed (82%). An example of *Skill Leakage* was found in the `AGENTS.md` file of `quickemu-project/quickemu`, which is a tool to simplify the creation and execution of virtual machines.[9] The detected smell instance is shown in Figure 4.5. As we can see, the section *Adding a new OS to quickget* contains instructions that are only useful for a specific subset of tasks. Since most interactions with the coding agent do not involve adding new operating systems, these instructions unnecessarily increase the size of the configuration file. Thus, they would be better placed

[8] https://github.com/javascript-obfuscator/javascript-obfuscator
[9] https://github.com/quickemu-project/quickemu

in a dedicated skill or documentation file. Beyond the `quickemu` case, two other confirmed instances illustrate this smell: (a) `promptfoo/promptfoo` (Figure 4.6), which describe a workaround to attach screenshots to pull request descriptions, since GitHub provides no API for this task; and (b)`TryGhost/Ghost` (Figure 4.7): the local setup of its analytics service, Tinybird. Both instructions are useful, but each one is relevant only when that specific task is performed.

Figure 4.5: Example of Skill Leakage (quickemu-project/quickemu).

```
## Adding a new OS to quickget

Follow the [guide in the wiki](...). Each OS requires:

1. Entry in `os_info()` case statement
2. `releases_<os>()` function returning available versions
3. `editions_<os>()` function if multiple editions exist
4. `arch_<os>()` function if ARM64 is supported
   (defaults to amd64 only if omitted)
5. Download URL construction logic
```

Most common leaked skills: We also manually classified the skills responsible for the identified *Skill Leakage* instances. The results of this classification are presented in Table 4.3. As can be seen, the most common skills incorrectly defined in `AGENTS.md` files are related to testing concerns, followed by workflow guidelines (e.g., procedures for code reviews, pull requests, and issue management). Particularly, the Skill Leakage instance presented in Figure 4.5 was classified as scaffolding, as it defines functions that must be implemented to support a new operating system image within a module.

Table 4.3: Most common leaked skills

Skill Type	**Frequency**
Testing	10
Workflow	8
Scaffolding	4
Infrastructure	4
Architecture	3

Figure 4.6: Example of Skill Leakage (promptfoo/promptfoo).

````
## Screenshots for Pull Requests

GitHub has no official API for uploading images to PR descriptions.
When asked to add screenshots to a PR:

1. **Take the screenshot** using browser tools or other methods
2. **Upload to freeimage.host** (no API key required):

```bash
curl -s -X POST \
  -F "source=@/path/to/screenshot.png" \
  -F "type=file" \
  -F "action=upload" \
  "https://freeimage.host/api/1/upload?key=..." \
  | jq -r '.image.url'
```

3. **Update the PR body** with the returned URL: (...)

**Do NOT:**
- Commit screenshots to the branch
- Upload to GitHub release assets
- Use GitHub's internal upload endpoints
  (require browser cookies, not PATs)
````

Figure 4.7: Example of Skill Leakage (TryGhost/Ghost).

```
### Analytics (Tinybird)
- **Local development:** `yarn docker:dev:analytics`
  (starts Tinybird + MySQL)
- **Config:** Add Tinybird config to
  `ghost/core/config.development.json`
- **Scripts:** `ghost/core/core/server/data/tinybird/scripts/`
- **Datafiles:** `ghost/core/core/server/data/tinybird/`
```

4.4.3 Lint Leakage

The proposed heuristic detected 62 cases of *Lint Leakage*. After a manual analysis conducted by the first author, 58 cases were confirmed (93%). An interesting case of this smell was identified in the `google/adk-python` project, which is an open-source SDK for building agent-based applications in Python.[10] As shown in Figure 4.8, the `AGENTS.md` file of this project includes a section called *Python Style Guide* containing instructions for writing Python code, including recommendations for indentation and line length, naming conventions, usage of docstrings, etc. Normally, these recommendations are enforced by linters, formatters, or widely adopted community conventions, making their inclusion in the file unnecessary.

Figure 4.8: Example of Lint Leakage (google/adk-python)

```
### Python Style Guide

* Indentation: 2 spaces.
* Line Length: Maximum 80 characters.
* Naming Conventions**:
  * `function_and_variable_names`: `snake_case`
  * `ClassNames`: `CamelCase`
  * `CONSTANTS`: `UPPERCASE_SNAKE_CASE`
* Docstrings: Required for all public modules, ...
* Imports: Organized and sorted.
* Error Handling: Specific exceptions should be ...
```

Interestingly, after creating our dataset (in January, 2026), we found that the project maintainers performed a major refactoring of the `AGENTS.md` file. Specifically, the *Python Style Guide* section, shown in Figure 4.8, was moved to a separate skill file. Therefore, this extraction confirms the relevance of the smell we initially detected in our dataset.

4.4.4 Blind Reference

First, to help explain this smell, Figure 4.9 shows an example in which an external reference is cited appropriately. Notice that the text references an external dependency,

[10] https://github.com/google/adk-python

includes a link to its GitHub repository, and provides a brief explanation of its purpose (*cdp-use only provides shallow typed interfaces for the websocket calls*). Consequently, the agent is able to understand the role of the dependency without needing to load or inspect the external repository directly.

Figure 4.9: External reference described with context (browser-use/browser-use)

```
## CDP-Use

We use a thin wrapper around CDP called cdp-use:
https://github.com/browser-use/cdp-use. cdp-use only provides shallow
typed interfaces  for the websocket calls, all CDP client and session
management + other CDP helpers still  live in
browser_use/browser/session.py.
```

However, after applying the proposed heuristic, we were able to detect 16 instances of *Blind Reference*, i.e., references cited in `AGENTS.md` files without appropriate context. After a manual analysis, 14 cases were confirmed (87%). An example is shown in Figure 4.10. As we can see, this configuration references an external document (`docs/plugin-reorg.md`) to explain the planned plugin system, but it does not provide any contextual information about the document itself. Therefore, the agent would need to load and inspect the referenced file to understand the architecture and goals of the plugin system.

Figure 4.10: Blind Reference (SuperClaude-Org/SuperClaude_Framework)

```
### Plugin System (v5.0 - Not Yet Available)

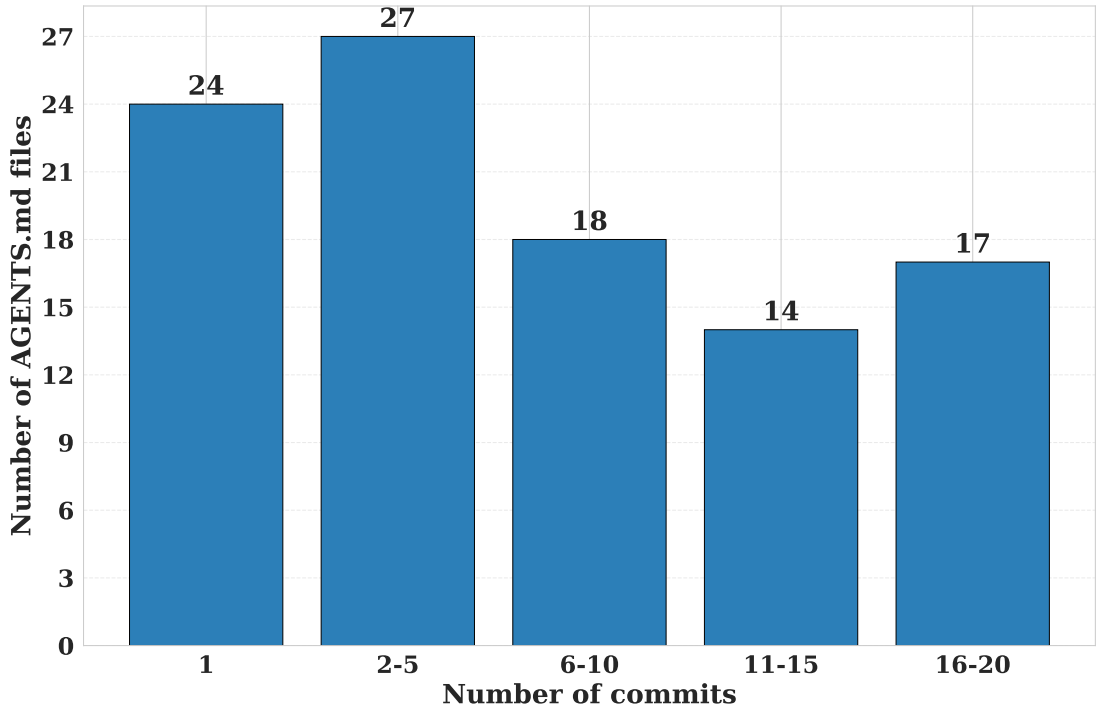
The TypeScript plugin system (`.claude-plugin/`, marketplace) is planned
for v5.0.
See `docs/plugin-reorg.md` for details.
...
```

4.4.5 Init Fossilization

We detected 24 cases of *Init Fossilization*, that is, `AGENTS.md` files with a single commit, as illustrated by the histogram in Figure 4.11. The histogram also shows that

configuration files are frequently updated in practice, thus reinforcing our argument that the absence of changes in such files is indeed a smell. For example, 14 of the analyzed files (14%) have between 11 and 15 commits, while 17 files (17%) have between 16 and 20 commits.

Figure 4.11: Number of changes in `AGENTS.md` files (*Init Fossilization* corresponds to files with a single commit)

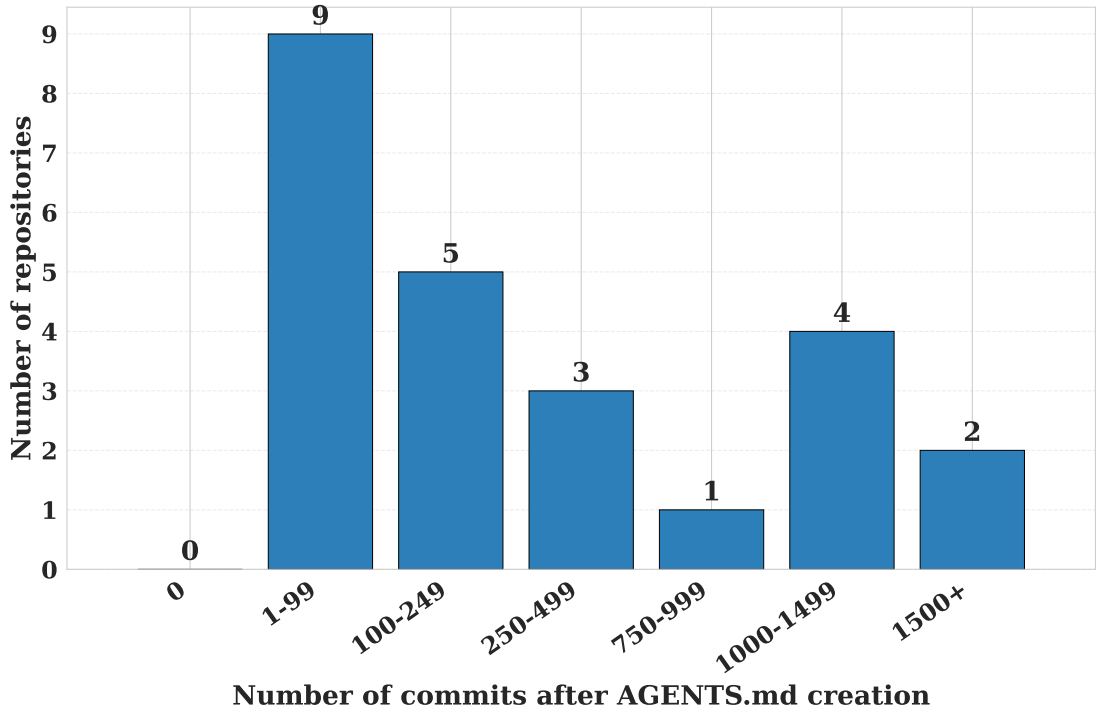


However, it is possible that the 24 projects exhibiting *Init Fossilization* were dormant projects, that is, projects with limited activity and few commits. Therefore, the histogram in Figure 4.12 shows the total number of commits made to these projects after the creation of their respective `AGENTS.md` files. As the histogram indicates, the hypothesis that these projects were inactive was not supported. In fact, we did not find a single project in such a situation, that is, with zero commits after the creation of the `AGENTS.md` file. On the contrary, we observed many projects with a substantial number of commits, including two projects with more than 1,500 commits, for example.

4.4.6 Conflicting Instructions

The proposed heuristic detected 28 cases of *Conflicting Instructions*. However, after a manual analysis conducted by the first author, only 16 cases were confirmed (57%). This lower precision is, to some extent, understandable, since identifying con-

Figure 4.12: Number of commits in projects exhibiting *Init Fossilization* (counting only commits made after the creation of `AGENTS.md`).



tradictory instructions is indeed a more complex task. Figure 4.13 shows an example of this smell. As can be observed, the configuration specifies two directory paths for creating new components. An agent cannot satisfy the requirement to place components in `packages/ui/components` while simultaneously following the instruction to create them in `packages/components`.

Figure 4.13: Example of Conflicting Instructions (inkline/inkline)

```
# Component Guidelines
- Components should be placed in the `packages/ui/components` directory
- ...

## How to create a new component

- Create a new folder in `packages/components` with the name of the
component.
```

Table 4.4: Co-existing smells detected by Apriori.

Antecedent		Consequent	Support	Confidence	Lift
Conflicting Instructions, *Skill Leakage*	→	*Context Bloat*	0.05	0.83	1.81
Init Fossilization, *Skill Leakage*	→	*Lint Leakage*	0.05	0.83	1.31
Conflicting Instructions	→	*Context Bloat*	0.14	0.81	1.76
Skill Leakage	→	*Lint Leakage*	0.24	0.76	1.19
Context Bloat, *Skill Leakage*	→	*Lint Leakage*	0.10	0.75	1.18
Conflicting Instructions, *Lint Leakage*	→	*Context Bloat*	0.07	0.67	1.44

4.5 Co-Occurrence Analysis

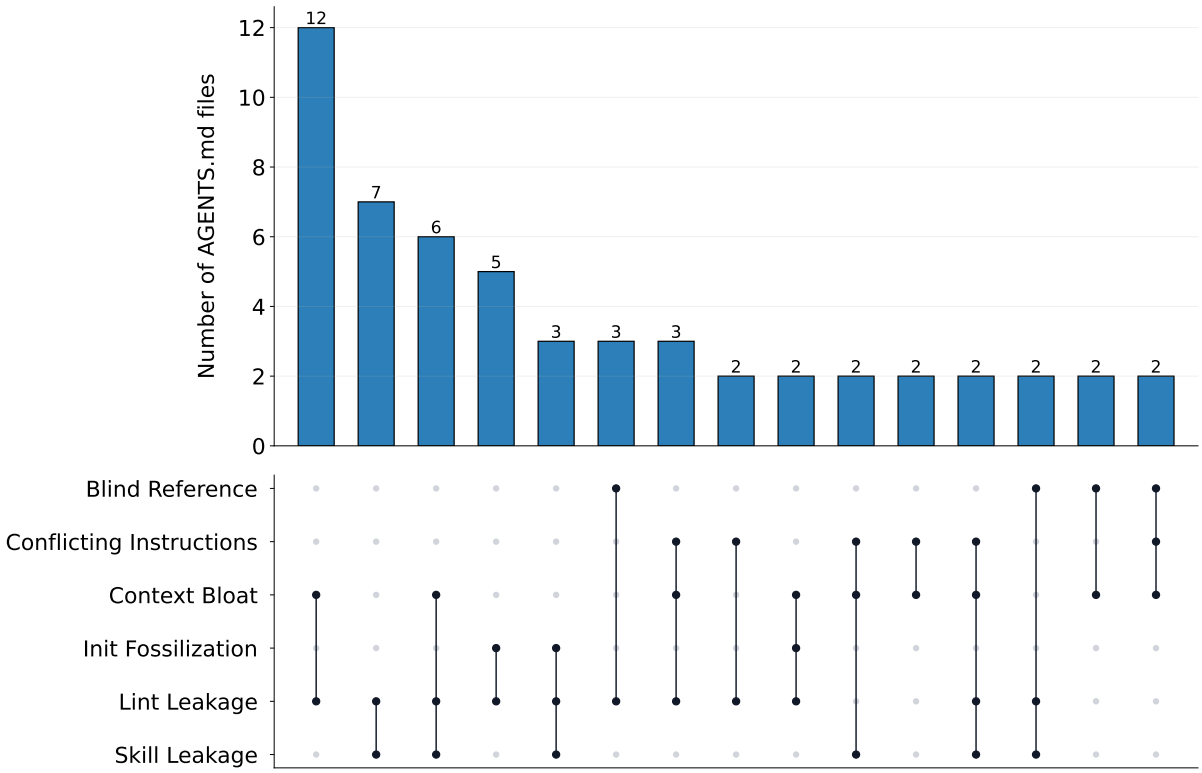
Besides analyzing smells individually, we also investigated which of them tend to co-exist in the same `AGENTS.md` files. To discover such relationships, we used Apriori to mine association rules between the smells detected in our dataset [1]. This procedure is frequently used in software engineering research to associate code smells [14, 26, 29], source code files [40], and bug types [9]. In our case, we mapped each `AGENTS.md` as a transaction record, and the presence of each smell in the file as an item. For instance, a given file F_1 with *Context Bloat* (*CB*), *Skill Leakage* (*SL*), and *Lint Leakage* (*LL*) is represented by the following transaction: $F_1 = \{CB, SL, LL\}$. For the 91 `AGENTS.md` files—our transactions—we applied Apriori with a support of 0.05.

Table 4.4 presents the association rules. The support varies between 0.06 and 0.24, i.e., the associated smells co-exist between 6% and 24% of the files in our dataset. The confidence levels measure the probability of the consequent smell being present given the presence of the antecedent ones. As we can see, the confidence levels are relatively high; for example, the presence of *Conflicting Instructions* and *Skill Leakage* increases the likelihood of *Context Bloat* to 83%. Similar confidence appears in other rules, such as the one associating *Init Fossilization* and *Skill Leakage* with *Lint Leakage* (83%), and *Conflicting Instructions* with *Context Bloat* (81%).

Lift measures how much more likely items are to appear together than by chance. A lift value greater than 1 indicates the smells appear in the same file more often than randomly. We observe a strong association in two rules: *Conflicting Instructions* and *Skill Leakage* with *Context Bloat* (1.81), and *Conflicting Instructions* with *Context Bloat* (1.76). In other words, the probability of *Conflicting Instructions*, *Skill Leakage*, and *Context Bloat* co-occurring in the same file is 1.81 times higher than if they were independent; likewise, *Conflicting Instructions* and *Context Bloat* are 1.76 times more likely to co-occur than by chance.

Figure 4.14 depicts an upset diagram showing the frequency `AGENTS.md` files—bar chart on top—for each unique smell combination—intersection matrix below. In total,

Figure 4.14: Smell accumulation in `AGENTS.md` files.



we detected 15 smell combinations with at least two occurrences. *Context Bloat* and *Lint Leakage* stands out with 12 occurrences; i.e., they appeared together in 12 configuration files. A second group of smells shows up next: (a) *Skill Leakage* and *Lint Leakage* with seven occurrences; (b) *Skill Leakage*, *Lint Leakage*, and *Context Bloat* in six files; and (c) *Lint Leakage* and *Init Fossilization* in five files. The other 11 remaining combinations were detected in three configuration files, at most. Thus, we observe that configuration smells frequently overlap in `AGENTS.md` files, suggesting that multiple issues can compromise the performance of coding agents.

Summary: We leveraged six rules with co-existing smells. *Context Bloat* is strongly associated with *Conflicting Instructions* and *Skill Leakage*, with lift values of 1.81 and 1.76, respectively. This suggests that long `AGENTS.md` files often contain inconsistent or overly specific instructions.

4.6 Threats to Validity

Some steps of this work may be subject to threats to validity. In this section, we highlight and discuss these threats.

Google Search Limitations. In a grey literature review, relevant results might be missed due to the specific combination of search terms. To mitigate this threat, we used keywords to consider both bad and good practices to expand search coverage.

Furthermore, following the guidelines from other works [11, 45, 46], we conducted preliminary search trials, adding and excluding keywords to refine our search strings.

Criteria for Document Selection. We selected the 30 documents returned by Google Search to derive our catalog of smells. Although this relatively small sample may limit the generalizability of our catalog, we manually analyzed all documents returned on the first three pages of the search results, which are the most relevant according to Google's PageRank algorithm. This strategy is consistent with previous grey literature reviews [45, 46]. We also applied a set of sanity checks to ensure the quality of the selected documents, excluding pages behind paywalls, pages focused solely on tools, and superficial content that merely redirected readers to official documentation.

Init Fossilization Heuristic. Our heuristic flags files with a single commit, but it neither checks whether the file was generated by an `/init` command nor whether its content is actually outdated. Conversely, files with only trivial edits, such as typo fixes, are not flagged. Thus, the 24 reported cases indicate files that were never revised.

Bad Practices Annotations. We manually annotated the sentences from the selected documents, which can introduce some bias in the smells identified further. To mitigate this issue, another author performed a sanity check and reviewed all annotations from the selected documents to confirm whether the content accurately represented a good or bad practice in writing configuration files.

Use of LLMs for Code Smell Detection. In this work, we rely on LLMs to evaluate each `AGENTS.md` file and identify the presence of a specific smell. Large Language Models (LLMs) are highly dynamic and frequently updated by their providers. Another key factor is the non-deterministic nature of LLMs, which poses challenges for exact replication. Consequently, the performance and results of code smell detection in configuration files may evolve over time. To mitigate this issue, we set the model temperature to 0 and carefully analyzed the responses to ensure the identification of true positive cases.

4.7 Final Remarks

In this chapter, we presented a comprehensive investigation into the quality problems affecting coding agent configuration files. Through a grey literature review and empirical repository mining, we established the first catalog of six *configuration smells*, along with automated detection heuristics. Our evaluation across 100 open-source projects demonstrated that misconfigurations are pervasive (affecting 91% of files), with *Lint Leakage* and *Context Bloat* representing the most common smells. Furthermore, association rule mining revealed strong co-occurrence dependencies among smells, highlighting that long files frequently accumulate leaked skills and contradictory instructions.

In Chapter 5, we synthesize the findings of both empirical studies, derive practical guidelines for writing and maintaining agent configuration files, and outline directions for future research.

Chapter 5

Conclusion

This chapter concludes the dissertation. Section 5.1 revisits the problem and the two studies conducted to investigate it. Section 5.2 summarizes the findings of each study. Section 5.3 derives practical guidelines for developers who write and maintain agent configuration files. Finally, Section 5.4 outlines directions for future research.

5.1 Overview

Autonomous coding agents have moved from research prototypes to tools used in real development workflows. As agents take on tasks that span multiple files and require knowledge of project conventions, developers need a way to tell them what the source code alone does not reveal: which architecture to follow, which commands to run, and which practices the team expects. Agent configuration files, such as `AGENTS.md` and `CLAUDE.md`, have become the standard answer to this need.

This dissertation investigated these artifacts through two empirical studies. The first, presented in Chapter 3, characterized how developers write them, analyzing 328 `CLAUDE.md` files from the most popular Claude Code projects on GitHub. The second, presented in Chapter 4, examined their quality, introducing a catalog of six *configuration smells*, automated heuristics to detect them, and an evaluation of their prevalence across 100 popular open-source repositories.

5.2 Summary of Findings

This section synthesizes the results of the two studies. Section 5.2.1 presents the findings on the anatomy of configuration files, and Section 5.2.2 presents the findings on

their quality.

5.2.1 Anatomy of Agent Configuration Files

Our first study shows that developers use configuration files to record classical software engineering concerns. Architecture is the dominant one, appearing in 72.6% of the analyzed files, followed by general development guidelines (44.8%), project overviews (39%), testing guidelines (35.4%), and commands (33.2%). These are concerns that go beyond the immediate task, and that the source code alone does not make explicit.

Code examples are concentrated in a few categories, reaching 17.68% of the sections classified as Development Guidelines, while links appear in only 1.83% of the files and diagrams are almost absent, with two instances in the entire dataset. Developers therefore describe conventions in words far more often than they demonstrate them.

Finally, configuration files are not organized arbitrarily. By mining co-occurrence patterns, we found that a small number of combinations account for how these files are structured. The most frequent one groups rules about Architecture, Dependencies, and Project Overview, present in 21.6% of the files, and Architecture participates in all five most frequent patterns. These patterns can serve as reference models for developers writing a configuration file from scratch.

5.2.2 Configuration Smells

Our second study shows that the quality of these files is uneven. Based on a grey literature review of 14 documents, we identified six recurring problems—*Context Bloat*, *Skill Leakage*, *Lint Leakage*, *Blind Reference*, *Init Fossilization*, and *Conflicting Instructions*—and proposed heuristics to detect them. Applied to 100 popular repositories, these heuristics found at least one smell in 91 of the analyzed files. The counts reported below refer to instances confirmed by manual inspection, after discarding the false positives listed in Chapter 4.

Lint Leakage is the most frequent, with 58 occurrences: developers restate in natural language formatting rules that a linter or formatter already enforces deterministically. *Context Bloat* follows, with 42 occurrences, and *Skill Leakage* with 29. *Init Fossilization* appears 24 times, *Conflicting Instructions* 16 times, and *Blind Reference* 14 times. The

gap between detected and confirmed instances is widest for *Conflicting Instructions*, whose heuristic reached 57% precision, since identifying contradictory rules requires deeper semantic reasoning than detecting redundancy. Four of the six smells—*Lint Leakage*, *Context Bloat*, *Skill Leakage*, and *Init Fossilization*—concern content that is present but no longer useful to the agent, rather than content that is missing.

Association rule mining further shows that these smells do not occur in isolation. The joint presence of *Conflicting Instructions* and *Skill Leakage* raises the probability of *Context Bloat* to 83%, and similar levels of confidence appear in other rules. Long files, in other words, often contain inconsistent or overly specific instructions.

This study also addressed a question left open by the first one. Analyzing 383 pull requests that modify configuration files, we found no quality problems beyond the six already cataloged, which reinforces the comprehensiveness of the catalog.

5.3 Guidelines for Writing Agent Configuration Files

The catalog proposed in this dissertation describes what to avoid. Stated positively, our findings support the following guidelines for developers who write and maintain agent configuration files.

Let deterministic tools do their job. Formatting rules, import ordering, and line limits belong in a linter or formatter configuration, not in a natural language manifest. The agent runs these tools; it does not need to be told what they already enforce. This was the most frequent smell in our evaluation, with 62 occurrences.

Treat the configuration file as code that evolves. The file produced by an `/init` command is a draft, not a final version. Projects change their architecture, their commands, and their conventions; we detected 24 files that were generated by `/init` and never revised afterwards. Reviewing the configuration file should be part of the same routine that keeps documentation and dependencies up to date.

Give every reference a purpose. Pointing the agent to a document is not enough. A reference should state what the document contains and when it should be consulted; otherwise, the agent either loads it unnecessarily or ignores it altogether. The same

applies to references to files that no longer exist, which consume context without providing information.

Keep the file short, and treat length as a symptom. Every rule written in a configuration file is re-injected into the model's context at the start of each session, and competes for attention with every other rule. Our co-occurrence results indicate that long files tend to accumulate contradictions and overly specific instructions. A file that keeps growing is therefore a good candidate for review, since part of what it contains may no longer apply.

5.4 Future Work

The results presented in this dissertation open several directions for further investigation.

Measuring behavioral impact. Our studies characterize configuration files and catalog their quality problems, but they do not measure how these problems affect the output of a coding agent. Future work could design controlled experiments comparing agent behavior on the same task with and without a given smell, in order to quantify the cost of each violation in terms of task success, token consumption, and adherence to project conventions. Such evidence would also allow the smells to be ranked by severity, which our catalog currently does not do.

Assessing the impact of instructions positioning on agent effectiveness. As discussed in Section 2.1.2, language models tend to pay less attention to content placed in the middle of long contexts. Future work could investigate whether the position of a section in a configuration file affects how often the agent follows its instructions, using the section's location as a proxy for the model's loss of attention.

Extending the catalog. The six smells cataloged in this dissertation emerged from a grey literature review and are not claimed to be exhaustive. Other candidates deserve investigation: configuration files that are too complex for humans to read and review, files that fall out of sync with the repository they describe—of which *Init Fossilization* is arguably a special case—problems related to how agents discover and load these files, and the absence of modularity in large monorepos, where a single flat file is used instead

of nested, scoped ones. Establishing whether these are distinct smells or manifestations of the ones we already describe would clarify the boundaries of the catalog.

Configuration files that govern their own maintenance. These files are frequently created and edited by the agent itself, through commands such as `/init` and through subsequent requests to update them. A possible use of our catalog is to embed it in the manifest, as a section stating the practices to avoid when the agent modifies the file. Whether this reduces the incidence of smells is an open question, and evaluating it would require the controlled experiments described above.

Tool support. The heuristics proposed in Chapter 4 were designed for empirical analysis, but they can be packaged as a linter for agent configuration files, integrated into pre-commit hooks, continuous integration pipelines, or editors. Such a tool could report smells as the file is edited and, in the case of *Lint Leakage* and *Blind Reference*, suggest concrete corrections.

Evolution of configuration files. Both of our datasets are snapshots. A longitudinal study following the commit history of these files would reveal when rules are added, how long obsolete instructions survive, and whether smells are introduced at creation time or accumulate through maintenance, in the spirit of what Tufano et al. [42] did for traditional code smells.

Broader coverage of tools and ecosystems. Our analysis focused on `CLAUDE.md` and `AGENTS.md` files from popular repositories. Other agents adopt their own conventions, and less popular projects may configure their agents differently. Extending the analysis to tools such as OpenAI Codex, GitHub Copilot, and Cursor, and to repositories outside the popularity range we studied, would establish how far our findings generalize.

Developer perception. Finally, our catalog reflects the practices reported in grey literature and the patterns we observed in repositories, but we did not ask maintainers whether they agree that these are problems. Surveying developers, as Palomba et al. [28] did for code smells and Rosa et al. [33] did for Dockerfile smells, would reveal which violations practitioners consider consequential and which they deliberately accept.

References

[1] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. *Advances in Knowledge Discovery and Data Mining*, 12(1):307–328, 1996.

[2] Aylton Almeida, Laerte Xavier, and Marco Tulio Valente. Using Copilot Agent Mode to Automate Library Migration: A Quantitative Assessment. In *1st International Workshop on Agentic Engineering*, 2026.

[3] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated Unit Test Improvement Using Large Language Models at Meta. In *32nd ACM Symposium on the Foundations of Software Engineering (FSE)*, 2024.

[4] Hudson Borges and Marco Tulio Valente. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software*, 146:112–129, 2018.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:1877–1901, 2020.

[6] Worawalan Chatlatanagulchai, Kundjanasith Thonglek, Brittany Reid, Yutaro Kashiwa, Pattara Leelaprute, Arnon Rungsawang, Bundit Manaskasemsak, and Hajimu Iida. On the Use of Agentic Coding Manifests: An Empirical Study of Claude Code. In *24th International Conference on Product-Focused Software Process Improvement (PROFES)*, volume 16361 of *Lecture Notes in Computer Science*. Springer, 2025. doi: 10.1007/978-3-032-12089-2_40.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[8] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. A Systematic Literature Review on Bad Smells–5 W's: Which, When, What, Who, Where. *IEEE Transactions on Software Engineering*, 2021.

[9] Taijara Loiola De Santana, Paulo Anselmo Da Mota Silveira Neto, Eduardo Santana De Almeida, and Iftekhar Ahmed. Bug Analysis in Jupyter Notebook Projects: An Empirical Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2024.

[10] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. Large Language Models for Software Engineering: Survey and Open Problems. In *IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, 2023.

[11] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 2019.

[12] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Workshop on Frequent Itemset Mining Implementations (FIMI)*, volume 90, page 65, 2003.

[13] Ian Guelman, Arthur Gregorio Leal, Laerte Xavier, and Marco Tulio Valente. On the Quality of AI-Generated Source Code Comments: A Comprehensive Evaluation. In *1st International Workshop on AI for Software Quality Evaluation (AI-SQE)*, 2026.

[14] Oumayma Hamdi, Ali Ouni, Eman Abdullah AlOmar, and Mohamed Wiem Mkaouer. An Empirical Study on Code Smells Co-occurrences in Android Applications. In *36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, 2021.

[15] Ahmed E. Hassan, Gustavo A. Oliva, Dayi Lin, Boyuan Chen, Zhen Ming, and Jiang. Towards ai-native software engineering (se 3.0): A vision and a challenge roadmap, 2024.

[16] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology*, 2024.

[17] Chip Huyen. *AI Engineering: Building Applications with Foundation Models*. O'Reilly, 2025.

[18] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *12th International Conference on Learning Representations (ICLR)*, 2024.

[19] Stefano Lambiase, Gemma Catolino, Fabio Palomba, and Filomena Ferrucci. Motivations, Challenges, Best Practices, and Benefits for Bots and Conversational Agents in Software Engineering: A Multivocal Literature Review. *ACM Computing Surveys*, 57, December 2024.

[20] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes*, 30(5):296–305, 2005.

[21] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. In *IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2023.

[22] Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. Using Transfer Learning for Code-Related Tasks. *IEEE Transactions on Software Engineering*, 2023.

[23] Seyedmoein Mohsenimofidi, Matthias Galster, Christoph Treude, and Sebastian Baltes. Context engineering for ai agents in open-source software. *arXiv preprint arXiv:2510.21413*, 2025.

[24] Manishankar Mondal, Banani Roy, Chanchal K Roy, and Kevin A Schneider. Associating code clones with association rules for change impact analysis. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 93–103, 2020.

[25] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. GitHub Copilot AI pair programmer: Asset or Liability? *Journal of Systems and Software*, 203:111734, 2023.

[26] Biruk Asmare Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. On the Prevalence, Impact, and Evolution of SQL Code Smells in Data-Intensive Systems. In *17th International Conference on Mining Software Repositories (MSR)*, 2020.

[27] Daye Nam, Andrew Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. Using an LLM to Help With Code Understanding. In *46th International Conference on Software Engineering (ICSE)*, 2024.

[28] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *International Conference on Software Maintenance and Evolution (ICSME)*, 2014.

[29] Fabio Palomba, Rocco Oliveto, and Andrea De Lucia. Investigating Code Smell Co-Occurrences Using Association Rule Learning: A Replicated Study. In *IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, 2017.

[30] Gustavo Pinto, Cleidson R. B. de Souza, João Batista Cordeiro Neto, Alberto de Souza, Tarcísio Gotto, and Edward Monteiro. Lessons from Building CodeBuddy: A Contextualized AI Coding Assistant. In *46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2024.

[31] Kaiser Pister, Dhruba Jyoti Paul, Patrick Brophy, and Ishan Joshi. PromptSet: A Programmer's Prompting Dataset. In *Proceedings of the 1st ACM International Conference on AI-Powered Software (AIware)*, 2024.

[32] Romain Robbes, Théo Matricon, Thomas Degueule, Andre Hora, and Stefano Zacchiroli. Agentic Much? Adoption of Coding Agents on GitHub. *ACM Transactions on Software Engineering and Methodology*, 2026.

[33] Giovanni Rosa, Federico Zappone, Simone Scalabrino, and Rocco Oliveto. Fixing Dockerfile Smells: An Empirical Study. *Empirical Software Engineering*, 2024.

[34] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[35] Helio Victor F. Santos, Vitor Costa, Joao Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. Configuration smells in AGENTS.md files: Common mistakes in configuring coding agents. In *26th IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, page 1–12, 2026.

[36] Yuya Sasaki, Hironori Washizaki, Jialong Li, Nobukazu Yoshioka, Naoyasu Ubayashi, and Yoshiaki Fukazawa. Landscape and Taxonomy of Prompt Engineering Patterns in Software Engineering. *IT Professional*, 2025.

[37] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. Prompt Engineering or Fine Tuning: An Empirical Assessment of Large Language Models in Automated Software Engineering Tasks. *ArXiv*, 2023.

[38] Mohammed Latif Siddiq, Joanna C. S. Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes. Using Large Language Models to Generate JUnit Tests: An Empirical Study. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, 2024.

[39] Luciana Lourdes Silva, Janio Rosa Da Silva, Joao Eduardo Montandon, Marcus Andrade, and Marco Tulio Valente. Detecting Code Smells Using ChatGPT: Initial Insights. In *18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 400–406, October 2024.

[40] Mauricio Soto and Claire Le Goues. Using a Probabilistic Model to Predict Bug Fixes. In *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018.

[41] Davide Taibi and Valentina Lenarduzzi. On the Definition of Microservice Bad Smells. *IEEE Software*, 2018.

[42] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and Why Your Code Starts to Smell Bad. In *37th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2015.

[43] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabic, Massimiliano Di Penta, and Gabriele Bavota. Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study. In *21st International Conference on Mining Software Repositories (MSR)*, pages 571–583, 2024.

[44] Francesco Urdih, Theodoros Theodoropoulos, and Uwe Zdun. Cache-Related Smells in GitLab CI/CD: Comprehensive Catalog, Automated Detection, and Empirical Evidence. *arXiv preprint arXiv:2604.17890*, 2026.

[45] Lucas Vegi and Marco Tulio Valente. Code Smells in Elixir: Early Results from a Grey Literature Review. In *30th International Conference on Program Comprehension (ICPC)*, 2022.

[46] Lucas Vegi and Marco Tulio Valente. Understanding Code Smells in Elixir Functional Language. *Empirical Software Engineering*, page 1–36, 2023.

[47] Markos Viggiato, Johnatan Oliveira, Eduardo Figueiredo, Pooyan Jamshidi, and Christian Kästner. How do code changes evolve in different platforms? A mining-based investigation. In *35th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 218–222, 2019.

[48] Miku Watanabe, Yutaro Kashiwa, Bin Lin, Toshiki Hirao, Ken'Ichi Yamaguchi, and Hajimu Iida. 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)*, pages 375–380, 2024.

[49] Miku Watanabe, Hao Li, Yutaro Kashiwa, Brittany Reid, Hajimu Iida, and Ahmed E. Hassan. On the Use of Agentic Coding: An Empirical Study of Pull Requests on GitHub, 2025.

[50] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[51] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, Qi Zhang, and Tao Gui. The rise and potential of large language model based agents: a survey. *Science China Information Sciences*, 2025.

[52] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based Software Engineering Agents, 2024.

[53] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering, 2024.

[54] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *11th International Conference on Learning Representations (ICLR)*, 2023.