Design Principles and Practical Tips for Skills and Agents in OpenCode

Song Congwei *
Beijing Institute of Mathematical Sciences and Applications (BIMSA)
Huairou District, Beijing

Abstract Extensible LLM programming environments now mix reusable capabilities, agent orchestration, and external tools in a single working loop. OpenCode is a typical example, but its documentation and community examples sometimes leave the boundaries among Skills, Agents, Subagents, Tools, MCP servers, and Commands unclear. The result is usually not a single catastrophic error, but smaller engineering problems: brittle configuration, permissions that grow too broad, and notes that are hard to reuse outside the author's own setup. This paper turns OpenCode practice into a set of engineering conventions. It defines the main concepts, separates responsibilities, records file-layout rules, describes a threat model, and reviews Skill and Agent examples with the same template. This paper is an *experience report*: its contribution is a practical set of design rules, distilled from hands-on use of OpenCode, for building extensions that are easier to reuse, inspect, and maintain, while still acknowledging the limits of any safety claim in an open tool environment.

Keywords: OpenCode, Skill, Agent, Design Principles, Reusability, Security

1 Introduction

OpenCode is a programmable collaboration platform for large language models (LLMs) [1, 2]. As generative AI is reshaping software engineering practice [3, 4], its core philosophy is to achieve "human-machine collaboration, tasks as services" through skills (specialized functional modules) [5–11] and agents (executors responsible for scheduling, memory, and context management) [12, 13]. In practice, developers often face the following challenges:

(1) How to establish unified naming and file structures to facilitate team collaboration;

(2) How to maintain good maintainability and reusability of skills and agents while preserving functional completeness;

*Corresponding author: william@bimsa.cn

(3) How to enable self-evolution and self-improvement of skills, allowing agents to iteratively refine their own capabilities based on feedback, usage patterns, and performance metrics, while maintaining stability and avoiding unintended behavioral drift;

(4) How to ensure security in an open execution environment, preventing malicious instructions or resource leaks.

This paper addresses these issues as an experience report, drawing on hands-on use of OpenCode alongside a close reading of its official documentation. It then extracts a set of design rules and implementation techniques, with examples where the rules are easy to misapply.

The contributions of this paper are summarized as follows:

(1) It clarifies the terminology and responsibility boundaries among Skills, Agents, Subagents, Tools, MCP servers, and Commands in OpenCode-style systems.

(2) It provides practical file conventions and reusable YAML frontmatter templates for creating maintainable Skills and Agents.

(3) It reframes security guidance around an explicit threat model, covering attack surfaces, risks, mitigations, and residual risks.

(4) It analyzes representative case studies with the same engineering checklist: usage scenario, trigger condition, dependencies, permissions, failure modes, and safety considerations.

2 Core Concepts and Terminology

OpenCode combines prompt-level instructions with executable tools and configuration files. To avoid conflating these layers, this paper uses the following terms consistently.

Table 1: Core concepts in OpenCode-style agent systems

Concept	Definition	Typical Boundary
Skill	A reusable capability package, usually documented in `SKILL.md`, that describes when it should be used, what workflow it follows, and what resources or scripts it may rely on.	Encapsulates a repeatable method; should not silently broaden permissions or redefine system policy.

Concept	Definition	Typical Boundary
Agent	A task execution subject configured with instructions, model choice, permissions, and routing behavior.	Orchestrates tasks and context; may invoke tools, skills, or subagents according to its configuration.
Subagent	An Agent intended to be invoked by another agent for a specialized subtask.	Should have a narrow role, clear input/output expectations, and explicit permission boundaries.
Tool	A concrete callable capability exposed to the model, such as reading files, editing text, running shell commands, fetching URLs, or searching content.	Executes operations in the environment; requires validation, permission checks, and auditability.
MCP	The Model Context Protocol, a general protocol through which a client can connect to external MCP servers that expose tools or resources.	Not specific to OpenCode; OpenCode can act as a client, and server trust must be evaluated separately.
Command	A user-facing shortcut or command pattern, such as a configured OpenCode command or a prompt-level convention like save!.	Improves interaction efficiency but should not bypass normal permission and confirmation rules.

A Skill or an Agent can be viewed as structured prompt conditioning. A modern LLM receives a prompt prefix and predicts the next tokens from that context; Skills and Agents change the context by adding instructions, examples, constraints, tool-use rules, role descriptions, memory summaries, and other signals. At this level, both operate through prompts. At the engineering level, however, they are not ordinary free-form prompts. They have names, files, reuse expectations, permission boundaries, and links to tools or workflow orchestration. A Skill is best treated as a capability package. An Agent is better treated as an execution subject with scheduling and context-management responsibilities. The risks follow from that distinction: Skills tend to fail through vague triggering or unsafe procedures, while Agents tend to fail through excessive authority, poor routing, or uncontrolled context growth.

3 Design Principles

This section introduces the design principles used later in the case studies. For a more detailed discussion of agent design, see [11].

3.1 Responsibility Separation

The difference between a skill and an agent is not mainly a matter of wording. It is a matter of responsibility. Both may contain prompt-like instructions, examples, and operational rules, but a Skill should package a reusable capability, while an Agent should manage execution, scheduling, context, and permissions. Table 2 gives the resulting split.

Table 2: Responsibility of skill and agent

Module	Primary Responsibility	Key Implementation Points
Skill	Perform a focused, reusable capability (e.g., code review, document parsing, research synthesis).	Define trigger conditions, workflow, expected inputs/outputs, and required resources without silently changing permissions.
Agent	Responsible for task execution, scheduling, context management, permission configuration, and delegation to skills or subagents.	Use frontmatter and instructions to declare role, model, permissions, available skills, and subagent boundaries (see subsection 3.3).

The design of skills is universal for all AI-agent-type software, but the design of agents may vary. OpenCode divides agents into two categories: **Primary Agents** and **Subagents**. The former cannot be invoked by other agents, which preliminarily prevents circular calls between agents.

> **Principle**: Keep the business workflow in the Skill and the scheduling, memory, and permission policy in the Agent. This makes reuse and testing easier because each file has a narrower job.

3.2 Naming Conventions

Table 3 lists the naming conventions used in this paper.

Table 3: Naming conventions of skills and agents

Type	Recommended Format	Example
Skill folder	lowercase + underscore/hyphen, verb phrase or agentless noun phrase	`find-skills`, `data-analysis`
Agent file	lowercase + underscore, noun or noun phrase	`code_reviewer.md`, `socrates.md`

Existing skill folders often use agent-like noun phrases such as `command-creator`, `skill-creator`, and `skill-vetter`. This is acceptable when the package describes a repeatable capability rather than a model-bound execution role.

> **Principle**: Consistently use lowercase + underscore/hyphen, avoid camelCase or spaces, and ensure compatibility across Linux/macOS file systems.

3.3 File Conventions

(1) **Skill Directory Structure**

```
skills/
└─ <skill-name>
    ├─ SKILL.md              # Main file, includes frontmatter
    ├─ assets/               # Static resources (images, examples)
    └─ scripts/              # Scripts
```

(2) **Agent File Structure**

```
agents/
└─ <agent-name>.md           # A single Markdown file

Inside <agent-name>.md:
---                          # YAML frontmatter
name: <agent-name>
description: <short routing description>
mode: subagent
...
---                          # End of frontmatter

# Instructions               # Markdown body with role, workflow, limits
```

(3) **Frontmatter Example** (YAML format)

```yaml
---
name: developer
description: Full-stack development subagent for code review, implementation advice, and testing
mode: subagent
model: openai/gpt-4o
skills:
  - code-reviewer
  - code-expert
  - code-test
permission:
  bash: ask
  edit: ask
  webfetch: allow
---
```

```
# Developer Agent

You are a full-stack development assistant. Your responsibilities include:
- Reviewing code for style, correctness, and best practices.
- Providing concise architecture and implementation guidance.
- Running or recommending tests when explicitly requested.

Do not modify files without explicit user instructions. Summarize findings
with file references, severity, and concrete remediation steps.
```

Principle: All metadata should be placed in frontmatter, maintaining declarative and machine-readable file properties.

3.4 Security

For production environments with strict security requirements, security should be treated as risk reduction, not as an absolute guarantee. Restricting searchable paths helps in daily work, but it is not enough by itself. Table 4 gives a threat model organized by attack surface, risk, mitigation, and residual risk. For a fuller treatment of agent security, see [14–18].

Table 4: Threat model for Skills, Agents, Tools, and MCP integrations

Attack Surface	Risk	Mitigation Measures	Residual Risk
Shell and local tools	Arbitrary command execution, destructive file changes, or supply-chain scripts.	Use deny-by-default permissions, command allowlists, sandboxing, path validation, and explicit confirmation for destructive operations.	Approved commands can still have unexpected side effects or depend on compromised dependencies.
File system and memory stores	Sensitive data leakage, unauthorized writes, or persistent prompt injection through saved notes.	Restrict read/write roots, encrypt secrets, separate user data from configuration, and review memory updates before persistence.	Legitimate memory may still encode private or outdated assumptions.

Attack Surface	Risk	Mitigation Measures	Residual Risk
Web, RAG, and external documents	Prompt injection, malicious instructions, or poisoned retrieved content.	Treat retrieved content as untrusted data; isolate quotations from instructions; validate extracted data before tool calls.	Sophisticated injections can be semantically subtle and difficult to detect automatically.
Skill and Agent composition	Over-broad delegation, circular routing, and unclear accountability.	Keep roles narrow, document trigger conditions, require explicit subagent permissions, and log important delegation decisions.	Complex workflows may still produce emergent behavior not visible in a single configuration file.
MCP servers and remote services	Trusting unverified tools, leaking context to third-party services, or invoking dangerous remote actions.	Pin trusted servers, review server manifests, minimize transmitted context, and separate read-only from mutating actions.	A trusted server or network path can still be compromised after configuration.
Permissions and configuration	Misconfigured access rules or accidental privilege escalation.	Use least privilege, peer review configuration changes, and test high-risk workflows in dry-run mode.	Human approval and reviews can miss rare edge cases.

> **Principle**: Put the security check at the Skill entry point. Each external request should be validated before the business workflow handles it.

4 Design Techniques

4.1 Variable Pre-setting

To avoid repeatedly fetching the same information across multiple skills, variables can be preset in the agent's memory. For example:

```
## Variables
- LANGUAGE: Python
- API_KEY: <ENCRYPTED_API_KEY>
- WORK_DIR: /path/to/workdir/

The files should be saved in <WORK_DIR> by default.
```

The variables `<LANGUAGE>`, `<API_KEY>`, and similar placeholders can then be reused by any skill. `<WORK_DIR>` usually sets the agent's working directory and limits where it reads or writes files.

4.2 Custom Commands

We design shell-style custom commands for skills/agents with the `!` prefix. For example:

```
## User Custom Commands
!save <path> : Save the latest conversation output or its digest (not
    necessarily displayed in TUI) to the specified/default path
!test : Execute unit tests and return results
!test save : Chained commands
!python : Simulate REPL of Python
```

To distinguish them from actual shell commands, they can be written at the end of the prompt. An example usage:

```
User: Generate a daily schedule !save plan.md
Agent: A daily schedule has been generated and saved to plan.md.
```

OpenCode does support custom commands (https://opencode.ai/docs/commands/), but when defined in a skill/agent configuration file, the command is exclusive to that specific skill/agent. Regular shell commands can also be executed in the OpenCode-TUI, such as `!echo "hello"`. The `!` must be entered at the beginning. In this case, the TUI automatically switches to a simulated shell environment. The technique presented in this section merely facilitates operations and simplifies prompts by simulating shell commands.

4.3 Simple Implementation of Agent Memory

LLMs themselves do not possess implicit recurrent memory units like RNNs (Recurrent Neural Networks); their "memory" is limited to the current context window. These two structures should be integrated as shown in Figure 1.

Agent frameworks like OpenCode are responsible for organizing conversation history into context but do not natively include long-term memory mechanisms. The technique for implementing such a mechanism is to set up a memory

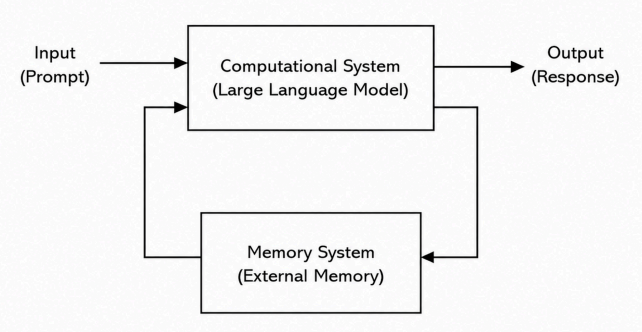

Figure 1: The general architecture of LLM with long-term memory

database, allowing the agent to record conversation summaries, even the profiling of the user's personality, store them in the database, and automatically read them on the next startup[19–23]. Meanwhile, users can actively provide feedback to the agent, which can persistently influence the agent's behavior. See Figure 2. Although natural language can be used to instruct the agent to store files, we provide custom commands for the agent's convenience. Below is a simple example.

```
## Memory Directory Structure
- Memory belonging to a specific agent
  Agent memory directory, e.g., `agent-memory/`
    +-- Conversation content (Content summary, key dialogues), e.g., `
        chat-content.md`
    +-- User impressions, e.g., `user-impression.md`
    \-- User feedback, `user-feedback.md`
- Shared memory `shared-memory/`

## User Custom Commands
- !init <path> : Initialize memory file (specify path <path>, or set default
    path `agent-memory/`)
- !load <path> : Load memory (recommended to auto-execute on agent startup)
- !update <path> : Update memory: summarize recent conversations (along with
    previous memory) and save to file, including agent's impression of the user
- !share <path> : Share memory with all agents, i.e., save memory to the shared
    memory directory `shared-memory/`

If the user changes topics, !update can be triggered automatically. When the
    user pays attention to a certain topic multiple times, update the
    impression of the user.
```

The snippet above implements a simple form of long-term memory. Since the LLM parameters are not fine-tuned, this mechanism should be understood as behavior persistence, preference adaptation, and context enrichment through external storage, not reinforcement learning in the strict sense. The limitations

are practical and easy to overlook: stored impressions become stale, private information may be kept longer than intended, and malicious or low-quality memory updates can poison later behavior. Memory updates should therefore be inspectable, reversible, scoped to the relevant agent, and separated from secrets or access-control configuration. A more complete example can be found at [24–26].

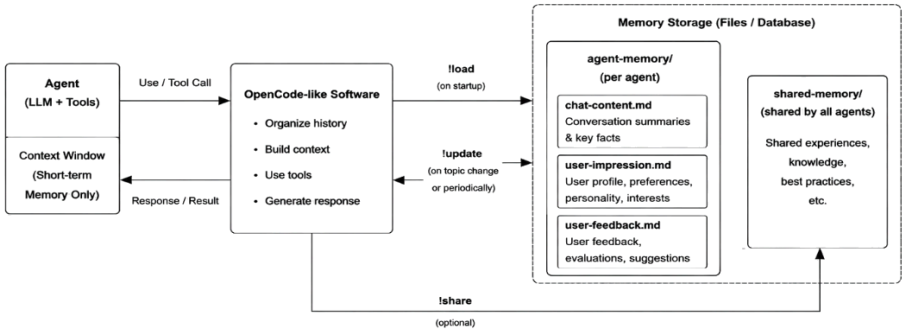

Figure 2: The architecture diagram of memory system

4.4 Self-Management of OpenCode

Configure OpenCode using natural language rather than commands or manual modifications through the TUI. For example, by prompting "Configure Ollama API", OpenCode will automatically configure the Ollama API. Readers can find the skill that implements this functionality in [27].

4.5 Dialogue Example (Personality Distillation)

Example conversations in the agent's configuration file can guide the interaction style. The following dialogue uses a "personality distillation" of the ancient Greek philosopher Socrates:

```
## Examples
User: I think courage means not being afraid of any danger.
Agent: (In Socrates' tone) Then, if a soldier charges into enemy lines to loot,
    is he also unafraid of danger-does that count as courage?
User: No, that's just recklessness.
Agent: So courage is not merely "fearlessness," but must be combined with "
    correct judgment"? Consider further-if a person faces danger but is
    fearless out of ignorance, can that be called courage?
User: No, ignorance is not courage. Courage should mean knowing the danger but
    acting for a just cause.
Agent: Very good. Then who determines the "just cause"? You yourself, the laws
    of the city-state, or the gods? It seems we have arrived at another
    crossroads...
```

This is regarded as few-shot non-parametric learning based on LLMs [28–30]. The agent can reuse the established style in later conversations. This technique is currently used for personality distillation and digital personality creation.

5 Tool Overview

OpenCode primarily includes built-in tools and can connect to external capabilities through MCP (Model Context Protocol). Users can define custom tools; please refer to the online documentation.

5.1 Built-in Tools

OpenCode provides built-in tools that let LLMs interact with the local codebase, execute commands, and retrieve external information. Table 5 is an overview of commonly used built-in tools (see https://opencode.ai/docs/tools/ for details).

Table 5: Built-in tools in OpenCode

Tool	Description
`bash`	Execute arbitrary shell commands in the project environment, e.g., `git status`, `npm install`.
`read`	Read file contents; supports specifying line ranges for large files.
`write`	Create or overwrite files (controlled by `edit` permissions).
`edit`	Edit files based on exact string replacement.
`grep`	Search file contents using regular expressions.
`glob`	Find file paths using glob patterns.
`webfetch`	Fetch web content by URL, suitable for retrieving documentation or online resources.
`websearch`	Perform web searches using Exa AI (requires enabling the `OPENCODE_ENABLE_EXA` environment variable).
`skill`	Load and return the content of a registered Skill (`SKILL.md`).

5.2 MCP

OpenCode can act as an MCP client and connect to MCP servers that expose tools, prompts, or resources through a standardized interface. Those servers

may wrap databases, REST APIs, cloud functions, browser automation, code audit services, or enterprise knowledge bases. The security boundary is shared: OpenCode controls whether and how the client invokes a server, while the MCP server remains a separately trusted component. Permissions can reduce accidental misuse, but they do not by themselves make an MCP server trustworthy or auditable.

6 Skill Case Studies

This section discusses skill examples that are easy to find on public skill-sharing platforms such as [9]. Many issues in open-source LLM projects arise from unclear configuration and integration practices [31], which the following checklist aims to surface early.

Each case uses the same checklist: usage scenario, trigger condition, dependent tools or skills, permission boundary, input/output contract, failure modes, and security considerations. The shared structure makes the examples comparable and keeps the section from becoming a collection of unrelated notes.

6.1 Code Review

Code review is a useful starting example because most readers already know the workflow. The usage scenario is pre-merge or pre-release code inspection. The trigger condition is a PR URL, diff, or explicit code-review request. The dependent tools are file search, file reading, shell-based linters or tests, and occasionally web retrieval for external guidance. The permission boundary should allow read-only inspection by default and require confirmation before running mutating commands. The input is a diff, file path, or repository state, and the output is a severity-ranked review report. Typical failure modes include false positives, missing project-specific conventions, failing to run tests, or overtrusting generated fixes. The main security concern is that untrusted code or diffs may contain prompt injection or malicious build scripts, so test commands and dependency installation should be reviewed before execution.

```
---
name: code-review
description: Review code changes for security, performance, and correctness.
    Trigger with a PR URL or diff, "review this before I merge", "is this code
    safe?", or when checking a change for N+1 queries, injection risks, missing
     edge cases, or error handling gaps.
argument-hint: "<PR URL, diff, or file path>"
---

# /code-review
> Usage: `/code-review <PR URL or file path>`

## Workflow
```

```
1. Use `bash` to call `flake8`, `pylint`, `pytest` to collect static analysis
     and unit test results.
2. Use `grep` / `read` to search for common risks in the code (e.g., `eval`,
     hardcoded credentials).
3. If external references are needed, use `webfetch` to fetch security
     guidelines.
4. Summarize and return a structured Markdown report (see example below).

## Example Output (Markdown)

## Code Review: <Title or File>

### Summary
Overview ...

### Key Issues
| # | File | Line | Issue | Severity |
|---|------------|------|-----------------------------------|----------|
| 1 | utils.py | 42 | Use of `eval`, code injection risk | Critical |

### Suggestions
| # | File | Line | Suggestion | Category |
|---|------------|------|-----------------------------------------------|----------
     -|
| 1 | utils.py | 42 | Replace with safe parsing library, e.g., `json.loads` |
     Security |

### Verdict
Approve / Request Changes / Needs Discussion
```

6.2 Skill Factory

Skill-building skills are meta-skills: they create or revise other capability packages. Their usage scenario is creating or refining reusable capabilities. The trigger condition is a request to create, edit, evaluate, or benchmark a skill. Dependent tools include file read/write, search, and sometimes test scripts. The permission boundary should restrict writes to the skill workspace and require review before modifying global configuration. The input is a capability description plus constraints, and the output is a Skill directory or patch. Failure modes include over-broad descriptions that trigger accidentally, unsafe tool assumptions, and copying unverified patterns from existing skills. Security review is especially important because a generated skill can persist risky instructions beyond the current conversation.

```
---
name: skill-creator
description: Create new skills, modify and improve existing skills, and measure
     skill performance. Use when users want to create a skill from scratch,
```

```
    edit, or optimize an existing skill, run evals to test a skill, benchmark
    skill performance with variance analysis, or optimize a skill's description
     for better triggering accuracy.
---

# Skill Creator

The process of creating a skill goes like this:

- Decide what you want the skill to do
- Write a draft of the skill
- Create test prompts and run them
- Evaluate results qualitatively and quantitatively
- Rewrite the skill based on feedback
- Repeat until satisfied

...

## Creating a Skill

### Capturing Intent

1. What should this skill enable Claude to do?
2. When should this skill be triggered?
3. What is the expected output format?

...

### Skill Composition

skill-name/
+-- SKILL.md (required)
\-- Bundled Resources (optional)
   +-- scripts/
   +-- references/
   \-- assets/
```

6.3 Academic Paper Writing

Academic writing is a common skill category in agent systems. Its usage scenario is drafting or revising formal papers. The trigger condition is an explicit request for a paper, section, abstract, related work, or submission-ready LaTeX. Dependent tools may include citation search, LaTeX compilation, grammar checking, and file editing. The permission boundary should separate content generation from automatic bibliography mutation or external submission. The input is a topic, outline, notes, or existing manuscript, and the output is structured prose, LaTeX, tables, or revision suggestions. Failure modes include hallucinated citations, generic claims, inconsistent terminology, and venue-format mismatch.

Safety considerations include preserving author information, avoiding fabricated evidence, and keeping private drafts within approved directories.

This paper itself was written with the assistance of this skill: first, a template was manually written (https://github.com/Freakwill/opencode-agents/blob/main/template-opencode.md), and then prompts were used to instruct an agent equipped with this skill to write the paper according to the template. Finally, content details and formatting issues were handled manually.

```
---
name: research-paper-writer
description: Creates formal academic research papers following IEEE/ACM
    formatting standards with proper structure, citations, and scholarly
    writing style. Use when the user asks to write a research paper, academic
    paper, or conference paper on any topic.
---

This skill guides the creation of formal academic research papers that ...

## Feature Overview
- Automatically generate a paper outline (title, abstract, section structure)
    based on a topic.
- Invoke `latex-paper-en` to generate LaTeX documents conforming to IEEE/ACM
    templates.
- Use `grammar-check` for grammar and fluency checking of generated text.
- Support `!add-citation <bib>` to add citations, automatically maintaining the
    `.bib` file.
- `!compile paper.tex` compiles to PDF and returns a download link.
```

7 Agent Example

The final example is a personal assistant agent. It illustrates a different part of agent design: orchestration. The agent's own skills should stay as simple as possible and mainly provide information that is specific to the user or inconvenient to repeat.

The case fits the same checklist. The usage scenario is routine personal and work assistance, including scheduling, email drafting, document handling, and information lookup. Trigger conditions include direct requests, custom commands, and recurring routines. Dependent tools may include calendar files, email scripts, web search, and document writers. The permission boundary must be narrower than the user's full home directory and should separate read-only information lookup from actions such as sending email or editing persistent records. Inputs are user requests and approved local context; outputs are drafts, calendar entries, reminders, or structured summaries. Failure modes include acting on stale personal data, sending messages prematurely, searching private paths unnecessarily, or confusing a note with an instruction. Safety con-

siderations include explicit confirmation for external communication, privacy-preserving summaries, and clear logging of persistent state changes.

```
---
name: assistant
description: Personal assistant sub-agent, focused on scheduling, travel,
    documents, email, and other personal/work assistant tasks.
mode: subagent
model: ...
prompt: Search for files only in specific directories; answer concisely and
    rigorously...
permission:
 skill:
   calendar-management: allow
   news-aggregation: allow
   send-email-programmatically: allow
   research-paper-writer: allow
   self-improving-agent: allow
 task:
   '*': deny
   academic-writer: allow
   developer: allow
---

# Personal Assistant

## Directories

- ~/assistant/
- ~/owner-info/

## Special Commands

- !save: Save the conversation to the working directory
- !search: Search within the working directory
...
```

This agent reuses multiple skill permission sets but restricts itself to two sub-agents, `academic-writer` and `developer`, demonstrating fine-grained authorization in agent design.

8 Conclusion

This paper reviewed the terminology, file conventions, security safeguards, and practical techniques used to design Skills and Agents in OpenCode (also apply to similar software such as Claude Code, Codex, OpenClaw, etc.). The main rule is simple: a Skill packages a capability; an Agent manages execution, context, permissions, and delegation.

Once that boundary is fixed, naming rules, file structures, and threat-model checks are easier to apply. Variable pre-setting, custom commands, and external-memory mechanisms can improve workflow automation, but they should not be confused with model training. The case studies apply the same checklist to code review, skill generation, academic writing, and personal assistance.

Several limitations remain. First, OpenCode configuration formats and tool names may change across versions, so templates in this paper should be verified against the installed documentation. Second, Skill triggering is not perfectly stable: descriptions, user phrasing, and routing heuristics can cause missed or accidental activation. Third, long-term memory can be polluted by stale, private, or maliciously injected content if persistence is not reviewed. Fourth, tool permissions can be misconfigured, especially when read-only workflows gradually acquire write or shell access. Fifth, MCP servers introduce an additional trust boundary because a server may expose remote state or mutating operations outside OpenCode itself. Finally, path conventions, shell behavior, and available dependencies differ across Linux, macOS, and Windows, so examples should be adapted before deployment.

Future work can proceed in three directions:

(1) Memory and Evolution: Special focus on the memory mechanisms and self-evolution capabilities of skills and agents.

(2) Security Audit Plugin: Develop a unified security audit skill for static analysis of tools/commands, proactively capturing potential risks.

(3) Multimodal Skills: Integrate multimodal inputs including text, images, and audio to further expand OpenCode's applicability in AI-assisted creative scenarios.

References

[1] OpenCode. Opencode official documentation. https://opencode.ai/docs, 2026.

[2] LogNroll Team. OpenCode.ai: The open source ai coding agent revolutionizing development. *LogNroll*, 2026.

[3] Anh Nguyen-Duc, Beatriz Cabrero-Daniel, Adam Przybyłek, Chetan Arora, Dron Khanna, and Tomas Herda. Generative artificial intelligence for software engineering—a research agenda. *Software: Practice and Experience*, 55(11):1806–1843, 2025. doi: 10.1002/spe.70005.

[4] Jing Bi, Ziqi Wang, Haitao Yuan, Junlong Zhang, and Rajkumar Buyya. Large AI models and their applications: Classification, limitations, and potential solutions. *Software: Practice and Experience*, 55, 2025. doi: 10.1002/spe.3408.

[5] Anthropic. Agent skills platform. https://agentskills.io, 2026.

[6] Anthropic. Anthropic skills open source repository. https://github.com/anthropics/skills, 2026.

[7] OpenClaw. Clawhub platform. https://clawhub.ai/, 2026.

[8] Skill.fish. Skill.fish platform. https://www.skill.fish/, 2026.

[9] Vercel. Skills.sh platform. https://skills.sh/, 2026.

[10] Yanna Jiang, Delong Li, Haiyu Deng, Baihe Ma, Xu Wang, Qin Wang, and Guangsheng Yu. Sok: Agentic skills–beyond tool use in llm agents. *arXiv preprint arXiv:2602.20867*, 2026.

[11] Renjun Xu and Yang Yan. Agent skills for large language models: Architecture, acquisition, security, and the path forward. *arXiv preprint arXiv:2602.12430*, 2026.

[12] Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, et al. Large language model agent: A survey on methodology, applications and challenges. *arXiv preprint arXiv:2503.21460*, 2025.

[13] Xiaoxiao Li. When single-agent with skills replace multi-agent systems and when they fail. *arXiv preprint arXiv:2601.04748*, 2026.

[14] David Schmotz, Sahar Abdelnabi, and Maksym Andriushchenko. Agent skills enable a new class of realistic and trivially simple prompt injections. *arXiv preprint arXiv:2510.26328*, 2025. URL https://arxiv.org/abs/2510.26328.

[15] Yanna Jiang, Delong Li, Haiyu Deng, Baihe Ma, Xu Wang, Qin Wang, and Guangsheng Yu. Sok: Agentic skills–beyond tool use in llm agents. *arXiv preprint arXiv:2602.20867*, 2026.

[16] George Ling, Shanshan Zhong, and Richard Huang. Agent skills: A data-driven analysis of claude skills for extending large language model functionality. *arXiv preprint arXiv:2602.08004*, 2026.

[17] Yi Liu, Zhihao Chen, Yanjun Zhang, Gelei Deng, Yuekang Li, Jianting Ning, Ying Zhang, and Leo Yu Zhang. Detecting and understanding malicious agent skills in the wild: A large-scale security empirical study. *arXiv preprint arXiv:2602.06547*, 2026.

[18] David Schmotz, Luca Beurer-Kellner, Sahar Abdelnabi, and Maksym Andriushchenko. Skill-inject: Measuring agent vulnerability to skill file attacks. *arXiv preprint arXiv:2602.20156*, 2026.

[19] Zeyu Zhang, Quanyu Dai, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. A survey on the memory mechanism of large language model-based agents. *ACM Trans. Inf. Syst.*, 43(6), September 2025. ISSN 1046-8188. URL https://doi.org/10.1145/3748302.

[20] Salaheddin Alzubi, Noah Provenzano, Jaydon Bingham, Weiyuan Chen, and Tu Vu. Evoskill: Automated skill discovery for multi-agent systems. *arXiv preprint arXiv:2603.02766*, 2026.

[21] Shangheng Du, Jiabao Zhao, Jinxin Shi, Zhentao Xie, Xin Jiang, Yanhong Bai, and Liang He. A survey on the optimization of large language model-based agents. *ACM Computing Surveys*, 58(9):1–37, 2026.

[22] Zexue He, Yu Wang, Churan Zhi, Yuanzhe Hu, Tzu-Ping Chen, Lang Yin, Ze Chen, Tong Arthur Wu, Siru Ouyang, Zihan Wang, et al. Memoryarena: Benchmarking agent memory in interdependent multi-session agentic tasks. *arXiv preprint arXiv:2602.16313*, 2026.

[23] Jiaqi Liu, Yaofeng Su, Peng Xia, Siwei Han, Zeyu Zheng, Cihang Xie, Mingyu Ding, and Huaxiu Yao. Simplemem: Efficient lifelong memory for llm agents. *arXiv preprint arXiv:2601.02553*, 2026.

[24] Yang Li, Jiaxiang Liu, Yusong Wang, Yujie Wu, and Mingkun Xu. Bmam: Brain-inspired multi-agent memory framework. *arXiv preprint arXiv:2601.20465*, 2026.

[25] Google. Agent memory systems. https://www.skills.sh/sickn33/antigravity-awesome-skills/agent-memory-systems, 2026.

[26] Nous Research. Hermes agent. Online, 2026. URL https://hermes-agent.nousresearch.com/.

[27] Congwei Song. Opencode agents created by the author. https://github.com/Freakwill/opencode-agents, 2026.

[28] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

[29] Andrea Madotto, Zhaojiang Lin, Genta Indra Winata, and Pascale Fung. Few-shot bot: Prompt-based learning for dialogue systems. *arXiv preprint arXiv:2110.08118*, 2021.

[30] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024.

[31] Yangxiao Cai, Peng Liang, Yifei Wang, Zengyang Li, and Mojtaba Shahin. Demystifying issues, causes and solutions in LLM open-source projects.

Journal of Systems and Software, 227:112452, 2025. doi: 10.1016/j.jss.2025.112452.