

Harness Engineering for Agentic AI Coding Tools: An Exploratory Study

Matthias Galster

University of Bamberg
Bamberg, Germany
mgalster@ieee.org

Syedmoein Mohsenimofidi

Heidelberg University
Heidelberg, Germany
s.mohsenimofidi@uni-heidelberg.de

Jai Lal Lulla

Singapore Management University
Singapore, Singapore
jailal.l.2025@phdcs.smu.edu.sg

Muhammad Auwal Abubakar

University of Bamberg
Bamberg, Germany
muhammad.abubakar@uni-bamberg.de

Christoph Treude

Singapore Management University
Singapore, Singapore
ctreude@smu.edu.sg

Sebastian Baltes

Heidelberg University
Heidelberg, Germany
sebastian.baltes@uni-heidelberg.de

Abstract

Agentic AI coding tools increasingly automate software development tasks. Developers can configure these tools through versioned repository-level artifacts such as Markdown and JSON files. The emerging term *harness engineering* extends context engineering, broadening the focus from a model's context to the full set of mechanisms configured around it. We present a systematic analysis of configuration mechanisms for agentic AI coding tools, covering Claude Code, GitHub Copilot, Cursor, Gemini, and Codex. We identify eight configuration mechanisms spanning from static context to executable and external integrations and, in an empirical study of 2,853 GitHub repositories, examine whether and how they are adopted, with a detailed analysis of CONTEXT FILES, SKILLS, and SUB-AGENTS. First, CONTEXT FILES dominate the configuration landscape and are often the sole mechanism in a repository, with AGENTS.md emerging as an interoperable standard across tools. Second, few repositories adopt advanced mechanisms such as SKILLS and SUB-AGENTS. SKILLS predominantly rely on static instructions rather than executable scripts. Third, distinct configuration practices are forming around different tools, with Claude Code users employing the broadest range of mechanisms. These findings establish an empirical baseline for understanding harness engineering, suggest that AGENTS.md serves as a natural starting point, and motivate longitudinal and experimental research on how configuration strategies evolve and affect agent performance.

CCS Concepts

• Software and its engineering;

Keywords

Software Engineering, Generative AI, AI Agents, Configuration

*This article is an extended version of *Configuring Agentic AI Coding Tools: An Exploratory Study*, published at AIware 2026 [10].



This work is licensed under a Creative Commons Attribution 4.0 International License. AIware '26, Montreal, QC, Canada

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2601-9/2026/07
<https://doi.org/10.1145/3805760.3814887>

1 Introduction

Agentic AI coding tools based on large language models (LLMs) [28, 30], automate time-consuming and repetitive tasks, such as generating and editing code, tests, and documentation. Unlike purely reactive conversational assistants, these tools proactively accomplish defined objectives [3] by autonomously interacting with development environments, project artifacts, external data, and command-line tools with reduced human interaction [32]. Certain functionality can be delegated to other *tools* and *agents*. An *agent* is a goal-directed component that interprets a user goal, decomposes it into substeps, selects and executes tools, and iteratively adjusts its plan in an *agent loop*. *Tools* are deterministic capabilities with a specific, bounded function (e.g., a Bash tool to run shell commands¹), invoked by the underlying model in an *agent loop*. We distinguish such fine-grained tools from broader *agentic AI (coding) tools* such as Claude Code or OpenAI Codex.

Initially, these tools implemented one central agent loop steered by a foundation model. Conversational tools such as GitHub Copilot and Cursor soon offered similar capabilities via an *agent mode*. More recently, tool vendors introduced extension and configuration mechanisms to customize tool behavior, one of which allows developers to define their own *subagents* that operate in parallel to the central agent loop, in their own context.

We use *context* to denote the complete input to a single model call. The software layers around the model(s) that drive the agent loop (i.e., the agent *harness*²) assemble this context for each call, expose tool schemas, and manage turn-by-turn state. *Context engineering* [18] is the practice of designing the context that the harness assembles at runtime. Through various configuration mechanisms, developers customize the harness for a given project, and thereby the context the model sees on each call. This customization of the harness, not only the context, is *harness engineering*.³

We define a *configuration mechanism* as a means for developers to tailor tool and agent behavior to a project or workflow (e.g., context files or dedicated subagent definitions). A *configuration artifact* is a tangible instance of a mechanism: either a single configuration file (e.g., CLAUDE.md, or one subagent file such as .claude/agents/reviewer.md) or a directory bundling several configuration files

¹<https://code.claude.com/docs/en/tools-reference>

²<https://magazine.sebastianraschka.com/p/components-of-a-coding-agent>

³<https://martinfowler.com/articles/harness-engineering.html>

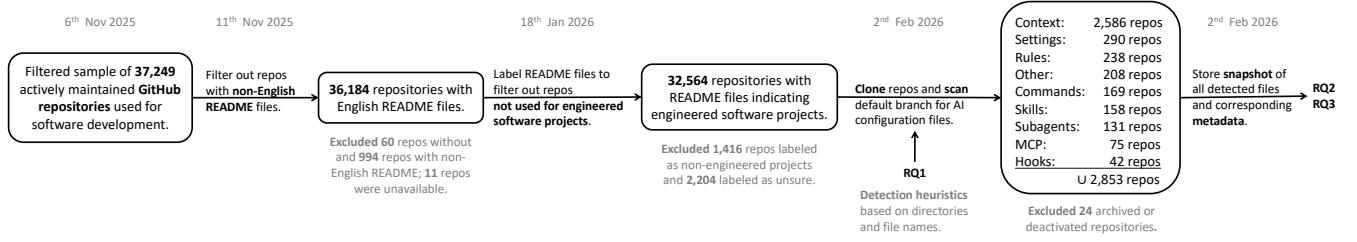


Figure 1: Data collection process.

that together define one artifact (e.g., a skill whose directory contains `SKILL.md` alongside scripts, references, and assets). Context files such as `AGENTS.md` or `CLAUDE.md` act as “READMEs for agents” with context-specific information about build commands, coding conventions, and rules for CI/CD pipelines [19, 21]. These configuration artifacts can be version-controlled, making them inspectable and collaboratively maintainable. With the increased availability and diversity of agentic tools, the number of available configuration mechanisms and related artifacts has increased.

To understand which configuration mechanisms agentic tools offer and how they are used in open-source software (OSS), we address the following **research questions**:

RQ1 What configuration mechanisms do agentic coding tools offer?

RQ2 Which mechanisms are adopted in OSS repositories?

RQ3 How are configuration mechanisms adopted?

We restrict our scope to configuration mechanisms whose artifacts are consumed by agentic tools and are repository-versioned.

With this paper, we provide the following **contributions**: (1) We systematically document eight configuration mechanisms: CONTEXT FILES, SKILLS, SUBAGENTS, COMMANDS, RULES, SETTINGS, HOOKS, and MCP servers. We identified these from the documentation of Claude Code, GitHub Copilot, Cursor CLI, Gemini CLI, and Codex CLI. Some mechanisms are available in all tools; others are specific to particular tools. (2) We analyzed the adoption of these configuration mechanisms in 2,853 GitHub repositories. CONTEXT FILES (Markdown files that provide contextual project information) dominated and are often the sole configuration mechanism. Claude Code users apply the broadest range of configuration mechanisms. (3) We analyzed the adoption of CONTEXT FILES, SKILLS, and SUBAGENTS in more detail. Three CONTEXT FILE formats showed the most dynamic development, with `AGENTS.md` emerging as a standard. Although SKILLS and SUBAGENTS can be defined for a wide range of purposes, most repositories that adopt them define only one or two artifacts. Moreover, to extend agent behavior, SKILLS primarily rely on static resources rather than executable scripts.

2 Related Work

Prior work on context engineering has focused on single artifact types in isolation, without systematic examination of the broader configuration mechanisms. Context engineering builds on earlier prompt engineering techniques [9, 24] but broadens the scope from individual prompts to the systematic design of all information provided to the model. Mei et al. [18] surveyed context retrieval,

processing, and management techniques for LLMs, while practitioner accounts demonstrate how structuring project-specific context improves agent effectiveness in complex codebases [13, 26]. In a recent review of LLM agents, Zhou et al. [36] traced a progression from model weights to context to the harness, naming the last step *harness engineering*. Empirical work has mainly focused on repository-maintained context files, such as `AGENTS.md`. Mohsenimofidi et al. [19] showed that repository-level context files describe project architecture, build commands, and contribution conventions; Chatlatanagulchai et al. [6] additionally found that they are actively maintained through small frequent changes. Santos et al. [25] analyzed 328 `CLAUDE.md` files from Claude Code projects and found that architectural concerns dominate, followed by development guidelines, project overviews, and testing.

Beyond structural characterization, Lulla et al. [16] ran a controlled study with and without an `AGENTS.md` file, reporting lower runtime and token consumption at comparable task completion. Villamizar et al. [31] argued that prompts should be treated as software engineering artifacts and outlined a research agenda for their evolution, reuse, and governance. Q. Zhang et al. [34] proposed a framework treating contexts as evolving instructions, addressing brevity bias and context collapse through incremental updates.

Other work formalizes context as a programmable abstraction. Y. Zhang et al. [35] introduced Monadic Context Engineering, modeling context construction and transformation through monadic composition. McMillan [17] evaluated structured file-native context schemas and found that a multi-file organization and retrieval strategy affects agent accuracy more than the choice of serialization format. Ye et al. [33] proposed Meta Context Engineering, a framework in which agentic skills and context artifacts co-evolve via evolutionary search. Jiang and Nam [14] explored Cursor rules and developed a taxonomy of their content.

Robbes et al. [22] measured coding-agent adoption on GitHub through file-, commit-, and pull-request-level heuristics in 128,018 repositories. Their categorization distinguishes structured *configuration files* (such as `YAML`) from natural-language *rules and guidance files* (including `CLAUDE.md` and `AGENTS.md`). This is a coarser split than our eight-mechanism taxonomy. Their sample applies activity and size thresholds without an engineered-project classification step, and their analysis centers on adoption rates and AI-assisted commit characteristics rather than on the internal structure and co-occurrence of configuration mechanisms.

However, no prior work maps configuration mechanisms across tools or studies their adoption patterns across repositories. We provide this cross-tool perspective.

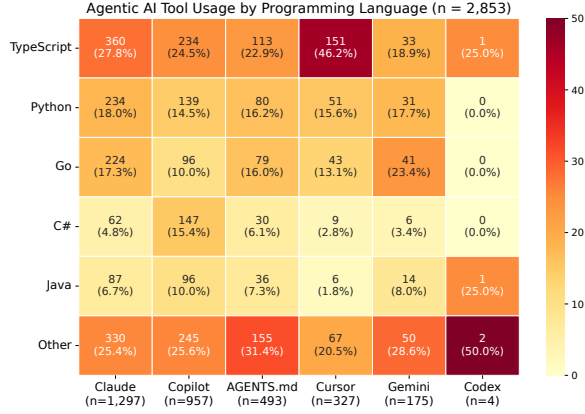


Figure 2: Adoption of agentic tools per programming language; AGENTS.md denotes repositories using only that file with no tool-specific configuration. Percentages relative to repository count per tool. Column totals can exceed the overall repository count (multiple tools per repository).

3 Data Collection and Analysis

We collected OSS projects hosted on GitHub, selecting repositories belonging to “engineered” software projects [20] using an updated version of a selection approach from prior work [19]. The repository selection started with the SEART GitHub search tool [8, 27]. We applied several filters, each addressing a specific sampling risk [15]. We selected non-fork repositories with at least two contributors and a license, because forks duplicate codebases, the presence of a license signals the author’s intent that the project be reused, and single-contributor projects often represent personal experiments rather than collaborative software. The creation cutoff of 1 January 2024 ensures that each repository predates the broad availability of agentic AI coding tools by more than a year, so detected configuration mechanisms represent additions to established projects rather than greenfield experiments built around the tools. The activity cutoff of commits since 1 June 2025 removes dormant projects and keeps repositories that were actively maintained during the period when agentic tools became widely available. We excluded archived, disabled, or locked repositories for the same reason. We applied a licensing filter that retained only OSI-approved software licenses and a popularity-based language filter, selecting the ten most common primary languages (Python, TypeScript, JavaScript, Go, Java, C++, Rust, PHP, C#, and C). We further excluded repositories with fewer than 271 commits or fewer than 7 watchers, the median values of the sampling frame following [Mohsenimofidi et al. \[19\]](#), filtering out low-activity or poorly-followed repositories. These filters resulted in a sample of 37,249 repositories.

Figure 1 outlines the data collection pipeline for this sample. We cloned the repositories and searched their default branch for a README file, excluding 11 that became unavailable and 60 without such a file. We then used the `lingua-language-detector` Python library to detect each file’s language, excluding 994 repositories with non-English README files.

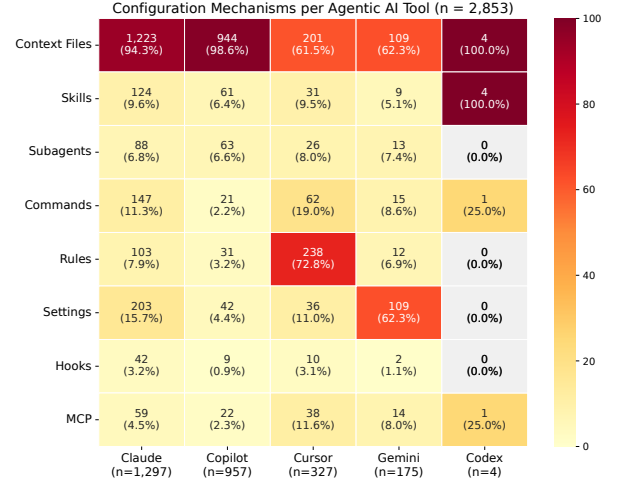


Figure 3: Usage of configuration mechanisms across agentic tools. Repositories using only AGENTS.md without tool-specific configuration are excluded. Percentages relative to repository count per tool. Column totals can exceed the overall repository count (multiple tools per repository).

For the remaining 36,184 repositories, we used a classification pipeline to determine, based on README content, whether each project qualifies as “engineered”, that is, it shows clear evidence of software engineering practices [20]. We operationalized this, among other criteria, as having a clear project purpose and documenting practices such as building, testing, and maintenance. We used the OpenAI GPT-5.2 model for classification. Our supplementary material includes the complete prompt, which we iteratively developed and tested on subsets of the sample, and the model configuration (we used the OpenAI defaults) [11]. The pipeline labeled 32,564 repositories as engineered and excluded the remaining 3,620 (1,416 classified as non-engineered, 2,204 ‘unsure’). We spot-checked randomly selected repositories from each category during prompt development and for the final sample. We then cloned the remaining 32,564 repositories and applied heuristics based on file names and file paths to detect the usage of AI coding tools and configuration mechanisms (see Table 1). These heuristics are based on the answers to **RQ1** and are briefly discussed in Section 4. This resulted in 2,853 repositories that use one or more AI coding tools; three repositories contained only empty configuration artifacts and were excluded. Our data collection and analysis scripts and the analyzed data are available online [11].

4 Configuration Mechanisms (RQ1)

We selected five agentic AI coding tools (Table 2) based on the *2025 Stack Overflow Developer Survey* [29]: the four most popular AI tools among the surveyed developers, with Codex substituting for ChatGPT (the agentic tool from the same vendor), plus Cursor, which has recently been studied in software engineering research [12, 14]. Our study focuses on the CLI-based agentic interfaces that first appeared in 2025, although prior non-agentic versions exist for some tools. One author systematically reviewed each tool’s online

Table 1: Overview of repository-level configuration mechanisms across agentic AI coding tools. Each cell lists the repository file(s) or directory implementing that mechanism; “–” indicates that the mechanism was not available when we checked.

Mechanism	Description	Claude	Copilot	Codex	Cursor	Gemini
CONTEXT FILES	Markdown file loaded into the context each session.	CLAUDE.md	.github/ { copilot-instructions.md instructions/*.md } ^a	AGENTS.md, AGENTS.override.md	AGENTS.md, .cursorrules ^c	GEMINI.md
SETTINGS	JSON/TOML config for project-level tool behavior.	.claude/settings (local)?.json	– ^b	.codex/ config.toml	.cursor/ cli.json	.gemini/{ settings.json config.yaml }
SKILLS	Reusable knowledge and invocable workflows.	.claude/skills/	.github/skills/	.codex/skills/	.cursor/skills/	.gemini/skills/
SUBAGENTS	Specialized agents that operate in parallel to the central agent loop, in their own context.	.claude/agents/	.github/agents/	–	.cursor/agents/	–
COMMANDS	User-triggered shortcuts for predefined prompts.	.claude/ commands/	–	–	.cursor/ commands/	.gemini/ commands/
HOOKS	Scripts executed at specific agent lifecycle points.	.claude/ settings.json	.github/hooks/*.json	–	.cursor/ hooks.json	.gemini/ settings.json
RULES	System-level instructions to control agent behavior.	–	–	.codex/rules/	.cursor/rules/	–
MCP	External tool or data connections via the Model Context Protocol.	.mcp.json	– ^b	.codex/ config.toml	.cursor/ mcp.json	.gemini/ settings.json

^a Copilot also supports CLAUDE.md, AGENTS.md, and GEMINI.md; ^b Configured via the GitHub web UI, not via files in the project repository;

^c Cursor deprecated .cursorrules and now suggests using AGENTS.md instead.

Table 2: Release dates of agentic AI coding tools and repositories ($n = 2,853$; one repository can use multiple tools).

Agentic Tool	Release (Month/Year)	#Repositories
Claude Code	02/2025 (CLI & agents since release)	1,297
GitHub Copilot	10/2021 (Release) 02/2025 (Copilot Agent Mode) 09/2025 (Copilot CLI)	957
AGENTS.md ^d	05/2025 (Adoption in Codex) 08/2025 (Specification [2])	493
Cursor CLI	03/2023 (Release) 06/2025 (Cursor Agents) 08/2025 (Cursor CLI)	327
Gemini CLI	02/2024 (Release) 05/2025 (Gemini Agent Mode) 06/2025 (Gemini CLI)	175
Codex CLI	04/2025 (CLI & agents since release)	4

^d Repositories using AGENTS.md without any tool-specific configuration artifact.

documentation, documenting configuration mechanisms together with repository-level files and directories that indicate their usage. Two other authors then cross-checked the extracted mechanisms and the heuristics developed to detect them, and all three authors discussed and reconciled the results.

Table 1 summarizes these heuristics. For example, a .claude directory or a CLAUDE.md file indicates Claude Code usage; a .claude/agents/ directory with Markdown files indicates SUB-AGENTS. AGENTS.md is a special case: as a tool-agnostic standard, it is supported by multiple tools, so repositories using only AGENTS.md

without tool-specific artifacts are tracked separately (see Table 2, footnote ^d). Note that for Copilot, Cursor, and Gemini, certain detected files apply to both conversational and agentic workflows. Our documentation of the mechanisms and detection heuristics (with links to the relevant tool documentation) and the Python scripts that implement our matching strategy are part of the supplementary material [11].

RQ1 (Summary):

- We identified eight configuration mechanisms spanning from static context (e.g., CONTEXT FILES) to executable and external integrations (e.g., SKILLS, MCP). Two mechanisms (CONTEXT FILES and SKILLS) are supported by all five tools.
- Despite this convergence, no single tool implements all eight mechanisms.

5 Adoption of Configuration Mechanisms (RQ2)

To understand how developers adopt configuration mechanisms in agentic AI coding tools, we analyzed the presence and co-occurrence of configuration artifacts across repositories.

5.1 Characterization of Repositories in Dataset

Before analyzing adoption, we characterize the 2,853 repositories in our dataset to contextualize the subsequent analysis. Of the 2,853 repositories, 2,015 (70.6%) adopted a single tool, while 295 (10.3%) configured two tools, and 50 (1.8%) configured three or more. An additional 493 repositories (17.3%) used AGENTS.md as a tool-agnostic standard without any tool-specific configuration artifact;

these are shown separately in the analysis (see Table 2, footnote ^d). Among repositories with multiple tools, Claude appeared most frequently with others. The most common combination was Claude and Copilot ($n = 167$), followed by Claude and Cursor ($n = 144$), Claude and Gemini ($n = 52$), and Copilot and Cursor ($n = 49$). Notably, 44.0% of Cursor repositories also configured Claude. The dominance of single-tool repositories limits the potential for multi-tool confounding in our tool-specific analyses. In our initial sample, the top five primary programming languages per repository ($n = 36,184$) were Python (8,133; 22.5%); TypeScript (4,999; 13.8%); Java (3,813; 10.5%); Go (3,786; 10.5%); and JavaScript (3,709; 10.3%). Interestingly, this order is slightly different for repositories that use agentic coding tools ($n = 2,853$, see Figure 2) where TypeScript and Go were more prominent and C# replaces JavaScript in the top five: TypeScript (736, 25.8%); Python (479, 16.8%); Go (420, 14.7%); C# (228, 8.0%); Java (223, 7.8%).

TypeScript was the most common primary language for repositories using Claude, Copilot, and Cursor. Usage in Java and C# repositories was lower for all tools but Copilot. As shown in Table 3, which compares each tool's adopting repositories against the complement of non-adopting repositories, repositories using Cursor were younger than those using other tools. Excluding Codex ($n=4$), repositories associated with Gemini had larger contributor counts and commit volumes than those associated with other tools. Cursor repositories were the largest by source code size (excluding Codex). AGENTS.md repositories tended to have fewer commits and smaller size than the overall sample, suggesting that these repositories adopted AGENTS.md as a lightweight, tool-agnostic starting point.

5.2 Distribution of Configuration Mechanisms

Figure 3 shows the distribution of configuration mechanisms per tool. CONTEXT FILES (e.g., CLAUDE.md, AGENTS.md) were the most frequently adopted mechanism, used by 61.5 to 100% of repositories across all tools. Two tool-specific patterns stand out: 72.8% of Cursor repositories adopt RULES, since Cursor was one of the first to introduce this mechanism [14], and 62.3% of Gemini repositories use SETTINGS. No other mechanism exceeds 20% adoption for Claude, Copilot, Cursor, or Gemini; Codex ($n=4$) is too small to characterize reliably.

Most repositories included only a single CONTEXT FILE artifact, and the adoption of multiple non-context file mechanisms within the same repository remained rare. These results suggest that most repositories rely on CONTEXT FILES as their baseline configuration.

5.2.1 Co-occurrence of Configuration Mechanisms. We quantified pairwise associations between the adoption of configuration mechanisms using Cramér's V from chi-squared tests (28 comparisons, Benjamini–Hochberg corrected). Several configuration mechanisms are frequently adopted together, and some of these associations can be explained by shared configuration artifacts. For example, SETTINGS and HOOKS show a positive association ($V = 0.36$), because hooks are usually defined in SETTINGS files. SETTINGS and HOOKS also both co-occur with SKILLS ($V = 0.15$ and $V = 0.24$, respectively), which are defined and configured differently. SUBAGENTS also show positive associations with multiple mechanisms, most strongly with COMMANDS ($V = 0.23$), SKILLS ($V = 0.15$), and HOOKS ($V = 0.13$). Other configuration mechanisms appear less frequently

in combination. CONTEXT FILES and RULES co-occur less often than independence would predict (Cramér's $V = 0.41$; 122 observed vs. 216 expected). This under-representation can be attributed to how the mechanisms are defined and which tools first introduced them. Overall, configuration mechanism adoption is characterized by recurring combinations rather than uniform usage, shaped by which mechanisms each tool supports (e.g., Claude, the dominating tool, does not support RULES). The full co-occurrence matrix is available in the supplementary material.

5.2.2 Adoption of Configuration Mechanisms over Time. Figure 4 shows the adoption of individual CONTEXT FILE artifacts, SKILLS, and SUBAGENTS over time. CONTEXT FILES (e.g., CLAUDE.md, AGENTS.md) clearly dominate and have increased continuously, while SKILLS and SUBAGENTS experienced comparatively slow growth. .cursorrules and copilot-instructions.md started being introduced in 2024, although Cursor and Copilot were originally released in 2023 and 2021, respectively; their agentic capabilities were only introduced in 2025 (see Table 2). The adoption of Copilot instructions has increased since then.

RQ2 (Summary):

- CONTEXT FILES are the dominant and often sole configuration mechanism; other mechanisms are adopted less frequently. Over time, this dominance has increased: SKILLS and SUBAGENTS have grown comparatively slowly.
- Adoption varies: Claude Code repositories use the broadest range of mechanisms, while Cursor repositories emphasize RULES. Co-occurrence patterns are shaped by which mechanisms each tool supports.
- Most repositories (70.6%) adopt a single tool, and multi-tool overlap is limited. An additional 17.3% use AGENTS.md without any tool-specific configuration.

6 Details of Configuration Mechanisms (RQ3)

We analyzed three mechanisms in detail: CONTEXT FILES and SKILLS because they are supported by all tools, and SUBAGENTS because they follow a similar format as SKILLS.

6.1 Configuration Mechanism: CONTEXT FILES

CONTEXT FILES are Markdown files that provide machine-readable context about a project. AGENTS.md, initially introduced by OpenAI, serves as an open tool-agnostic convention with growing cross-tool support [19]. We identified 4,768 CONTEXT FILES across 2,586 of the 2,853 repositories in our sample (90.6%). Most repositories include one or two such files (Figure 5).

Of the 4,768 CONTEXT FILES, CLAUDE.md is most common (1,640 files, 34.4%), followed by AGENTS.md (1,508; 31.6%) and copilot-instructions.md (1,393; 29.2%). GEMINI.md (154 files, 3.2%) and .cursorrules (73 files, 1.5%) are rare (.cursorrules are now deprecated in favor of AGENTS.md and are distinct from Cursor's RULES, stored in .cursor/rules/). At the repository level, among the 2,586 repositories with CONTEXT FILES, CLAUDE.md has the highest adoption rate at 45.9% (1,187 repos), followed by AGENTS.md

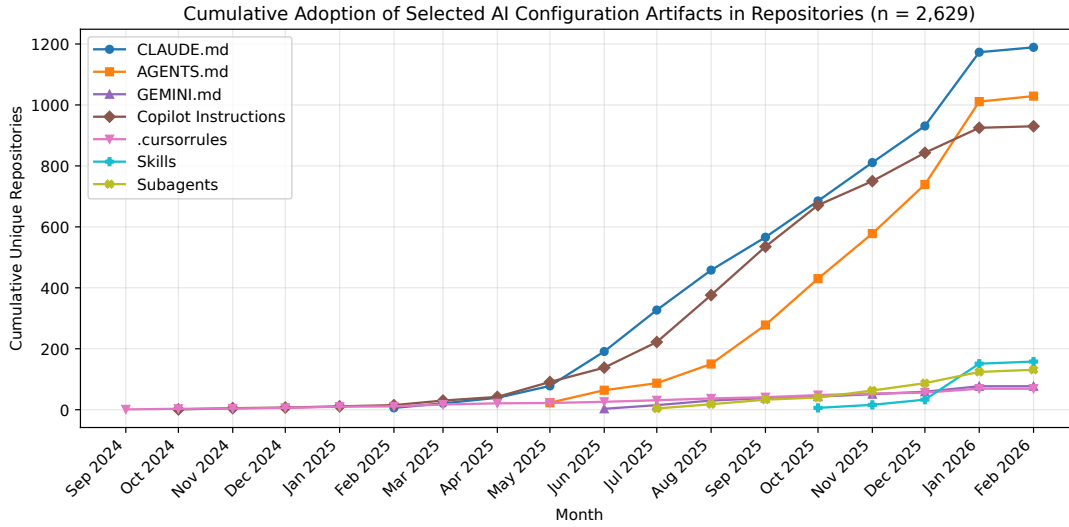


Figure 4: Cumulative adoption of selected configuration artifacts for agentic tools. Copilot Instructions comprise two CONTEXT FILE artifact types (file: copilot-instructions.md, directory: instructions/*.md, see Table 1).

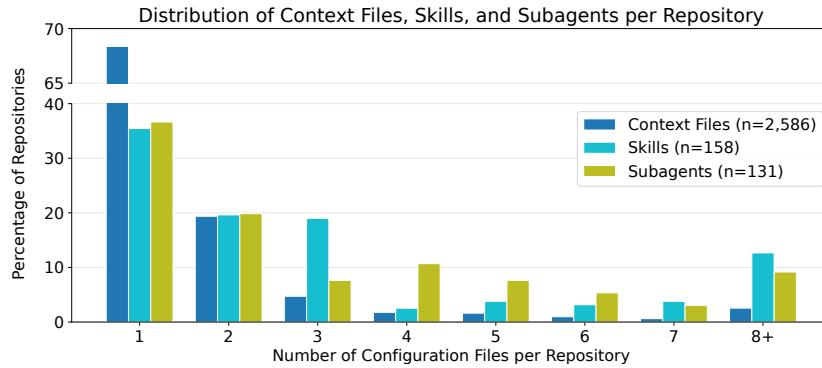


Figure 5: Configuration mechanism count per repository.

(39.5%; 1,021 repos) and copilot-instructions.md (36.0%; 930 repos). CONTEXT FILE adoption was uniformly high across languages (88.5–96.2%). CLAUDE.md was the most common CONTEXT FILE across languages, except for Java, C#, and C++, where copilot-instructions.md dominated.

Figure 6 shows the creation order of CONTEXT FILES in repositories with multiple types (single-type repositories are omitted). CLAUDE.md was typically created first, with AGENTS.md commonly added afterward—likely because Claude Code is the most popular agentic tool but does not yet support the emerging standard AGENTS.md [5]. Repositories that started with copilot-instructions.md often later added CLAUDE.md or AGENTS.md, despite Copilot supporting all major CONTEXT FILE types, reinforcing their de facto standard status.

Finally, CONTEXT FILES commonly reference each other (Figure 7). We identified three types of references, where a source file points to a target file rather than providing its own content:

- (1) Direct pointer: CONTEXT FILES include one line that references files either via their filename (e.g., “AGENTS.md”, “@./AGENTS.md”), imperative statements (e.g., “Read @AGENTS.md”), or a Markdown link (e.g., “[anchor text](AGENTS.md)”).
- (2) Short reference with context: CONTEXT FILES provide 2–5 lines of what to do with the referenced file, e.g., “Use instructions from AGENTS.md to guide your work.”, “Always read AGENTS.md before answering”, or a Markdown header followed by one of the above.
- (3) Brief summary plus reference: CONTEXT FILES include a header, 1–2 sentences of context, and a reference to another file, e.g., “# AI Guidelines [...] **Read @AGENTS.md for comprehensive guidelines.**”

We found 497 reference pairs in our sample. CLAUDE.md had the most outgoing references (344), predominantly pointing to AGENTS.md (301 times). Conversely, AGENTS.md received the most incoming

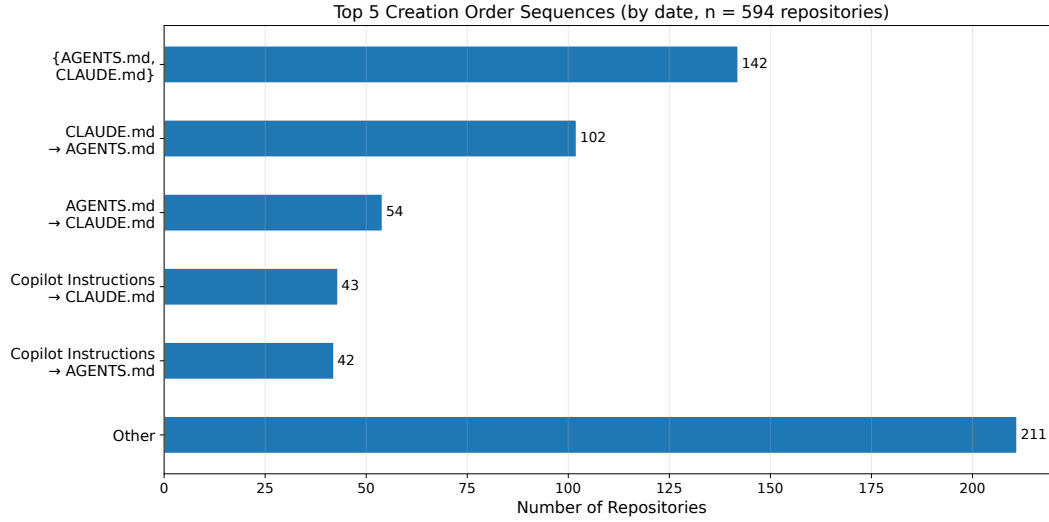


Figure 6: Creation order of CONTEXT FILES per repository. Curly braces indicate that files were added on the same day.

Table 3: Repository metadata by agentic tool. Cells show median, IQR, and Cliff’s δ for tool-adopting repositories, each compared against the complement of non-adopting repositories. Significant differences (BH-adjusted) are in bold.

Agentic tool	Age (years)	Contrib.	Commits	Size (KB)
All (n=2,853)	6.7 (4.3–9.4)	41 (19–104)	2,119 (965–5,073)	39k (9,280–132k)
Claude (n=1,297)	6.2*** (4.1–9.2) $\delta=-0.08$	40 (19–110) $\delta=0.01$	2,216 (986–5,260) $\delta=0.03$	41k (9,915–152k) $\delta=0.04$
Copilot (n=957)	7.1*** (5.0–9.7) $\delta=0.11$	42 (19–113) $\delta=0.03$	2,231 (1,001–5,607) $\delta=0.04$	45k (9,958–143k) $\delta=0.04$
AGENTS.md (n=493)	6.9 (4.2–9.5) $\delta=0.03$	42 (18–88) $\delta=-0.04$	1,841** (869–3,980) $\delta=-0.10$	24k*** (7,753–81k) $\delta=-0.13$
Cursor (n=327)	5.5*** (3.4–7.9) $\delta=-0.20$	52* (23–118) $\delta=0.10$	2,780*** (1,264–6,029) $\delta=0.14$	75k*** (21k–198k) $\delta=0.22$
Gemini (n=175)	6.7 (4.5–9.4) $\delta=0.01$	62*** (30–139) $\delta=0.19$	3,301*** (1,274–9,688) $\delta=0.19$	57k* (15k–189k) $\delta=0.11$
Codex**** (n=4)	7.6 (4.6–11) $\delta=0.12$	46 (32–62) $\delta=-0.07$	6,014 (4,997–6,514) $\delta=0.48$	505k* (153k–939k) $\delta=0.66$

Note: Mann–Whitney U test (two-sided) comparing each tool’s adopters against non-adopters (complement), with Benjamini–Hochberg (BH) FDR correction (6 categories $\times$ 4 metrics = 24 comparisons). Effect sizes: Cliff’s delta. * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$ (adjusted). **** Interpret Codex results with caution due to sample size.

references (353) and, when acting as a pointer, most frequently referenced CLAUDE.md; the same pairing pattern appears in the creation-order analysis.

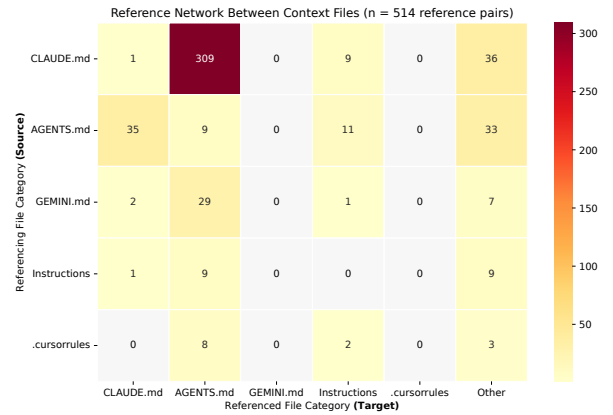


Figure 7: Reference-only CONTEXT FILES. Examples of other files include README.md and CONTRIBUTING.md.

6.2 Configuration Mechanism: SKILLS

SKILLS were introduced by Anthropic and now serve as an open standard for extending agentic AI tools with specialized contextual knowledge and workflows. SKILLS bundle prompts, tools, and documentation that an agent can invoke on demand. Each SKILL is a directory containing a SKILL.md file with YAML frontmatter (name, description) and instructions in the body. The specification recommends that SKILL.md files contain fewer than 500 lines, with detailed material moved to separate files [1]. To provide detailed information and instructions, SKILLS can include additional resource directories loaded by agents when required:

- (1) scripts/ contains executable code that agents can run. Language options include Python, Bash, and JavaScript.

- (2) `references/` contains documentation that an agent can read when needed. This can include technical references, templates or structured data and other domain-specific files.
- (3) `assets/` contains static resources, such as templates (document templates, configuration templates), images (diagrams, examples) and data files (lookup tables, schemas).

We found 601 SKILLS in 158 repositories. On average, these repositories use 3.8 SKILLS ($min = 1.0$; $max = 28.0$; $median = 2$). The distribution is right-skewed: most repositories define only one or two SKILLS, while a few define eight or more (see Figure 5). Only 29 of the 601 SKILLS (4.8%) exceeded the 500-line recommendation. Repositories defining SKILLS show the same language distribution as the overall agentic-tool sample, with TypeScript and Python being the most common (see Figure 2).

We scanned all SKILLS directories for resource folders. Of the 601 skills analyzed, the vast majority (514, 85.5%) included no additional resources. Among those that did, the most common patterns were a `references` directory and a `scripts` directory (35 skills each, 5.8%). Eleven SKILLS (1.8%) combined both `scripts` and `references`, while an `assets` directory appeared in only four SKILLS (0.7%). The remaining combinations, i.e., `references` with `assets`, and all three directories together, each occurred only once (0.2%). In summary, SKILLS mostly use static resources (documentation that the agent reads when needed) rather than dynamic resources, such as executable scripts that extend agent behavior.

6.3 Configuration Mechanism: SUBAGENTS

SUBAGENTS are specialized AI agents to which agentic AI coding tools or other agents can delegate tasks. They share the same YAML frontmatter structure as SKILLS but operate within their own context window and return results to the parent agent, whereas SKILLS execute within the calling agent’s context [4, 7].

We found 450 SUBAGENTS across 131 repositories. On average, these repositories use 3.44 SUBAGENTS ($min = 1.0$; $max = 17.0$; $median = 2$). As with SKILLS, the distribution is right-skewed: most repositories define only one or two SUBAGENTS, while a few define eight or more (Figure 5). Claude Code’s SUBAGENTS can have their own *memory*, i.e., a persistent directory that survives across interactions and can build up knowledge over time, e.g., to store debugging insights [4]. However, we did not find repositories that store such memory files.

RQ3 (Summary):

- **CONTEXT FILES:** The most common artifacts were `CLAUDE.md`, `AGENTS.md` and `copilot-instructions.md`. File creation and reference patterns revealed that `CLAUDE.md` and `AGENTS.md` serve as de facto standards, with `AGENTS.md` becoming a tool-agnostic standard adopted across tools.
- **SKILLS:** Despite their extensibility, most repositories that adopt SKILLS only define one or two and predominantly use static documentation rather than dynamic scripts.
- **SUBAGENTS:** Usage patterns match those of SKILLS, with most repositories defining only one or two. No repositories use the persistent memory feature available for Claude Code’s SUBAGENTS.

6.4 Threats to Validity

We organize this section following established guidelines [23].

Construct validity: Our heuristics (Table 1) detect the *presence* of configuration artifacts but not whether the corresponding tool is actively used, and a generic file such as `AGENTS.md` could in principle refer to any kind of AI agent, not only those used for software development. Two sampling filters mitigate this threat: The language filter restricts repositories to the ten most common programming languages on GitHub (Python, TypeScript, JavaScript, Go, Java, C++, Rust, PHP, C#, and C) so that every repository in our sample is most likely a software project, and the “engineered project” classification further excludes, among others, tutorials and resource-aggregation repositories. We additionally scanned the complete commit history of every repository that had at least one configuration file for AI-authored commits, using author/commmitter identity fields (verified bot accounts such as `copilot-swe-agent[bot]` and `claude[bot]`), git trailers (e.g., `Co-authored-by`), and commit message patterns. Among the 2,853 repositories with at least one configuration artifact, 2,058 (72.1%) also contain AI-authored commits from these tools, a lower bound because commit detection requires explicit attribution that not every tool or workflow leaves behind. This provides direct evidence that detected artifacts are associated with active AI-coding tool usage. The detection scripts and full results are included in the supplementary material [11].

Additionally, for GitHub Copilot, Cursor, and Gemini, detected files apply to both conversational and agentic modes, so we cannot isolate agentic usage. This limitation does not affect agentic-only mechanisms such as SKILLS and SUBAGENTS, nor the broader `AGENTS.md` trend. Although multi-tool overlap could conflate tool-specific patterns, 85.4% of tool-adopting repositories configure a single tool, limiting this concern. To verify that detected artifacts contain meaningful content, we programmatically scanned all configuration files across the eight artifact types. We excluded empty files, files inside vendored directories (i.e., bundled copies of third-party dependencies, such as `node_modules/` or `vendor/`), and files our earlier “`.claude-plugin parent`” rule had over-collected from nested submodules. This reduced the initial 2,926 detected repositories to 2,853 through two cleanup steps. The first step removed 45 repositories whose detected configurations were all discarded: 42 repositories whose only artifacts came from vendored dependencies and 3 removed for other reasons such as empty files. The second step excluded a further 28 repositories whose only configuration files were in `spec` directories (e.g., `.claude/`, `.cursor/`) but did not match any of the eight named mechanisms or `AGENTS.md`.

Finally, one of the authors used Claude Code (Opus 4.6 with high effort) to check all 451 files with ten or fewer lines. Of these, 396 contained substantive configuration content (e.g., coding conventions, agent definitions, tool settings), 51 were reference-only files analyzed separately in Section 6, and four were borderline cases (e.g., a bare heading or filename). The supplementary material includes the inspection script, the checked files, and their classification.

Internal validity: We classified repositories as “engineered” software projects based on their README content. During iterative prompt development, we tested GPT-5-mini, GPT-5-nano, and GPT-5.2. We selected GPT-5.2 because it produced fewer “unsure” labels

and spot-checks revealed that it produced fewer false positives and negatives. We used a single labeling run; 2,204 “unsure” cases (often due to inaccessible linked resources) were excluded. Future work should assess the reliability of this approach by adding alternative models and configurations. Detection heuristics were designed by one author and cross-checked by two different authors. Tool conventions evolve quickly; new mechanisms may not yet be captured.

External validity: Our study covers only open-source repositories on GitHub; practices in proprietary or enterprise settings may differ. We selected repositories showing established engineering practices, but cannot claim representativeness for closed-source development, nor did we examine variation across application domains. Finally, our findings are a point-in-time snapshot (February 2026) of a field whose conventions are still consolidating.

7 Discussion

Our study offers a first cross-tool snapshot of configuration mechanisms and their artifacts across five agentic AI coding tools and 2,853 GitHub repositories. Below, we discuss the implications of our findings. These mechanisms are a repository-level record of harness engineering. Developers widely adopt the static context mechanisms (e.g., CONTEXT FILES) but rarely the executable and external ones (e.g., SKILLS with scripts, HOOKS, and MCP). Harness engineering in open source today is therefore mostly context engineering.

Standardization around AGENTS.md: Our results show convergence on AGENTS.md as a tool-agnostic configuration file. Creation order analysis shows that CLAUDE.md typically appears first with AGENTS.md added later (Figure 6). Reference patterns confirm this: CLAUDE.md most frequently points to AGENTS.md (301 cases), which receives 353 incoming references overall, far more than any other file type. Together, these patterns suggest bottom-up convergence on AGENTS.md across tools, driven by developer practice rather than vendor mandate. This is further reinforced by the 493 repositories that use AGENTS.md without any tool-specific configuration, indicating that developers adopt it as a tool-agnostic standard. At the same time, layering multiple context files in a single repository creates a risk of redundant or conflicting instructions. Native AGENTS.md support is already provided by Copilot, Cursor, and Codex; Claude Code, despite being the most popular tool, does not yet support it (Section 6), creating a clear gap between developer practice and tool capability.

Limited adoption of advanced mechanisms: Adoption of all three mechanisms remains limited. For each mechanism, most repositories include only one or two artifacts (Figure 5). Moreover, 85.5% of SKILLS do not include additional resources, and when resources are present, static documentation (references/) is as common as executable scripts (scripts/). SKILLS therefore function primarily as structured text rather than executable scripts. We also found no evidence of repositories using the persistent memory feature of Claude Code’s SUBAGENTS.

This gap likely reflects both the novelty of these mechanisms and the effort required to configure them. Developers may prefer the simplest mechanism (i.e., CONTEXT FILES), since defining executable SKILLS with scripts and structured resources requires more design and maintenance effort than authoring Markdown

files. Future work should assess whether deeper configuration leads to measurable performance gains, extending early evidence on the impact of CONTEXT FILES [16].

Distinct tool ecosystems: Configuration practices differ across tools: Claude Code repositories use the broadest range of mechanisms, Cursor projects emphasize RULES and COMMANDS, and Copilot repositories rarely extend beyond CONTEXT FILES. These differences likely reflect the configuration options each tool exposes (see Table 1). Repository characteristics also differ: Cursor repositories tend to be younger and larger, while Gemini repositories show higher activity levels (see Table 3). It remains an open question whether these tool-specific configuration practices will converge as tool capabilities increasingly overlap, or whether distinct usage patterns will persist.

Practical implications and future directions: Table 1 helps developers understand available configuration mechanisms. Based on our findings, we highlight four implications for practitioners. First, CONTEXT FILES, particularly AGENTS.md, are the simplest entry point for configuring agentic tools and are already widely adopted. Second, developers who rely on multiple tools should maintain an AGENTS.md file as the shared core configuration, given its cross-tool support and the reference patterns we observed. When multiple CONTEXT FILES coexist, teams may benefit from structuring them hierarchically (e.g., using tool-specific files as adapters that reference a shared core file). Third, for recurring, well-defined workflows, SKILLS offer the option of bundling scripts and structured resources, but in practice most SKILLS do not include such resources. Finally, adopting tool-specific mechanisms such as RULES or COMMANDS ties workflows to a specific tool.

For researchers, our findings motivate longitudinal studies to track how configuration practices evolve as tools mature, and controlled studies to assess whether SKILLS with executable resources or dedicated SUBAGENTS provide measurable benefits over CONTEXT FILES alone [16]. Given that some repositories configure multiple tools (Section 5.1), research should also investigate how conflicts between overlapping instructions across files can be detected and resolved. Our data collection and analysis pipeline [11] supports such ongoing analyses.

Finally, our findings represent a point-in-time snapshot. The trends we identify—toward standardization around AGENTS.md, limited adoption of advanced mechanisms, and tool-specific configuration practices—are early empirical signals rather than settled findings.

8 Conclusions

Our study provides the first systematic, repository-level analysis of harness engineering for agentic AI coding tools, identifying eight configuration mechanisms across five tools and analyzing their adoption in 2,853 GitHub repositories. Three findings stand out. First, CONTEXT FILES dominate and are often the only mechanism present. AGENTS.md is adopted across tools as a tool-agnostic standard, even when a developer’s primary tool does not yet support it natively. Second, few repositories adopt advanced mechanisms such as SKILLS and SUBAGENTS; most SKILLS do not bundle scripts or other executable resources, and we found no repositories using

the persistent memory feature available for Claude Code's SUB-AGENTS. Third, distinct configuration practices are forming around different tools: Claude Code repositories use the broadest range of mechanisms, Cursor projects emphasize RULES and COMMANDS, and Copilot repositories rarely extend beyond CONTEXT FILES.

For practitioners, AGENTS.md is a natural starting point for configuring agentic tools, especially in multi-tool environments, and SKILLS offer further options for encoding recurring workflows through scripts and structured resources. Tool providers should improve onboarding for SKILLS and SUBAGENTS, which few repositories use despite their support for executable scripts and isolated agent contexts. For researchers, controlled studies should determine whether richer configuration improves outcomes over CONTEXT FILES alone, and longitudinal research should track how these patterns evolve as tools mature and developer norms consolidate.

References

- [1] agentskills.io. 2026. Agent Skills. <https://agentskills.io/>.
- [2] agentsmd community. 2025. AGENTS.md: A Simple, Open Format for Guiding Coding Agents. Website. <https://agents.md/>. Accessed 2026-01-18.
- [3] Anthropic. 2025. Claude 3.7 Sonnet and Claude Code. <https://www.anthropic.com/news/claude-3-7-sonnet>.
- [4] Anthropic. 2026. Create custom subagents. <https://code.claude.com/docs/en/sub-agents>.
- [5] anthropics/claude-code on GitHub. 2026. Feature Request: Support AGENTS.md. <https://github.com/anthropics/claude-code/issues/6235>.
- [6] Worawalan Chatlatanagulchai, Hao Li, Yutaro Kashiwa, Brittany Reid, Kundjana-sith Thonglek, Pattara Leelaprute, Arnon Rungsawang, Bundit Manaskasemsak, Bram Adams, Ahmed E. Hassan, and Hajimu Iida. 2025. Agent READMEs: An Empirical Study of Context Files for Agentic Coding. arXiv:2511.12884 [cs.SE] doi:10.48550/arXiv.2511.12884
- [7] Cursor. 2026. Subagents. <https://cursor.com/docs/context/subagents>.
- [8] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, Madrid, Spain, 560–564. doi:10.1109/MSR52588.2021.00074
- [9] DAIR.AI Prompt Engineering Guide. 2025. Elements of a Prompt | Prompt Engineering Guide. <https://www.promptingguide.ai/introduction/elements>.
- [10] Matthias Galster, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. 2026. Configuring Agentic AI Coding Tools: An Exploratory Study. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software (AIware '26)*. ACM, Montreal, QC, Canada. doi:10.1145/3805760.3814887
- [11] Matthias Galster, Seyedmoein Mohsenimofidi, Jai Lal Lulla, Muhammad Auwal Abubakar, Christoph Treude, and Sebastian Baltes. 2026. *Configuring Agentic AI Coding Tools: An Exploratory Study (Supplementary Material)*. doi:10.5281/zenodo.18625980
- [12] Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. 2025. Speed at the Cost of Quality: How Cursor AI Increases Short-Term Velocity and Long-Term Complexity in Open-Source Projects. arXiv:2511.04427 [cs.SE] doi:10.48550/arXiv.2511.04427 To appear at the 23rd IEEE/ACM International Conference on Mining Software Repositories (MSR 2026), Rio de Janeiro, Brazil.
- [13] Dexter Horthy. 2025. Getting AI to Work in Complex Codebases. <https://github.com/humanlayer/advanced-context-engineering-for-coding-agents/blob/main/ace-fca.md>.
- [14] Shaokang Jiang and Daye Nam. 2025. Beyond the Prompt: An Empirical Study of Cursor Rules. arXiv:2512.18925 [cs.SE] doi:10.48550/arXiv.2512.18925 To appear at the 23rd IEEE/ACM International Conference on Mining Software Repositories (MSR 2026), Rio de Janeiro, Brazil.
- [15] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. 2014. The promises and perils of mining GitHub. In *11th Working Conference on Mining Software Repositories, MSR 2014*. ACM, Hyderabad, India, 92–101. doi:10.1145/2597073.2597074
- [16] Jai Lal Lulla, Seyedmoein Mohsenimofidi, Matthias Galster, Jie M. Zhang, Sebastian Baltes, and Christoph Treude. 2026. On the Impact of AGENTS.md Files on the Efficiency of AI Coding Agents. arXiv:2601.20404 [cs.SE] doi:10.48550/arXiv.2601.20404 To appear at the 1st Journal Ahead Workshop (JAWs@ICSE 2026).
- [17] Damon McMillan. 2026. Structured Context Engineering for File-Native Agentic Systems: Evaluating Schema Accuracy, Format Effectiveness, and Multi-File Navigation at Scale. arXiv:2602.05447 [cs.SE] doi:10.48550/arXiv.2602.05447
- [18] Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, Chenlin Zhou, Jiayi Mao, Tianze Xia, Jiafeng Guo, and Shenghua Liu. 2025. A Survey of Context Engineering for Large Language Models. arXiv:2507.13334 [cs.CL] doi:10.48550/arXiv.2507.13334
- [19] Seyedmoein Mohsenimofidi, Matthias Galster, Christoph Treude, and Sebastian Baltes. 2025. Context Engineering for AI Agents in Open-Source Software. arXiv:2510.21413 [cs.SE] doi:10.48550/arXiv.2510.21413 To appear at the 23rd IEEE/ACM International Conference on Mining Software Repositories (MSR 2026), Rio de Janeiro, Brazil.
- [20] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empir. Softw. Eng.* 22, 6 (2017), 3219–3253. doi:10.1007/S10664-017-9512-6
- [21] OpenAI. 2025. Introducing Codex. <https://openai.com/index/introducing-codex/>.
- [22] Romain Robbes, Théo Matricon, Thomas Degueule, André C. Hora, and Stefano Zacchiroli. 2026. Agentic Much? Adoption of Coding Agents on GitHub. arXiv:2601.18341 [cs.SE] doi:10.48550/arXiv.2601.18341
- [23] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empir. Softw. Eng.* 14, 2 (2009), 131–164. doi:10.1007/S10664-008-9102-8
- [24] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. 2024. A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. arXiv:2402.07927 [cs.AI] doi:10.48550/arXiv.2402.07927
- [25] Helio Victor F. Santos, Vitor Costa, João Eduardo Montandon, and Marco Túlio Valente. 2025. Decoding the Configuration of AI Coding Agents: Insights from Claude Code Projects. arXiv:2511.09268 [cs.SE] doi:10.48550/arXiv.2511.09268
- [26] Philipp Schmid. 2025. The New Skill in AI is Not Prompting, It's Context Engineering. <https://www.philpschmid.de/context-engineering>.
- [27] SEART. 2025. GitHub Search. <https://seart-ghs.su.uzi.ch/>.
- [28] Agnia Sergeev, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. 2025. Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Information and Software Technology* 178 (2025), 107610. doi:10.1016/j.infsof.2024.107610
- [29] Stack Exchange Inc. 2026. Stack Overflow Developer Survey 2025: AI Agent out-of-the-box tools. <https://survey.stackoverflow.co/2025/ai/#3-ai-agent-out-of-the-box-tools>.
- [30] Valerio Terragni, Annie Vella, Partha Roop, and Kelly Blincoe. 2025. The Future of AI-Driven Software Engineering. *ACM Trans. Softw. Eng. Methodol.* 34, 5, Article 120 (May 2025), 20 pages. doi:10.1145/3715003
- [31] Hugo Villamizar, Jannik Fischbach, Alexander Korn, Andreas Vogelsang, and Daniel Méndez. 2025. Prompts as Software Engineering Artifacts: A Research Agenda and Preliminary Findings. In *Product-Focused Software Process Improvement - 26th International Conference, PROFES 2025, Salerno, Italy, December 1-3, 2025, Proceedings (Lecture Notes in Computer Science)*. Springer, Cham, 470–478. doi:10.1007/978-3-032-12089-2_32
- [32] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10-15, 2024*. Curran Associates, Inc., Red Hook, NY, USA, 50528–50652. doi:10.52202/079017-1601
- [33] Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. 2026. Meta Context Engineering via Agentic Skill Evolution. arXiv:2601.21557 [cs.AI] doi:10.48550/arXiv.2601.21557
- [34] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. 2025. Agentic Context Engineering: Evolving Contexts for Self-Improving Language Models. arXiv:2510.04618 [cs.CL] doi:10.48550/arXiv.2510.04618
- [35] Yifan Zhang, Yang Yuan, Mengdi Wang, and Andrew Chi-Chih Yao. 2025. Monadic Context Engineering. arXiv:2512.22431 [cs.AI] doi:10.48550/arXiv.2512.22431
- [36] Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, Congming Zheng, Jiachen Zhu, Zeyu Zheng, Zhuosheng Zhang, Xingyu Lou, Changwang Zhang, Zhihui Fu, Jun Wang, Weiwen Liu, Jianghao Lin, and Weinan Zhang. 2026. Externalization in LLM Agents: A Unified Review of Memory, Skills, Protocols and Harness Engineering. arXiv:2604.08224 [cs.SE] doi:10.48550/arXiv.2604.08224

Received 2026-02-15; accepted 2026-03-28