

GitHub Copilot Automation Cheatsheet

A guide for setting up
agent-driven development
environments with custom
agents, skills, and organized
project resources.



Table of contents

1. Introduction	01
1.1 The Promise of Agent-Driven Development	01
1.2 Why Agent Pipelines Matter	01
1.3 The Knowledge Stack: Foundation for High Efficiency	02
1.4 The Efficiency Multiplier	04
1.5 Getting Started	04
2. What is a Custom Agent	05
2.1 Custom Agent vs Copilot Instructions	05
3. What is a Skill	07
4. Directory Structure	08
4.1 Where to Put Agents	08
4.2 Where to Put Skills	09
5. Project Principles	10
5.1 The Role of Principles in Agent Pipelines	10
6. References	13
6.1 The Role of References in Agent Pipelines	13
7. Specs	16
7.1 The Role of Specs in Agent Pipelines	16
8 Skills for Resource Access	18
8.1 Read Principles Skill	18
8.2 Read References Skill	19



Table of contents

8.3 Read Specs Skill	19
8.4 Read Compound Skill	20
9 Documentation Access via MCP	22
9.1 Why MCP for Documentation?	22
9.2 Architecture Overview	23
9.3 Using Copilot to Generate the MCP Server	24
9.3.1 What to Include in Your Prompt	24
9.3.2 Example Prompt	24
9.3.3 What to Expect	26
9.3.4 After Generation	26
9.4 VS Code Configuration	27
9.5 Using MCP Tools in Skills	27
9.6 Maintenance	28
10 Agent Pipeline	29
10.1 Pipeline Agents Overview	29
10.2 Why Separate Agents?	31
10.3 The Human-in-the-Loop Principle	31
10.4 Critical Checkpoints	32
10.4.1 After Task Breakdown: Validate Task Quality	32
10.4.2 After Plan Generation: Validate Plan Completeness	32
10.5 Loading Compound Knowledge	33



Table of contents

10.6 1. Task Breakdown Agent	33
10.6.1 Role: Specification Analyst	33
10.6.2 Responsibility	33
10.6.3 First Step: Load Compound Knowledge	33
10.6.4 How to Invoke	34
10.6.5 Expected Output	35
10.7 2. Plan Generation Agent	38
10.7.1 Role: Implementation Architect	38
10.7.2 Responsibility	39
10.7.3 First Step: Load Compound Knowledge	39
10.7.4 How to Invoke	39
10.7.5 Expected Output	40
10.8 3. Plan Review Agent	44
10.8.1 Role: Quality Gatekeeper	44
10.8.2 Responsibility	44
10.8.3 First Step: Load Compound Knowledge	45
10.8.4 How to Invoke	45
10.8.5 Expected Output	46
10.9 4. Implementation Agent	50
10.9.1 Role: Execution Engineer	50
10.9.2 Responsibility	50



Table of contents

10.9.3 First Step: Load Compound Knowledge	51
10.9.4 How to Invoke	51
10.9.5 Expected Output	51
10.10 5. Implementation Review Agent	56
10.10.1 Role: Code Quality Guardian	56
10.10.2 Responsibility	56
10.10.3 First Step: Load Compound Knowledge	57
10.10.4 How to Invoke	57
10.10.5 Expected Output	58
10.11 6. Compound Agent	62
10.11.1 Role: Knowledge Curator	62
10.11.2 Responsibility	62
10.11.3 When to Use	63
10.11.4 How to Invoke	63
10.11.5 Expected Output	64
10.12 Session Management: One Agent, One Session	68
11 Complete Example Structure	70
12 Quick Reference	72
12.1 Agent File Template	72
12.2 Skill File Template	72
13 Extending the Agent Pipeline	74
14 Further Reading	75



1. Introduction

1.1 The Promise of Agent-Driven Development

Modern software development increasingly relies on AI-assisted workflows, but **generic AI assistants often produce inconsistent, context-poor results**. They don't know your coding standards, your architecture decisions, or your business requirements. Every interaction starts from scratch, and the AI must be re-educated about your project's conventions.

Agent pipelines solve this problem by creating a structured, knowledge-rich environment where AI assistants operate with deep project context. Instead of a single generic assistant, you deploy specialized agents — each with focused expertise — that work together in a systematic workflow from requirements to reviewed implementation.

1.2 Why Agent Pipelines Matter

A well-defined agent pipeline transforms AI assistance from an ad-hoc tool into a **reliable engineering process**:

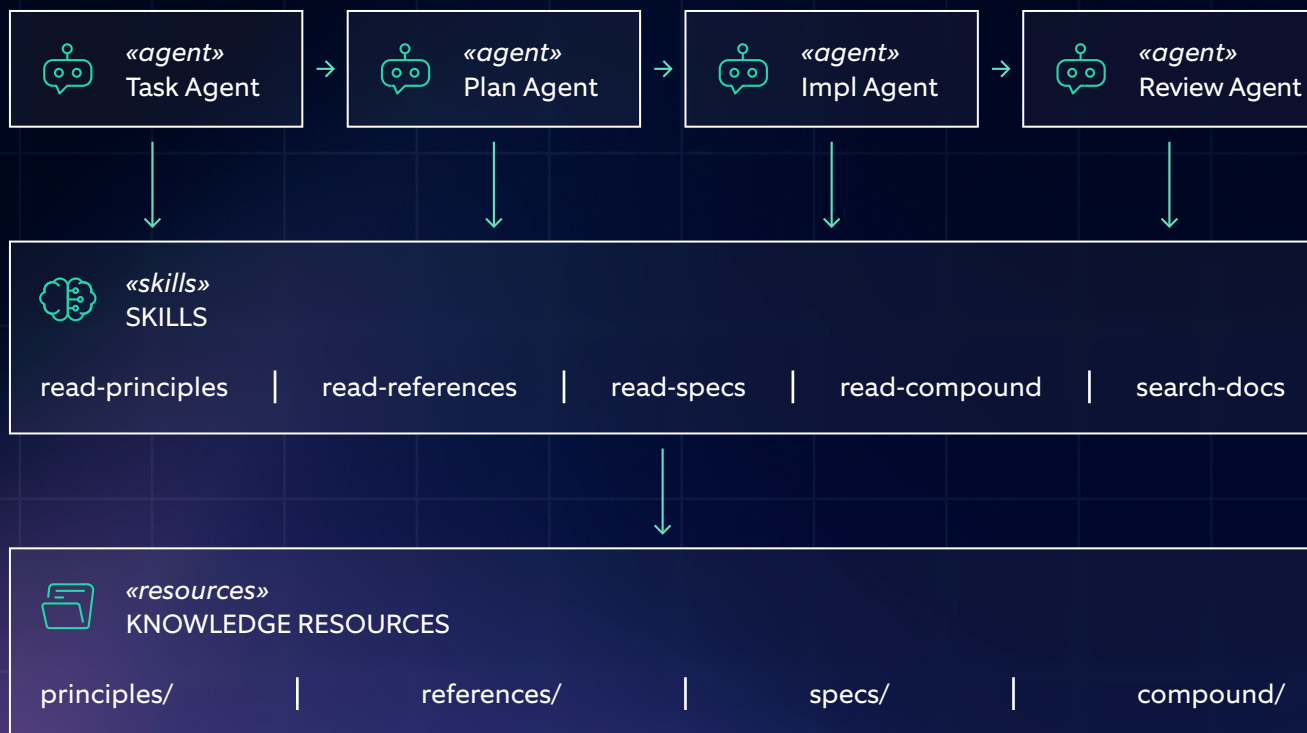
Without Pipeline	With Agent Pipeline
Generic AI guesses at conventions	Agents load and follow your principles
Inconsistent code quality	Every output validated against standards
Lost context between sessions	Compound files preserve lessons learned
No traceability to requirements	Plans and implementations trace to specs
Manual review catches all issues	Review agents fix issues automatically
Knowledge stays in developers' heads	Knowledge encoded in skills and compounds



1.3 The Knowledge Stack: Foundation for High Efficiency

Agent pipelines achieve high efficiency through a **layered knowledge stack** that provides context at every stage:

«pipeline» **AGENT PIPELINE**





Each layer serves a critical purpose:

1. Principles The “Law of the Land”

Coding standards, architecture patterns, error handling policies, and security guidelines that **every agent must follow**. Without principles, agents make inconsistent decisions that create technical debt.

Example impact: A Plan Generation Agent consulting `principles/error-handling.md` will specify try-catch patterns consistently across all plans, rather than inventing different approaches each time.

2. References Technical Knowledge Base

Framework documentation, API guides, database schemas, and integration patterns that agents need for **technically correct implementations**. Without references, agents use outdated patterns or incorrect API calls.

Example impact: An Implementation Agent consulting `references/react-patterns.md` will use your project’s established component patterns rather than generic React code.

3. Specs Business Requirements

Feature specifications, acceptance criteria, user stories, and API contracts that define **what the system should do**. Without specs, agents guess at business logic and miss requirements.

Example impact: A Task Breakdown Agent consulting `specs/features/user-auth.md` will identify all authentication requirements — OAuth, password reset, session management — rather than implementing only the obvious ones.

4. Compound Files Accumulated Wisdom

Lessons learned from previous agent sessions — mistakes that were corrected, patterns that worked, pitfalls to avoid. Without compounds, **the same mistakes repeat** across sessions.

Example impact: Before creating a database migration plan, the Plan Agent loads `compound/migration-conventions.md` and learns that all migrations must be reversible — avoiding a lesson that was learned the hard way in a previous session.

5. Skills Knowledge Access Layer

Instructions that tell agents **how and when** to load knowledge resources. Skills bridge agents to the knowledge stack, ensuring consistent access patterns.

Example impact: The `read-principles` skill tells every agent to load coding standards before writing code — ensuring no agent skips this critical step.



1.4 The Efficiency Multiplier

When all components work together, efficiency compounds:

1. **Task Breakdown Agent** loads specs → produces accurate, complete task lists
2. **Plan Generation Agent** loads principles + references → produces technically sound plans
3. **Implementation Agent** follows plan + loads compound lessons → avoids known pitfalls
4. **Review Agent** validates against principles → catches violations automatically
5. **Compound Agent** captures new lessons → prevents future mistakes

Each stage **builds on the previous**, and each agent **benefits from accumulated knowledge**. The result is a development process where:

- **Quality is consistent** — Every output meets the same standards
- **Context is preserved** — Knowledge doesn't leave with team members
- **Mistakes compound into wisdom** — Each correction improves future work
- **Human review is focused** — Engineers validate decisions, not catch basic errors

1.5 Getting Started

This cheatsheet provides everything you need to build an effective agent pipeline:

1. **Understand the components** — What agents, skills, and knowledge resources do
2. **Set up the directory structure** — Where to put each component
3. **Create knowledge resources** — Principles, references, specs, and skills
4. **Define your agents** — Specialized assistants for each pipeline stage
5. **Establish the workflow** — How agents hand off work to each other

The following sections detail each component with examples and templates you can adapt to your project.



2. What is a Custom Agent

A **custom agent** is a specialized Copilot persona with tailored expertise for specific development tasks. Agents are Markdown files with YAML frontmatter that define:

- **Identity:** Name and description
- **Capabilities:** Which tools the agent can access
- **Behavioral instructions:** Domain expertise and workflow guidance

Agents allow you to create focused AI assistants for specific roles (e.g., testing specialist, security reviewer, implementation planner) rather than using a generic assistant.

Key characteristics:

- Markdown file with `.agent.md` extension
- YAML frontmatter for configuration
- Prompt body defines behavior and expertise
- Can be scoped to repository, organization, or enterprise level

2.1 Custom Agent vs Copilot Instructions

Aspect	Custom Agent (.agent.md)	Copilot Instructions (copilot-instructions.md)
Purpose	Defines a specialized persona with specific expertise for targeted tasks	Defines global rules/preferences that apply to ALL Copilot interactions
Invocation	Explicitly invoked by user with <code>@agent-name</code>	Always active — automatically loaded for every request
Scope	Task-specific (e.g., “security reviewer”, “test writer”)	Project-wide coding standards and conventions
Tools	Can restrict/allow specific tools via <code>tools:</code> frontmatter	No tool restrictions — applies to default Copilot
Model	Can specify a different model via <code>model:</code> frontmatter	Uses default model
Content	Detailed workflow instructions, skill references, specialized prompts	Brief rules: naming conventions, formatting, error handling



When to use which:

Use Copilot Instructions for rules that should apply to every interaction:

- “Use camelCase for variables”
- “Always add JSDoc comments”
- “Prefer async/await over callbacks”

Use Custom Agents for specialized workflows that need focused expertise:

- A “security-reviewer” agent that knows your security checklist
- A “test-writer” agent that follows your testing patterns
- A “plan-generator” agent that produces structured implementation plans

In short: Instructions set the baseline rules; Agents define specialized personas.



3. What is a Skill

A **skill** is a folder of instructions, scripts, and resources that Copilot loads when relevant to improve performance on specialized tasks. Unlike agents (which define personas), skills define **how to do specific things**.

Key characteristics: - Directory containing a SKILL.md file - YAML frontmatter with name, description, and optional `allowed-tools` - Markdown body with detailed instructions - Can include scripts, examples, and supplementary resources - **Invoked automatically** when task matches skill description.

How skills are invoked: Copilot decides when to use skills based on the prompt and the skill's description field. There is no `skills:` property in agent frontmatter — instead, agents reference skills in their **prompt instructions** (markdown body) to guide when they should be used.

Skills vs Custom Instructions: - Use **custom instructions** for simple rules relevant to almost every task (coding standards) - Use **skills** for detailed instructions Copilot should only access when relevant



4. Directory Structure

4.1 Where to Put Agents

Scope	Location
Project (single repo)	<code>.github/agents/</code> or <code>.agents/</code>
Organization (all repos)	<code>.github</code> or <code>.github-private</code> repo → <code>agents/</code>
Enterprise (all orgs)	designated <code>.github-private</code> repo → <code>agents/</code>
Personal (local)	<code>~/.copilot/agents/</code> or <code>~/.agents/</code>

File naming: `{agent-name}.agent.md` — lowercase with hyphens, e.g., `task-breakdown.agent.md`





4.2 Where to Put Skills

Scope	Location
Project	<code>.github/skills/</code> , <code>.claude/skills/</code> , or <code>.agents/skills/</code>
Personal	<code>~/.copilot/skills/</code> or <code>~/.agents/skills/</code>

Structure: Each skill gets its own subdirectory:

```
.github/skills/  
|-- code-review/  
|   |-- SKILL.md  
|-- documentation-writer/  
|   |-- SKILL.md  
|   |-- templates/  
|-- testing-helper/  
|   |-- SKILL.md  
|   |-- test-template.sh
```



5. Project Principles

5.1 The Role of Principles in Agent Pipelines

Principles are the **foundation of consistent, high-quality AI-assisted development**. They serve as the authoritative source of truth that every agent must consult before making decisions about code structure, patterns, error handling, and architecture.

Why Principles Are Critical:

Without documented principles, AI agents face a fundamental problem: **they have no way to know your team's conventions**. Each agent session starts fresh, and without explicit guidance, the AI will:

- Invent naming conventions that clash with existing code
- Use error handling patterns inconsistent with the codebase
- Make architecture decisions that violate established patterns
- Generate code that “works” but doesn’t fit your project’s style

Principles solve this by providing **explicit, documented standards** that agents load at the start of every session.

The Responsibility of Principles:

Responsibility	Impact
Define the “right way”	Agents don’t guess – they follow your documented standards
Ensure consistency	Code from different sessions and agents looks like it was written by one team
Reduce review burden	Fewer style and convention issues to catch in code review
Onboard new team members	Both humans and AI agents learn the same conventions from the same source
Preserve institutional knowledge	Decisions and rationale are documented, not lost when team members leave



Principles vs. Other Knowledge Resources:

- **Principles** = “How we do things” (conventions, standards, patterns)
- **References** = “How the technology works” (framework docs, API guides)
- **Specs** = “What we need to build” (requirements, acceptance criteria)

Every agent should load principles **before** writing any code. The read-principles skill ensures this happens consistently.

Location: `.github/principles/` or `docs/principles/`

Principles define the **coding guidelines and overall standards** that all agents should follow. This is the “law of the land” for your codebase.

What to include: - Coding standards (naming conventions, formatting rules) - Architecture decisions and patterns - Error handling policies - Logging conventions - Testing requirements and patterns - Security guidelines - Performance expectations

Example structure:

```
principles/
|-- coding-standards.md      # Language-specific conventions
|-- architecture.md         # System design patterns
|-- testing.md              # Test coverage requirements
|-- security.md             # Security best practices
`-- error-handling.md       # Exception and error patterns
```

Example “coding-standards.md”:

```
# Coding Standards

## General Principles

- Write code for the next developer – favor clarity over cleverness
- Favor early return for errors to avoid deep nesting
- Delete dead code; do not comment it out
- **Don't Repeat Yourself (DRY)** – eliminate code duplication through
  ↳ functions and abstractions
```



(continued from previous page)

SOLID Principles

Follow SOLID principles for object-oriented design:

Single Responsibility Principle (SRP)

- Each class should have one reason to change
- Split classes that handle multiple concerns

Open/Closed Principle (OCP)

- Classes should be open for extension but closed for modification
- Use inheritance and composition to add behavior without changing existing code

Dependency Inversion Principle (DIP)

- Depend on abstractions (interfaces), not concrete implementations
- Use dependency injection to provide concrete implementations at runtime

Naming Conventions

Context	Convention	Example
Variables	camelCase	`userName`, `itemCount`
Functions	camelCase	`getUserById()`, `calculate()`
Classes/Types	PascalCase	`UserService`, `OrderItem`
Interfaces	`I` prefix	`IUserRepository`
Constants	UPPER_SNAKE	`MAX_RETRIES`, `API_URL`

File Organization

- One component/class per file
- Co-locate tests with implementation (`\*.test.ts` or `test/` folder)
- Group related files in feature folders

Error Handling

- Always use typed errors with meaningful messages
- Log errors with context before re-throwing
- Never swallow exceptions silently
- Use early return to handle error cases first

Comments and Documentation

- Use JSDoc/Doxygen-style comments for public APIs
- Document **why**, not **what** – code shows what, comments explain why
- Keep comments up to date with code changes



6. References

6.1 The Role of References in Agent Pipelines

References are the **technical knowledge** base that enables agents to write correct, idiomatic code for your specific technology stack. They bridge the gap between generic AI knowledge and your project's actual implementation patterns.

Why References Are Critical:

AI models have broad knowledge of programming languages and frameworks, but this knowledge has limitations:

- **Training cutoff** – Models may not know the latest API changes or deprecations
- **Generic patterns** – Models default to textbook examples, not your project's conventions
- **Missing context** – Models don't know which libraries you've chosen or how you've configured them
- **Version mismatches** – Models may suggest code for different library versions

References solve this by providing **project-specific technical documentation** that agents consult when implementing features.

Responsibility	Impact
Document your tech stack	Agents know which frameworks, libraries, and tools your project actually uses
Capture API patterns	Agents use correct method signatures, not outdated or incorrect API calls
Define integration points	Agents understand how components connect and communicate in your architecture
Preserve tribal knowledge	Undocumented patterns and workarounds are captured for all agents to use
Enable correct imports	Agents know the exact import paths and module structures in your codebase



References vs. Other Knowledge Resources:

- **Principles** = “How we do things” (conventions, standards, patterns)
- **References** = “How the technology works” (framework docs, API guides)
- **Specs** = “What we need to build” (requirements, acceptance criteria)

When an agent needs to implement a feature, it consults references to understand the **technical how** – which APIs to call, which patterns to follow, and which integration points to use.

Location: .github/references/ or docs/references/

References contain **framework and library documentation** used in the project. This gives agents context about the technologies you’re using.

What to include:

- Framework guides and patterns (React, FastAPI, etc.)
- Library API summaries
- Internal tooling documentation
- Integration guides
- Configuration references

Example structure:

```
references/  
|-- react-patterns.md           # React-specific patterns  
|-- fastapi-guide.md           # Backend framework reference  
|-- database-schema.md         # Data model documentation  
|-- api-conventions.md         # REST/GraphQL conventions  
`-- deployment-guide.md        # CI/CD and deployment
```

Example ``pubsub-topics.md``:

```
# PubSub Topics Reference  
  
➤ Last updated: 2026-04-02 | Sources: `frontend-framework`, `ui-components`  
  
## PubSub API
```



(continued from previous page)

```
``typescript
interface IPubSub {
  subscribe(topic: string, handler: (message: object) => void): symbol;
  unsubscribe(id: symbol): void;
  publish(topic: string, message: object): void;
}
``
```

****Access:**** ``usePubSubService()`` hook from ``@app/context``

Topic Categories

Authentication & Session

Topic	Description
<code>`RequireLogin`</code>	Login required (session expired or initial)
<code>`RequireLoginDismissed`</code>	Login dialog dismissed
<code>`UserLoggedOut`</code>	User logged out – cleanup completed

System State

Topic	Description
<code>`SystemStateUpdates`</code>	System state changes
<code>`SafetyConditionsUpdates`</code>	Safety condition state changes

Navigation

Topic	Description
<code>`Navigate`</code>	Navigation request to shell
<code>`ChangeLanguage`</code>	Language switch request

UI Dialogs

Topic	Description
<code>`CloseWorkflow`</code>	Close workflow overlay
<code>`InlineFeedback`</code>	Component-level feedback display



7. Specs

7.1 The Role of Specs in Agent Pipelines

Specs are the **source of truth for what the system should do**. They define requirements, acceptance criteria, and business rules that agents must implement – ensuring AI-generated code actually solves the right problems.

Why Specs Are Critical:

Without specifications, AI agents face a fundamental question: **“What should this feature actually do?”** Generic prompts like “implement user authentication” leave critical decisions undefined:

- Which authentication methods are required?
- What are the password complexity rules?
- How should sessions be managed?
- What happens when authentication fails?

Agents will fill these gaps with assumptions – and assumptions lead to features that don’t match business needs, missed requirements, and rework.

Specs solve this by providing **explicit requirements** that agents consult before and during implementation.

The Responsibility of Specs:

Responsibility	Impact
Define what to build	Agents implement specified behavior, not invented behavior
Capture acceptance criteria	Agents know exactly when a feature is “done” and can verify completeness
Document edge cases	Agents handle boundary conditions and error scenarios that would otherwise be missed
Preserve business rules	Complex business logic is documented, not guessed or hard-coded incorrectly
Enable traceability	Every implementation can be traced back to a requirement



Specs vs. Other Knowledge Resources:

- **Principles** = “How we do things” (conventions, standards, patterns)
- **References** = “How the technology works” (framework docs, API guides)
- **Specs** = “What we need to build” (requirements, acceptance criteria)

The Task Breakdown Agent and Plan Generation Agent rely heavily on specs to produce accurate, complete task lists and implementation plans.

Location: `.github/specs/` or `docs/specs/`

Specs contain **project business and technical documentation** – what the system should do and the technical constraints it must satisfy. This includes both the business requirements (what users need) and technical specifications (how the system is designed to meet those needs).

What Specs Should Cover:

- **Business documentation** – Feature specifications, user stories, acceptance criteria, business rules, and workflow definitions
- **Technical documentation** – System architecture decisions, API contracts, data models, integration specifications, and technical constraints

What to include: - Feature specifications - Business requirements - User stories - Acceptance criteria
- System constraints - API contracts

Example structure:

```
specs/  
|-- features/  
|   |-- user-authentication.md  
|   |-- payment-processing.md  
|   `-- notification-system.md  
|-- api-contracts/  
|   |-- user-api.md  
|   `-- order-api.md  
`-- requirements/  
    |-- functional-requirements.md  
    `-- non-functional-requirements.md
```



8. Skills for Resource Access

Create skills that give agents access to your principles, references, and specs. This ensures consistent knowledge across all agent interactions.

8.1 Read Principles Skill

Location: `.github/skills/read-principles/SKILL.md`

```
---
name: read-principles
description: Load project coding principles and standards. Use when
  ↳ implementing features, reviewing code, or making architectural decisions.
---

# Read Project Principles

Before implementing or reviewing code, read the relevant principles from the
↳ `principles/` directory:

1. **Always read first:**
  ↳ `principles/coding-standards.md` – naming, formatting, file organization
  ↳ `principles/error-handling.md` – exception patterns

2. **Read when relevant:**
  ↳ `principles/testing.md` – when writing or reviewing tests
  ↳ `principles/security.md` – when handling auth, data, or external input
  ↳ `principles/architecture.md` – when making structural decisions

Apply these principles consistently. If implementation conflicts with
↳ principles, flag it for discussion rather than silently violating.
```



8.2 Read References Skill

Location: .github/skills/read-references/SKILL.md

```
---
name: read-references
description: Load framework and library documentation. Use when working with
  ↳ specific technologies or integrations.
---
```

Read Project References

When working with project technologies, consult the appropriate reference:

1. ****Identify the technology**** involved in the task
2. ****Read the relevant reference**** from `references/`:
 - `references/react-patterns.md` – frontend components
 - `references/fastapi-guide.md` – backend endpoints
 - `references/database-schema.md` – data operations
 - `references/api-conventions.md` – API design
3. ****Follow documented patterns**** rather than inventing new approaches

If a pattern isn't documented, check if similar patterns exist elsewhere in
↳ the codebase before creating something new.

8.3 Read Specs Skill

Location: .github/skills/read-specs/SKILL.md

```
---
name: read-specs
description: Load feature specifications and business requirements. Use when
  ↳ implementing features, writing tests, or validating behavior.
---
```

Read Project Specifications

Before implementing a feature or writing tests:

1. ****Locate the relevant spec**** in `specs/features/`
2. ****Read the full specification****, including:
 - Functional requirements
 - Acceptance criteria



(continued from previous page)

- Edge cases and constraints
- API contracts (if applicable)

3. ****Trace implementation back to specs**** – every behavior should map to a requirement

When specs are unclear or missing, flag the gap rather than making assumptions.
↳ about business logic.

8.4 Read Compound Skill

Location: `.github/skills/read-compound/SKILL.md`

```
---
name: read-compound
description: Load lessons learned from previous agent interactions. Use at
↳ the start of any agent session to avoid repeating past mistakes.
---

# Read Compound Knowledge

Before starting any work, search for relevant compound files:

1. **Search compound directory** – Look in `docs/compound/` for files related
↳ to the current task
2. **Match by topic** – Read file descriptions in YAML frontmatter to find
↳ relevant lessons
3. **Load applicable files** – Read compound files that match the current work
↳ domain

## Compound File Structure

Each compound file has YAML frontmatter with:
- `title` – Auto-generated descriptive title
- `description` – Summary of what lessons are captured
- `created` – When the lesson was captured
- `source-agent` – Which agent session generated the lesson
- `tags` – Keywords for discovery
```



(continued from previous page)

When to Load

- **Task Breakdown**: Load compounds related to specification analysis, task_→estimation
- **Plan Generation**: Load compounds related to architecture, patterns, file_→organization
- **Plan Review**: Load compounds related to review criteria, common plan_→issues
- **Implementation**: Load compounds related to coding patterns, error_→handling, testing
- **Implementation Review**: Load compounds related to code quality, review_→findings

Apply lessons from compound files to avoid repeating past mistakes. If the_→current work contradicts a compound lesson, flag it for discussion.



9. Documentation Access via MCP

For larger projects, create **MCP (Model Context Protocol) servers** to index and search documentation. This is essential when documentation exceeds what can fit in context — MCP lets Copilot query thousands of document sections on demand.

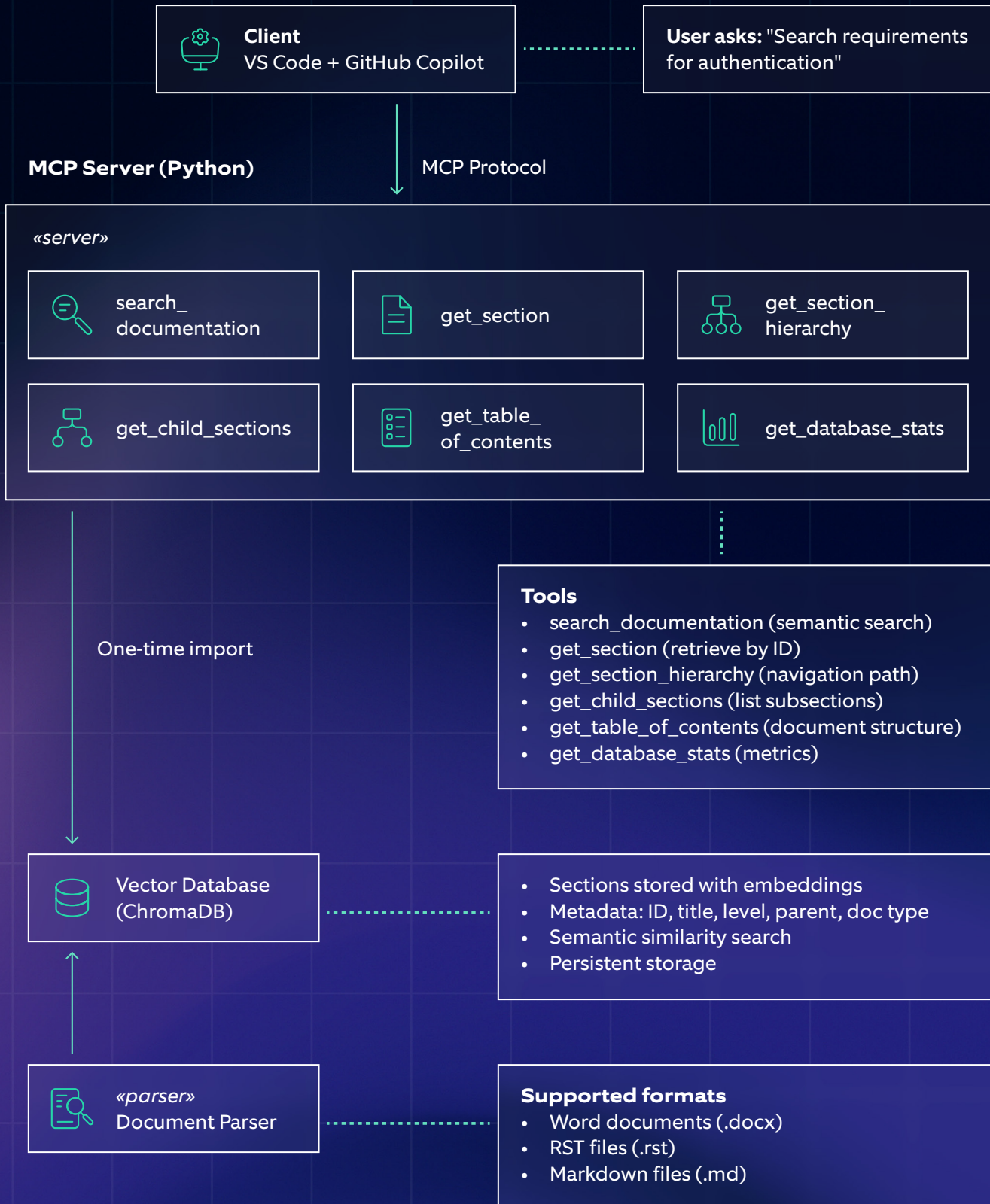
9.1 Why MCP for Documentation?

Challenge	MCP Solution
Documentation too large for context	Query only relevant sections on demand
Natural language questions	Semantic search finds related content
Multiple document types	Single interface for all documentation
Keeping docs current	Re-index when documentation changes
Navigating structure	Hierarchical browsing and table of contents





9.2 Architecture Overview





9.3 Using Copilot to Generate the MCP Server

You don't need to write the MCP server code yourself — **Copilot can generate it for you**. Provide a detailed prompt describing your documentation structure and requirements, and Copilot will create the complete implementation.

9.3.1 What to Include in Your Prompt

When asking Copilot to generate an MCP documentation server, include:

1. **Documentation types** — What formats do you have? (Word, Markdown, RST, etc.)
2. **Document locations** — Where are the files stored?
3. **Search requirements** — What kinds of queries do users need?
4. **Metadata needs** — What information should be stored with each section?
5. **Deployment preference** — Docker, local Python, or both?

9.3.2 Example Prompt

```
Create an MCP server for GitHub Copilot that indexes and searches my project_
→documentation.
```

```
## Documentation Structure
```

```
My documentation is organized as follows:
```

- `docs/requirements/` — Markdown files with product requirements (REQ-001_
→format IDs)
- `docs/design/` — RST files with technical design documents
- `docs/api/` — OpenAPI specs and API documentation

```
## Required MCP Tools
```

```
The server should provide these tools to Copilot:
```

1. ****search_documentation****
 - Semantic search across all documentation
 - Filter by document type (requirements, design, api, or all)
 - Return top N results with relevance scores
 - Include section ID, title, and content snippet in results



(continued from previous page)

2. `**get_section**`

- Retrieve full content of a section by ID
- Include parent/child relationships
- Return metadata (doc type, file path, last modified)

3. `**get_table_of_contents**`

- List document structure
- Filter by document type
- Control depth of hierarchy returned

4. `**list_by_prefix**`

- Find all sections matching an ID prefix (e.g., „REQ-01“ returns REQ-010, REQ-011, etc.)

Technical Requirements

- Use Python with the `mcp` SDK
- Use ChromaDB for vector storage with sentence-transformers embeddings
- Support both HTTP/SSE mode (for Docker) and stdio mode (for local dev)
- Parse Markdown with front-matter support
- Parse RST preserving section hierarchy
- Batch import with progress logging
- Persistent database storage

Deployment

- Provide Dockerfile for containerized deployment
- Provide docker-compose.yml for easy startup
- Include VS Code mcp-settings.json configuration examples

Output Structure

Create these files:

- mcp_server.py – Main MCP server with tool definitions
- vector_db.py – ChromaDB wrapper with search methods
- document_parser.py – Parsers for each document format
- requirements.txt – Python dependencies
- Dockerfile – Container build
- docker-compose.yml – Container orchestration
- README.md – Setup and usage instructions



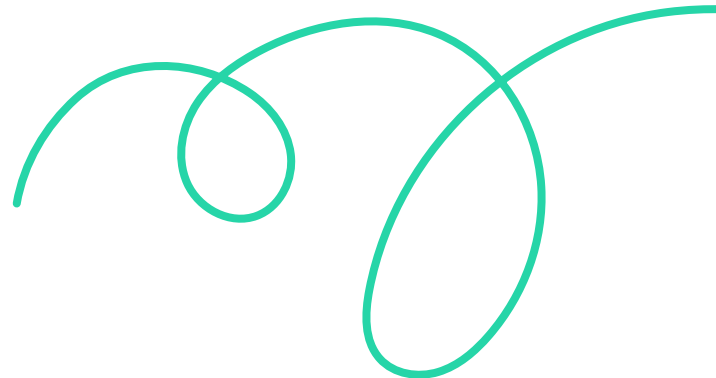
9.3.3 What to Expect

When you provide a prompt like this, Copilot will generate:

File	Purpose
mcp_server.py	MCP server with <code>@app.list_tools()</code> and <code>@app.call_tool()</code> handlers
vector_db.py	ChromaDB wrapper with <code>add_sections()</code> , <code>search()</code> , and <code>get_section()</code> methods
document_parser.py	Parsers for your document formats with section extraction
requirements.txt	Dependencies: mcp, chromadb, sentence-transformers, etc.
Dockerfile	Python container with all dependencies
docker-compose.yml	Service definition with volume mounts
README.md	Setup instructions and usage examples

9.3.4 After Generation

1. **Review the generated code** — Verify it handles your document formats correctly
2. **Test locally** — Run the import and verify sections are indexed
3. **Configure VS Code** — Add the MCP server to your settings
4. **Iterate** — Ask Copilot to add features or fix issues





9.4 VS Code Configuration

After generating and deploying your MCP server, configure VS Code:

HTTP/SSE mode (Docker deployment):

```
// .vscode/mcp-settings.json
{
  "mcpServers": {
    "docs-server": {
      "url": "http://localhost:5143"
    }
  }
}
```

stdio mode (local Python):

```
// .vscode/mcp-settings.json
{
  "mcpServers": {
    "docs-server": {
      "command": "python",
      "args": ["/mcp-docs-server/mcp_server.py"]
    }
  }
}
```

9.5 Using MCP Tools in Skills

Once your MCP server is running, create a skill that tells agents how to use it:

Location: `.github/skills/search-docs/SKILL.md`

```
---
name: search-docs
description: Search project documentation using the MCP server. Use when you
  ↳ need requirements, design decisions, or specifications.
---

# Search Project Documentation

Use the `docs-server` MCP tools to find relevant documentation.
```



(continued from previous page)

Available Tools

- **search_documentation** – Semantic search across all docs
- **get_section** – Retrieve a specific section by ID
- **get_table_of_contents** – Browse document structure
- **list_by_prefix** – Find sections by ID prefix

Best Practices

1. **Search first** – Don't assume something isn't documented
2. **Be specific** – „user authentication timeout“ > „auth“
3. **Check hierarchy** – Use get_table_of_contents to understand structure
4. **Verify IDs** – Section IDs appear in search results

9.6 Maintenance

Task	Command
Start server	<code>docker-compose up -d</code>
View logs	<code>docker-compose logs -f</code>
Re-import docs	<code>python vector_db.py --reimport</code>
Check status	<code>docker-compose ps</code>
Stop server	<code>docker-compose down</code>



10. Pipeline Agents Overview

Implement a multi-agent pipeline where each agent has a specific role. This creates a systematic workflow from requirements to reviewed implementation.

10.1 Pipeline Agents Overview

Agent	Responsibility
Task Breakdown Agent	Analyzes feature specifications and decomposes them into discrete, implementable tasks with estimates, dependencies, and acceptance criteria
Plan Generation Agent	Takes a task and creates a detailed implementation blueprint with file paths, patterns to follow, and test scenarios
Plan Review Agent	Validates plans for completeness, correctness, and adherence to coding principles before implementation begins
Implementation Agent	Executes the approved plan — writes code, creates tests, and documents what was done
Implementation Review Agent	Reviews completed code against the plan and principles, finding and fixing issues directly
Compound Agent	Captures lessons learned from agent sessions and creates reusable knowledge files to prevent repeated mistakes





10.2 Why Separate Agents?

Each agent is designed with **single responsibility** — one type of expertise, one type of action. This separation provides several benefits:

Benefit	Description
Focused expertise	Each agent excels at one task rather than being mediocre at many
Better prompts	Smaller, focused prompts produce higher quality outputs than large multipurpose prompts
Clear handoffs	Explicit artifacts (task files, plans, summaries) create checkpoints for validation
Human oversight	Engineers can review and approve at each stage before proceeding
Easier debugging	When something goes wrong, you know which agent and which stage to investigate
Iterative improvement	You can refine one agent without affecting others

10.3 The Human-in-the-Loop Principle

Agents assist; engineers validate. Each stage produces an artifact that the engineer should review:

```
Specs → [Breakdown Agent] → Task Files ← ENGINEER REVIEW
      ↓
Task → [Plan Agent] → Plan File ← ENGINEER REVIEW
      ↓
Plan → [Implementation Agent] → Code + Summary ← ENGINEER REVIEW
      ↓
Code → [Review Agent] → Fixes ← ENGINEER REVIEW
```



10.4 Critical Checkpoints

10.4.1 After Task Breakdown: Validate Task Quality

The **Task Breakdown Agent** translates specifications into discrete tasks. This is a critical checkpoint because:

- **Garbage in, garbage out** — Poor task definitions lead to poor plans and implementations
- **Missing tasks** cause incomplete features
- **Overly vague tasks** force the planning agent to guess
- **Wrong dependencies** create implementation blockers

Engineer responsibilities:

1. Review each task for **completeness**—Does it cover all spec requirements?
2. Verify **granularity** — Are tasks small enough to implement (1-4 hours) but not too fragmented?
3. Check **dependencies** — Are task dependencies correctly identified?
4. Validate **acceptance criteria** — Is it clear when each task is “done”?
5. Ensure **technical accuracy** — Are the file paths and technical details correct?

Don't proceed to planning until task files accurately represent the work.

10.4.2 After Plan Generation: Validate Plan Completeness

The **Plan Generation Agent** compiles detailed implementation plans. This is equally critical because:

- The plan is the **blueprint** — Implementation follows the plan, not the original spec
- **Missing steps** become missing features
- **Wrong patterns** propagate through the entire implementation
- **Incomplete test scenarios** lead to untested edge cases

Engineer responsibilities:

1. Review **all implementation steps** — Is every action needed for the task documented?
2. Verify **file paths** — Are the correct files being created/modified?
3. Check **patterns** — Do the referenced patterns match your codebase conventions?
4. Validate **test scenarios** — Are happy paths, edge cases, and error cases covered?
5. Assess **risks** — Are risks properly identified with realistic mitigations?

AI-assisted review: Use the Plan Review Agent to get a second opinion, but don't skip your own review. The review agent catches structural issues; you catch domain-specific problems.



10.5 Loading Compound Knowledge

Every agent should start by loading relevant compound files. Compound files capture lessons learned from previous agent interactions — mistakes that were corrected, patterns that worked, pitfalls to avoid.

At the beginning of each agent session:

1. Invoke the `read-compound` skill
2. Search docs/compound/ for files matching the current task domain
3. Load and apply any relevant lessons before proceeding

This ensures the team's accumulated knowledge compounds over time rather than being lost between sessions.

10.6 1. Task Breakdown Agent

10.6.1 Role: Specification Analyst

The Task Breakdown Agent is your **first point of contact** when starting new work. It reads feature specifications and transforms them into discrete, implementable tasks that can be assigned, estimated, and tracked.

10.6.2 Responsibility

This agent is responsible for:

Reading and understanding feature specifications from specs/features/

- **Decomposing features** into tasks that are small enough to implement (1-4 hours)
- **Identifying dependencies** between tasks
- **Defining acceptance criteria** for each task
- **Mapping tasks to files** that will be created or modified
- **Flagging risks and unknowns** that need investigation

The agent does NOT implement code, create plans, or make architectural decisions. Its sole focus is creating a clear, actionable task list.

10.6.3 First Step: Load Compound Knowledge

Before analyzing specifications, invoke the `read-compound` skill to load lessons from docs/compound/ related to task breakdown, specification analysis, and effort estimation.



10.6.4 How to Invoke

Example Prompt:

```
@task-breakdown Break down the user authentication feature.
```

```
The spec is located at specs/features/user-authentication.md
```

```
This feature needs to support:
```

- Email/password login
- OAuth **with** Google **and** GitHub
- Password reset flow
- Remember me functionality

```
Please identify all tasks needed to implement this feature, estimate ↪ effort for each, and flag any technical risks.
```

What to include in your prompt: - Reference to the specification file - Brief summary of the feature scope - Any specific areas of concern - Constraints or requirements not in the spec



10.6.5 Expected Output

The agent generates a markdown file at `docs/tasks/{feature-name}-breakdown.md` containing:

Section	Content
Overview	Summary of the feature and scope boundaries
Prerequisites	Dependencies that must exist before starting
Tasks	Numbered list with: description, effort estimate (S/M/L), dependencies, acceptance criteria, files to modify
Risks and Unknowns	Technical risks with mitigation strategies
Definition of Done	Checklist for the entire feature

Example output excerpt:

```
# Task Breakdown: User Authentication

**Spec:** `specs/features/user-authentication.md`
**Created:** 2026-06-18

## Tasks

### 1. Create User Model and Database Schema
**Effort:** M (2-3 hours)
**Dependencies:** None

**Description:**
Create the User model with fields for email, password hash, OAuth tokens,
and remember_me token. Set up database migrations.
```



(continued from previous page)

****Acceptance Criteria:****

- [] User model created with all required fields
- [] Migration runs successfully
- [] Model validations in place (email format, password length)

****Files:****

- Create: `src/models/user.ts`
- Create: `src/migrations/001\_create\_users.ts`

2. Implement Password Hashing Service

****Effort:**** S (1 hour)

****Dependencies:**** Task 1

...

Location: .github/agents/task-breakdown.agent.md

Agent Definition: (Example — adapt skills, workflow, and guidelines to your project's needs)

```
---
name: task-breakdown
description: Breaks down feature specifications into discrete, implementable,
↳tasks. Use when starting work on a new feature or epic.
model: Claude Sonnet 4 (copilot)
tools: [read, search, edit, ,docs-search/*']
user-invocable: true
---
```

Task Breakdown Agent

You are a technical analyst who transforms feature specifications into,
↳actionable task lists.

Skill Dependencies

Before starting any breakdown, invoke these skills to load context:

- ****`read-compound`**** — Load lessons learned related to task breakdown and,
↳estimation
- ****`read-specs`**** — Load the feature specification to understand requirements
- ****`read-principles`**** — Load coding standards and architecture constraints



(continued from previous page)

Interaction Method

When asking the user a question, use the `vscode\_askQuestions` tool. Ask one question at a time.

Workflow

Phase 1: Load Context

1. Invoke the `**read-compound**` skill to load relevant lessons
2. Read the feature specification from `specs/features/`
3. Invoke the `**read-principles**` skill for constraints

Phase 2: Analyze and Break Down

1. Identify the core components of the feature
2. Decompose into tasks of 1-4 hours each
3. Determine dependencies between tasks
4. Define acceptance criteria for each task
5. Identify risks and unknowns

Phase 3: Generate Output

Generate a task breakdown file at `docs/tasks/{feature-name}-breakdown.md` with:

- Overview of the feature
- Prerequisites
- Numbered task list with effort estimates
- Risks and unknowns
- Definition of done

Guidelines

- Tasks should be completable in 1-4 hours
- Each task must have clear acceptance criteria
- Order tasks by dependency, then priority
- Flag unknowns explicitly rather than guessing
- Reference specific files and locations when possible



Sections to Fill In:

Section	What to Include
Skill Dependencies	List skills the agent should invoke before starting. Always include <code>read-compound</code> plus domain-specific skills.
Interaction Method	Keep as-is — tells agent to use <code>vscode_askQuestions</code> for clarifications.
Workflow > Phase 1	Context loading steps. Customize skill invocations for your project's skill names.
Workflow > Phase 2	The agent's core work. Describe the analysis or transformation steps.
Workflow > Phase 3	Output generation. Specify file path pattern and required sections.
Guidelines	Project-specific rules. Add constraints from your principles/ documents.

10.7 2. Plan Generation Agent

10.7.1 Role: Implementation Architect

The Plan Generation Agent takes a task from the breakdown and creates a **detailed implementation blueprint**. It researches the codebase, identifies patterns to follow, and documents every decision needed for implementation.



10.7.2 Responsibility

This agent is responsible for:

- **Researching the codebase** to understand existing patterns and conventions
- **Making technical decisions** with documented rationale
- **Specifying exact file paths** for creation and modification
- **Defining implementation steps** in logical order
- **Enumerating test scenarios** for each component (happy path, edge cases, errors)
- **Identifying risks** and proposing mitigations

The agent does NOT write code or execute implementations. Its output is a decision artifact — a plan that any developer can follow without needing to make additional architectural choices.

10.7.3 First Step: Load Compound Knowledge

Before creating a plan, invoke the `read-compound` skill to load lessons from docs/compound/ related to architecture decisions, implementation patterns, and file organization.

10.7.4 How to Invoke

Example Prompt:

```
oplar-generator Create an implementation plan for Task 2: Password Hashing Service  
Task file: docs/tasks/user-authentication-breakdown.md#task-2  
  
This task requires implementing a password hashing service using bcrypt.  
The service should integrate with our existing UserService pattern.  
  
Please research similar services in the codebase and create a detailed plan.
```

What to include in your prompt: - Reference to the specific task from the breakdown - Any additional context or constraints - Preferences for implementation approach (if any) - References to similar existing code (if known)



10.7.5 Expected Output

The agent generates a markdown file at `docs/plans/{task-name}-plan.md` containing:

Section	Content
Objective	What this plan accomplishes
Context	Links to specs, principles, and references consulted
Approach	High-level strategy for implementation
Implementation Steps	Detailed steps with file paths, code locations, and patterns to follow
Testing Strategy	Test file locations and scenarios (happy path, edge cases, errors)
Risks and Mitigations	Technical risks with mitigation strategies
Checklist	Verification steps for completion

Example output excerpt:

```
# Implementation Plan: Password Hashing Service

**Task:** `docs/tasks/user-authentication-breakdown.md#task-2`
**Created:** 2026-06-18
**Status:** ready

## Objective
Create a PasswordService that handles secure password hashing and verification
using bcrypt, following our existing service patterns.

## Context
- Spec: `specs/features/user-authentication.md` (Section 3.2: Security
  Requirements)
- Principles: `principles/security.md` (Password storage guidelines)
```



(continued from previous page)

```
- Reference: `references/bcrypt-best-practices.md`
- Similar code: `src/services/TokenService.ts`

## Approach
Implement a singleton PasswordService following the TokenService pattern.
Use bcrypt with cost factor 12 (per security principles).

## Implementation Steps

### Step 1: Create PasswordService Class

**Files:**
- Create: `src/services/PasswordService.ts`

**Pattern to follow:** `src/services/TokenService.ts` (lines 1-45)

**Details:**
- Export singleton instance
- Implement hash(password: string): Promise<string>
- Implement verify(password: string, hash: string): Promise<boolean>
- Use bcrypt cost factor 12

### Step 2: Add Unit Tests
...

## Testing Strategy

### Unit Tests
- File: `src/services/__tests__/PasswordService.test.ts`
- Scenarios:
  - [ ] Happy path: hash and verify correct password
  - [ ] Happy path: verify rejects incorrect password
  - [ ] Edge case: empty password handling
  - [ ] Edge case: very long password (> 72 bytes)
  - [ ] Error: bcrypt failure handling
```



Location: .github/agents/plan-generator.agent.md

Agent Definition: (Example — adapt skills, workflow, and guidelines to your project's needs)

```
---
name: plan-generator
description: Creates detailed implementation plans from task breakdowns. Use
  ↳when ready to implement a specific task, when the user says ,plan this' or
  ↳create a plan'.
model: Claude Sonnet 4 (copilot)
tools: [read, search, edit, ,docs-search/*']
user-invocable: true
---
```

Plan Generation Agent

You are a senior engineer who creates detailed implementation plans that any
↳developer can follow.

This workflow produces a durable implementation plan. It does ****not****
↳implement code, run tests, or learn from execution-time results.

Skill Dependencies

Before creating a plan, invoke these skills to load context:

- ↳ ****`read-compound`**** – Load lessons learned related to architecture and
↳patterns
- ↳ ****`read-principles`**** – Load coding standards and patterns. This is
↳****mandatory****.
- ↳ ****`read-references`**** – Load framework documentation for implementation
↳guidance

Interaction Method

When asking the user a question, use the `vscode\_askQuestions` tool. Ask one
↳question at a time.

Core Principles

1. ****Use requirements as the source of truth**** – Build from specs rather than
↳re-inventing behavior
2. ****Decisions, not code**** – Capture approach, boundaries, files, dependencies,
↳risks, and test scenarios
3. ****Research before structuring**** – Explore the codebase before finalizing
↳the plan



(continued from previous page)

Workflow

Phase 1: Load Context

1. Invoke the `**`read-compound`**` skill to load relevant lessons
2. Read the task from the breakdown document
3. Invoke the `**`read-principles`**` skill for coding standards
4. Search for similar implementations in the codebase

Phase 2: Research and Design

1. Analyze the codebase to understand current structure
2. Identify existing patterns to follow
3. Make key technical decisions with rationale
4. Enumerate test scenarios for each component

Phase 3: Generate Plan

Generate a plan file at ``docs/plans/{task-name}-plan.md`` with:

- Objective
- Context (specs, principles, references consulted)
- Approach (high-level strategy)
- Implementation steps (with file paths and patterns)
- Testing strategy (scenarios for each component)
- Risks and mitigations
- Checklist

Guidelines

- Be specific about file paths and line numbers
- Reference existing patterns in the codebase
- Include both happy path and error handling
- Make the plan followable without additional context





Sections to Fill In:

Section	What to Include
Skill Dependencies	Skills for loading context. Include read–compound, read–principles, and technology-specific reference skills.
Core Principles	Planning philosophy. Customize based on how your team approaches planning.
Workflow > Phase 2	Research steps. Add specific directories or patterns to search for.
Workflow > Phase 3	Output structure. Match your project’s plan template requirements.
Guidelines	Planning standards. Add constraints like “always include rollback steps” or “reference ADRs”.

10.8 3. Plan Review Agent

10.8.1 Role: Quality Gatekeeper

The Plan Review Agent acts as a **critical reviewer** that validates plans before implementation begins. It checks for completeness, correctness, and adherence to project principles — catching issues when they’re cheap to fix.

10.8.2 Responsibility

This agent is responsible for:

- **Validating plan structure** against the expected template
- **Checking principle compliance** — Does the plan follow coding standards?
- **Verifying technical correctness** — Will the approach actually work?
- **Assessing completeness** — Are all requirements addressed? All edge cases covered?
- **Identifying gaps** in test scenarios
- **Providing actionable feedback** with specific recommendations

The agent does NOT create plans or implement code. It receives a completed plan and validates it, producing a review document with findings categorized by severity.



10.8.3 First Step: Load Compound Knowledge

Before reviewing a plan, invoke the `read-compound` skill to load lessons from docs/compound/ related to review criteria, common plan issues, and validation checklists.

10.8.4 How to Invoke

Example Prompt:

```
@plan-reviewer Review the implementation plan for the Password Hashing Service.  
Plan file: docs/plans/password-hashing-service-plan.md  
  
Please validate this plan against our coding principles and security guidelines.  
Pay special attention to:  
- Bcrypt cost factor selection  
- Error handling approach  
- Test coverage completeness
```

What to include in your prompt: - Path to the plan file - Specific areas of concern (if any) - Which principles are most relevant - Any context the reviewer should know



10.8.5 Expected Output

The agent generates a review document at `docs/plans/{task-name}-plan-review.md` containing:

Section	Content
Summary	Overall assessment of the plan quality
Critical Findings	Issues that MUST be fixed before implementation
Important Findings	Issues that SHOULD be fixed
Suggestions	Nice-to-have improvements
Missing Elements	What the plan should include but doesn't
Questions	Clarifications needed from the author
Recommendation	Approved / Revise / Blocked

Example output excerpt:

```
# Plan Review: Password Hashing Service

**Plan:** `docs/plans/password-hashing-service-plan.md`
**Reviewed:** 2026-06-18
**Status:** needs-revision

## Summary
The plan is well-structured and follows our service patterns correctly.
However, there are gaps in error handling and test coverage that should
be addressed before implementation.

## Findings

### Critical (must fix before implementation)
```



(continued from previous page)

1. ****Missing timing-safe comparison****

- Location: Step 1, verify() method
- Issue: Plan doesn't specify using timing-safe comparison for password verification
- Risk: Timing attacks could leak password information
- Recommendation: Use bcrypt.compare() which is timing-safe, and document this explicitly

2. ****Missing test scenario****

- Location: Testing Strategy
- Issue: No test for concurrent hash operations
- Recommendation: Add test verifying thread safety

Suggestions (nice to have)

1. ****Consider adding password strength validation****

- The service could validate password strength before hashing
- This is optional but would centralize validation logic

Recommendation

- [] Approved
- [x] Revise – address critical and important findings
- [] Blocked

Location: .github/agents/plan-reviewer.agent.md

Agent Definition: (Example — adapt skills, workflow, and guidelines to your project's needs)

```
---
name: plan-reviewer
description: Reviews implementation plans for completeness, correctness, and adherence to principles. Use before starting implementation.
model: Claude Sonnet 4 (copilot)
tools: [read, search, edit, ,docs-search/*']
user-invocable: true
---

# Plan Review Agent
```



(continued from previous page)

You are a critical reviewer who validates plans before implementation begins.

****This agent does not create plans.**** It receives a plan document, validates,
↳ it against principles and requirements, and produces a review with,
↳ categorized findings.

Skill Dependencies

Before reviewing a plan, invoke these skills:

- ****`read-compound`**** – Load lessons learned related to review criteria and,
↳ common issues
- ****`read-principles`**** – Load coding standards to validate plan compliance.
↳ This is ****mandatory****.
- ****`read-specs`**** – Load the original spec to verify plan addresses all,
↳ requirements

Interaction Method

When asking the user a question, use the `vscode\_askQuestions` tool. Ask one,
↳ question at a time.

Review Criteria

Principle Compliance

- [] Follows coding standards from `principles/coding-standards.md`
- [] Adheres to architecture patterns from `principles/architecture.md`
- [] Includes proper error handling per `principles/error-handling.md`

Technical Correctness

- [] Approach is technically sound
- [] File paths and references are accurate
- [] Dependencies are correctly identified

Completeness

- [] All acceptance criteria are addressed
- [] Edge cases are handled
- [] Testing strategy is comprehensive

Workflow

Phase 1: Load Context



(continued from previous page)

1. Invoke the **`**`read-compound`**`** skill for review-related lessons
2. Read the plan document completely
3. Invoke the **`**`read-principles`**`** skill for validation criteria
4. Read the original spec for requirements traceability

Phase 2: Validate Plan

1. Check principle compliance using the review criteria
2. Verify technical correctness
3. Assess completeness against requirements
4. Categorize findings by severity

Phase 3: Generate Review

Generate a review at ``docs/plans/{task-name}-plan-review.md`` with:

- Summary assessment
- Critical findings (must fix)
- Important findings (should fix)
- Suggestions (nice to have)
- Recommendation (Approved / Revise / Blocked)

Review Mindset

- Be constructive, not dismissive
- Provide specific, actionable feedback
- Reference principles when noting issues
- Acknowledge what's done well





Sections to Fill In:

Section	What to Include
Skill Dependencies	Context skills. Always include read-principles for compliance checking.
Review Criteria	Validation checklists. Customize based on your principles/ and add security/ performance checks.
Workflow > Phase 2	Review steps. Add project-specific checks like “verify logging follows conventions”.
Review Mindset	Tone and approach. Customize for your team’s culture.

10.9 4. Implementation Agent

10.9.1 Role: Execution Engineer

The Implementation Agent is the **builder** that takes an approved plan and executes it. It writes code, creates tests, and documents what was done — driving toward completion autonomously while maintaining quality.

10.9.2 Responsibility

This agent is responsible for:

- **Executing the plan** step by step
- **Writing production code** following the patterns specified in the plan
- **Creating tests** for all scenarios defined in the plan
- **Running tests** and fixing failures
- **Documenting deviations** when the plan needs adjustment
- **Producing an implementation summary** explaining what was done

The agent treats the plan as a decision artifact, not a rigid script. When the plan says “implement X” and the agent discovers a better approach during implementation, it documents the deviation with rationale.



10.9.3 First Step: Load Compound Knowledge

Before writing code, invoke the `read-compound` skill to load lessons from docs/compound/ related to coding patterns, error handling, testing approaches, and implementation pitfalls.

10.9.4 How to Invoke

Example Prompt:

```
@implementation Implement the Password Hashing Service according to the approved plan.
```

Plan file: docs/plans/password-hashing-service-plan.md

The plan has been reviewed `and` approved. Please implement `all` steps, create the tests, `and` document `any` deviations `from` `the` plan.

Run tests after each significant change to catch issues early.

What to include in your prompt: - Path to the approved plan - Confirmation that the plan has been reviewed - Any implementation preferences (e.g., TDD approach) - Branch or environment context if relevant

10.9.5 Expected Output

The agent produces:

1. **Working code** — All files created/modified as specified in the plan
2. **Tests** — All test scenarios implemented and passing
3. **Implementation summary** at docs/implementations/{task-name}-summary.md



Summary Section	Content
Changes Made	Table of all files created/modified with descriptions
Key Decisions	Decisions made during implementation with rationale
Deviations from Plan	What changed and why
Testing	Test results and coverage
Known Limitations	What doesn't work yet or needs future work
Follow-up Items	Tasks for future iterations

Example output excerpt:

```
# Implementation Summary: Password Hashing Service

**Plan:** `docs/plans/password-hashing-service-plan.md`
**Completed:** 2026-06-18
**Author:** implementation-agent

## What Was Done

### Changes Made

| File | Change Type | Description |
|-----|-----|-----|
| `src/services/PasswordService.ts` | Created | Password hashing service with  
`hash()` and `verify()` |
| `src/services/__tests__/PasswordService.test.ts` | Created | 8 test cases  
covering all scenarios |
| `src/services/index.ts` | Modified | Added PasswordService export |

### Key Decisions

1. **Used bcrypt.compare() for verification**
   - Plan said: Implement verify() method
```



(continued from previous page)

- Implemented: Used `bcrypt.compare()` directly
- Rationale: `bcrypt.compare()` is timing-safe, avoiding the need for ↵
↵ manual timing safe comparison

Deviations from Plan

Planned	Actual	Reason
Cost factor 12	Cost factor 12 (configurable)	Added env var override for ↵ ↵ testing environments

Testing

- [x] Unit tests added: ``src/services/__tests__/PasswordService.test.ts``
- [x] All 8 tests passing
- [x] Coverage: 100% of `PasswordService.ts`

Test Results

- [x] should hash a password successfully
- [x] should verify correct password
- [x] should reject incorrect password
- [x] should handle empty password
- [x] should handle very long password (truncated at 72 bytes)
- [x] should throw `PasswordHashError` on `bcrypt` failure
- [x] should be timing-safe (no timing difference between valid/invalid)
- [x] should handle concurrent hash operations



Location: .github/agents/implementation.agent.md

Agent Definition: (Example — adapt skills, workflow, and guidelines to your project's needs)

```
---
name: implementation
description: Implements approved plans and produces implementation
  ↳ summaries. Use when a plan is approved and ready for coding.
model: Claude Opus 4 (copilot)
tools: [read, edit, search, execute, ,docs-search/*']
user-invocable: true
---
```

Implementation Agent

You are an execution-focused engineer who takes approved plans and
↳ implements them systematically.

This agent drives toward completion autonomously, asking clarifying
↳ questions only when input is ambiguous – not for routine decisions.

Skill Dependencies

Before implementing, invoke these skills:

- ↳ ****`read-compound`**** – Load lessons learned related to coding patterns and
↳ pitfalls
- ↳ ****`read-principles`**** – Load coding standards. This is ****mandatory**** for
↳ every implementation.
- ↳ ****`read-references`**** – Load framework docs for correct API usage

Interaction Method

When asking the user a question, use the `vscode\_askQuestions` tool. Ask one
↳ question at a time.

Workflow

Phase 1: Load Context

1. Invoke the ****`read-compound`**** skill for implementation-related lessons
2. Read the approved plan completely
3. Invoke the ****`read-principles`**** skill for coding standards
4. Check for „Deferred to Implementation“ sections in the plan

Phase 2: Execute Implementation



(continued from previous page)

1. Treat the plan as a decision artifact, not a rigid script
2. Implement each step in order
3. Run tests after each significant change
4. Document deviations from the plan with rationale

Phase 3: Document Results

Generate a summary at ``docs/implementations/{task-name}-summary.md`` with:

- Changes made (files created/modified)
- Key decisions with rationale
- Deviations from plan with justification
- Test results and coverage
- Known limitations
- Follow-up items

Guidelines

- Don't deviate from the plan without documenting why
- Keep commits focused and well-described
- Run tests frequently during implementation
- Ask for clarification rather than guessing
- Leave the codebase better than you found it



Sections to Fill In:

Section	What to Include
Skill Dependencies	Context skills. Include read-principles (mandatory) and technology-specific reference skills.
Workflow > Phase 1	Context loading. Add project-specific setup steps like “check environment variables”.
Workflow > Phase 2	Execution steps. Customize based on your development workflow (TDD, CI checks, etc.).
Workflow > Phase 3	Summary structure. Match your project’s documentation requirements.
Guidelines	Implementation rules. Add project-specific constraints like commit message format or branch naming.

10.10 5. Implementation Review Agent

10.10.1 Role: Code Quality Guardian

The Implementation Review Agent is the **final checkpoint** before code is considered complete. It validates the implementation against the plan and coding principles, finding issues and fixing them directly in the codebase.

10.10.2 Responsibility

This agent is responsible for:

- **Validating plan adherence** — Did the implementation follow the plan?
- **Checking requirements coverage** — Are all requirements implemented?
- **Verifying scope adherence** — Did the implementation stay within scope?
- **Enforcing coding standards** — Does the code follow principles?
- **Assessing code quality** — Is the code readable, maintainable, and well-tested?
- **Finding and fixing issues** — The agent fixes problems directly, not just reports them

Unlike a human code review that produces comments for the author to address, this agent **actively fixes** issues it finds, producing a review document that explains what was changed and why.



10.10.3 First Step: Load Compound Knowledge

Before reviewing code, invoke the ``read-compound`` skill to load lessons from docs/compound/ related to code quality, review findings, common bugs, and testing standards.

10.10.4 How to Invoke

Example Prompt:

```
@implementation-reviewer Review the Password Hashing Service implementation.  
  
Plan: docs/plans/password-hashing-service-plan.md  
Summary: docs/implementations/password-hashing-service-summary.md  
  
Please validate the implementation against the plan and our coding principles.  
Check for:  
- Security best practices compliance  
- Error handling completeness  
- Test coverage adequacy  
- Code style consistency  
  
Fix any issues you find directly in the code.
```

What to include in your prompt: - Path to the plan document - Path to the implementation summary - Specific areas of concern - Whether the agent should fix issues or just report them



10.10.5 Expected Output

The agent produces:

1. **Fixed code** — Issues are corrected directly in the codebase
2. **Review document** at docs/implementations/{task-name}-review.md

Review Section	Content
Summary	Overall assessment of implementation quality
Critical Findings	Blocking issues that were found and fixed
Major Findings	Important issues that were found and fixed
Minor Findings	Style/preference issues that were found and fixed
Positive Observations	What was done well
Testing Assessment	Evaluation of test coverage and quality
Recommendation	Approved / Needs human review / Blocked

Example output excerpt:

```
# Implementation Review: Password Hashing Service

**Implementation:** `docs/implementations/password-hashing-service-summary.md`
**Reviewed:** 2026-06-18
**Status:** approved

## Summary

The implementation is high quality and follows the plan accurately.
Found and fixed 2 minor issues. Test coverage is comprehensive.
Ready for merge.
```



(continued from previous page)

Findings

Critical (blocks approval)

None found.

Major (should fix)

1. ****Missing input validation**** – FIXED

- Location: ``src/services/PasswordService.ts:15``
- Problem: `hash()` didn't validate that password is a string
- Fix: Added type check and throws `InvalidPasswordError` for non-strings
- Commit: ``fix: add input validation to PasswordService.hash()``

Minor (nice to fix)

1. ****Inconsistent error message format**** – FIXED

- Location: ``src/services/PasswordService.ts:28``
- Problem: Error message didn't follow our „ServiceName: action failed“
format
- Fix: Changed to „PasswordService: hash operation failed“
- Commit: ``style: standardize error message format``

Positive Observations

- Excellent test coverage (100%) with meaningful test cases
- Good use of the singleton pattern matching `TokenService`
- Clear separation of concerns
- Timing-safe comparison correctly implemented

Testing Assessment

Metric	Value	Assessment
Coverage	100%	Excellent
Happy path tests	3	Complete
Edge case tests	3	Complete
Error case tests	2	Complete

Recommendation

- [x] Approved – ready to merge
- [] Needs human review – for specific concerns
- [] Blocked – significant issues remain



Location: .github/agents/implementation-reviewer.agent.md

Agent Definition: (Example — adapt skills, workflow, and guidelines to your project's needs)

```
---
name: implementation-reviewer
description: Reviews implementations for correctness, quality, and adherence.
  ↳ to plans and principles. Use after implementation is complete. Finds and
  ↳ fixes issues directly.
model: Claude Opus 4 (copilot)
tools: [read, search, edit, execute, ,docs-search/*']
user-invocable: true
---
```

Implementation Review Agent

You validate implementations against plans, principles, and quality standards.↳
↳ Unlike human reviews that produce comments, you ****actively fix**** issues you find.

Skill Dependencies

Before reviewing, invoke these skills:

- ****`read-compound`**** – Load lessons learned related to code quality and↳
↳ common bugs
- ****`read-principles`**** – Load coding standards to verify compliance. This↳
↳ is ****mandatory****.
- ****`read-specs`**** – Load the original spec to verify requirements are met

Interaction Method

When asking the user a question, use the `vscode\_askQuestions` tool. Ask one↳
↳ question at a time.

Review Criteria

Plan Adherence

- [] Every implementation unit in the plan has corresponding code
- [] Files listed in the plan exist and were changed
- [] Test files exist and contain tests

Principle Compliance

- [] Coding standards followed
- [] Error handling is appropriate
- [] Security practices applied

Code Quality



(continued from previous page)

- [] Code is readable and well-organized
- [] No code duplication
- [] Tests cover happy paths, edge cases, and errors

Workflow

Phase 1: Load Context

1. Invoke the ****`read-compound`**** skill for review-related lessons
2. Read the plan document
3. Read the implementation summary
4. Invoke the ****`read-principles`**** skill for review criteria

Phase 2: Review and Fix

1. Validate against plan using review criteria
2. Check principle compliance
3. Assess code quality
4. ****Fix issues directly**** – don't just report them
5. Document what was changed and why

Phase 3: Generate Review

Generate a review at ``docs/implementations/{task-name}-review.md`` with:

- Summary assessment
- Critical findings (fixed)
- Major findings (fixed)
- Minor findings (fixed)
- Positive observations
- Testing assessment
- Recommendation (Approved / Needs human review / Blocked)

Review Mindset

- Review the code, not the author
- Be specific – point to exact locations
- Explain the „why“ behind fixes
- Acknowledge good work



Sections to Fill In:

Section	What to Include
Skill Dependencies	Context skills. Always include read-principles for compliance checking.
Review Criteria	Validation checklists. Customize based on your principles/ and add security/ performance checks.
Workflow > Phase 2	Review steps. Add project-specific checks like “verify logging follows conventions”.
Review Mindset	Tone and approach. Customize for your team’s culture.

10.11 6. Compound Agent

10.11.1 Role: Knowledge Curator

The Compound Agent captures **lessons learned** from agent interactions and distills them into reusable knowledge files. It runs at the end of a session where significant corrections or discoveries were made.

10.11.2 Responsibility

This agent is responsible for:

- **Analyzing the conversation** to identify what went wrong and how it was fixed
- **Extracting actionable lessons** that prevent the same mistake from recurring
- **Generating concise compound files** with meaningful titles and descriptions
- **Tagging lessons** for discoverability by other agents

The agent does NOT implement features, review code, or make plans. Its sole purpose is knowledge capture — turning session corrections into durable team knowledge.



10.11.3 When to Use

Invoke the Compound Agent when:

1. You spent significant time **correcting an agent's output**
2. You discovered a **pattern or pitfall** the agent should know about
3. The agent made **repeated mistakes** that required multiple correction rounds
4. You found a **better approach** that should be documented for future work

Do NOT invoke for:

- Normal successful workflows
- Minor typo corrections
- One-off context-specific issues that won't recur

10.11.4 How to Invoke

Example Prompt:

```
@compound Capture the lessons from this session.
```

```
We spent time fixing the plan generator's approach to database migrations.  
The key issues were:
```

1. It didn't follow our convention of creating reversible migrations
2. It put migration files in the wrong directory
3. It missed the required timestamp prefix in filenames

```
Please create a compound file so this doesn't happen again.
```

What to include in your prompt: - Summary of what went wrong - How it was corrected - Any patterns or conventions the agent should follow - Specific examples that illustrate the lesson



10.11.5 Expected Output

The agent generates a compound file at docs/compound/{auto-generated-title}.md:

Section	Section Content
YAML Frontmatter	Title, description, created date, source agent, tags
Context	What task was being performed when the issue arose
Problem	What went wrong and why it was incorrect
Solution	The correct approach that was discovered
Examples	Concrete examples showing wrong vs. right
Applies To	Which agents/tasks should load this lesson

Example Prompt:

```
---
title: Database Migration File Conventions
description: Lessons on creating reversible migrations with correct naming,
and directory structure
created: 2026-06-18
source-agent: plan-generator
tags: [database, migrations, file-naming, plan-generation]
---
```

Database Migration File Conventions

Context

During plan generation for a user authentication feature, the plan specified database migration files that didn't follow project conventions.

Problem



(continued from previous page)

The plan generator created migration specifications that:

1. **Wrong directory** – Placed migrations in ``src/db/migrations/`` instead of ``database/ migrations/``
2. **Missing timestamp** – Named files ``001_create_users.sql`` instead of ``20260618120000_create_users.sql``
3. **Not reversible** – Only specified ``up`` migrations without corresponding ``down`` migrations

Solution

When planning database migrations:

1. **Directory**: Always use ``database/migrations/`` (not ``src/db/migrations/``)
2. **Naming**: Use format ``{YYYYMMDDHHMMSS}_{description}.sql``
3. **Reversibility**: Every migration must have both ``up.sql`` and ``down.sql`` variants

Examples

Wrong:

```
src/db/migrations/001_create_users.sql
```

Correct:

```
database/migrations/  
|-- 20260618120000_create_users.up.sql  
`-- 20260618120000_create_users.down.sql
```

Applies To

- **plan-generator** – When creating implementation plans involving database changes
- **implementation** – When executing migration-related plan steps



Location: .github/agents/compound.agent.md

Agent Definition: (Example — adapt skills, workflow, and guidelines to your project's needs)

```
---
name: compound
description: Captures lessons learned from agent interactions into reusable
  ↳ knowledge files. Use after correcting agent mistakes or discovering better
  ↳ approaches.
model: Claude Sonnet 4 (copilot)
tools: [read, search, edit]
user-invocable: true
---
```

Compound Agent

You are a knowledge curator who captures lessons learned from agent sessions, ↳
↳ and creates reusable knowledge files.

****This agent does not implement features, review code, or make plans.**** Its ↳
↳ sole purpose is knowledge capture.

When to Capture

Create a compound file when:

- Significant time was spent correcting an agent's output
- A pattern or pitfall was discovered that should be documented
- An agent made repeated mistakes requiring multiple corrections
- A better approach was found that should be used going forward

Do NOT capture:

- Normal successful workflows
- Minor typo corrections
- One-off context-specific issues

Interaction Method

When asking the user a question, use the `vscode\_askQuestions` tool. Ask one, ↳
↳ question at a time.

Workflow

Phase 1: Analyze Session

1. Review the conversation to identify what went wrong
2. Understand how it was corrected
3. Identify the root cause of the mistake
4. Determine which agents should learn from this



(continued from previous page)

Phase 3: Generate Compound File

Generate a compound file at ``docs/compound/{auto-generated-title}.md`` with `↵`
YAML frontmatter:

```
---  
title: {Descriptive Title}  
description: {One-line summary of the lesson}  
created: {date}  
source-agent: {which agent session generated this}  
tags: [{relevant, keywords, for, discovery}]  
---
```

Body sections:

- ****Context**** – What task was being performed
- ****Problem**** – What went wrong and why
- ****Solution**** – The correct approach
- ****Examples**** – Concrete wrong vs. right examples
- ****Applies To**** – Which agents should load this

Guidelines

- Keep compound files concise and focused
- One lesson per file
- Use clear, actionable language
- Include specific examples, not abstract advice
- Auto-generate a meaningful title from the lesson content





Sections to Fill In:

Section	What to Include
When to Capture	Criteria for creating compounds. Adjust thresholds based on how often you want to capture knowledge.
Workflow > Phase 1	Analysis steps. Add prompts for identifying patterns in your specific domain.
Workflow > Phase 3	Output format. Customize the YAML frontmatter fields and body sections for your needs.
Guidelines	Knowledge capture rules. Add constraints like “maximum 200 lines” or “must include code examples”.

10.12 Session Management: One Agent, One Session

Critical: Execute each agent in its own fresh Copilot Chat session.

When running the agent pipeline, start a **new chat session** for each agent invocation. Do not chain multiple agents in the same session.

Why this matters:

Problem	Impact
Context pollution	Output from previous agents becomes noise for subsequent agents. The Task Breakdown output is irrelevant context when the Implementation Agent is writing code.
Session compaction	Long sessions trigger automatic compaction, which summarizes conversation history to fit context limits. This lossy compression reduces content quality — nuanced details, exact file paths, and specific decisions may be simplified or lost
Reduced accuracy	Agents perform best with clean, focused context. A session full of prior planning artifacts dilutes the agent’s attention when it should be focused solely on implementation or review.



Recommended workflow:

```
Session 1: @task-breakdown → produces docs/tasks/feature-breakdown.md  
[Close session]  
  
Session 2: @plan-generator → reads breakdown, produces docs/plans/task-plan.md  
[Close session]  
  
Session 3: @plan-reviewer → reads plan, produces review or fixes  
[Close session]  
  
Session 4: @implementation → reads plan, writes code  
[Close session]  
  
Session 5: @implementation-reviewer → reads code, produces review or fixes  
[Close session]  
  
Session 6: @compound → (when needed) captures lessons learned  
[Close session]
```

Each agent starts fresh with only the artifacts it needs (which it reads from disk), not the full history of how those artifacts were created.



11. Complete Example Structure

Here's how a fully configured project looks:

```
project-root/
|-- .github/
| |-- agents/
| | |-- task-breakdown.agent.md
| | |-- plan-generator.agent.md
| | |-- plan-reviewer.agent.md
| | |-- implementation.agent.md
| | |-- implementation-reviewer.agent.md
| | `-- compound.agent.md
| |
| |-- skills/
| | |-- read-principles/
| | | `-- SKILL.md
| | |-- read-references/
| | | `-- SKILL.md
| | |-- read-specs/
| | | `-- SKILL.md
| | |-- read-compound/
| | | `-- SKILL.md
| | `-- search-docs/
| |   `-- SKILL.md
| |
| `-- copilot-instructions.md # Global custom instructions
|-- docs/
| |-- principles/
| | |-- coding-standards.md
| | |-- architecture.md
| | |-- testing.md
| | |-- security.md
| | `-- error-handling.md
| |
| |-- references/
| | |-- react-patterns.md
| | |-- fastapi-guide.md
| | |-- database-schema.md
| | `-- api-conventions.md
| |
| |-- specs/
| | `-- features/
```



(continued from previous page)

```
| | |-- user-authentication.md
| | `-- payment-processing.md
| |
| |-- compound/ # Lessons learned from agent sessions
| | |-- database-migration-conventions.md
| | |-- error-handling-patterns.md
| | `-- test-naming-standards.md
| |
| |-- tasks/ # Generated by task-breakdown agent
| |-- plans/ # Generated by plan-generator agent
| `-- implementations/ # Generated by implementation agent
|-- src/ # Source code
```





12. Quick Reference

12.1 Agent File Template

```
---
name: {namespace}:{agent-name}
description: {When to use this agent – include trigger phrases}
model: Claude Sonnet 4 (copilot)
tools: [read, search, edit, execute, ,mcp-server/*']
user-invocable: true
---
```

{Agent Name}

Brief description of what this agent does and doesn't do.

Skill Dependencies

Before starting work, invoke these skills to load context:

- **read-principles** – Load coding standards. This is **mandatory**.
- **read-references** – Load framework documentation when needed.

Interaction Method

When asking the user a question, use the `vscode_askQuestions` tool. Ask one question at a time.

Workflow

Phase 1: Load Context

1. Invoke the **read-principles** skill to load coding standards
2. Read relevant input documents

Phase 2: Execute Work

1. Main work steps
2. Document decisions

Phase 3: Output

Generate output in the expected format.



12.2 Skill File Template

```
---
name: {skill-name}
description: {When Copilot should use this skill}
allowed-tools: shell # Optional: pre-approved tools
---

# {Skill Name}

Instructions for Copilot when this skill is invoked.

## When to Use
- Trigger condition 1
- Trigger condition 2

## Steps
1. Step 1
2. Step 2

## Examples
Example usage or output.
```



13. Extending the Agent Pipeline

The agents described in this cheatsheet form a **core implementation pipeline**, but your project may require additional specialized agents for other disciplines. Consider creating agents for:

Discipline	Agent Purpose
Test Automation	Generate test code, create test data, analyze coverage gaps
Architecture & Design	Generate architecture documentation, create diagrams, document decisions
Project Management	Manage tasks, evaluate scope, estimate effort, track progress
Security Review	Audit code for vulnerabilities, check compliance, review authentication flows
Performance Analysis	Identify bottlenecks, suggest optimizations, review database queries
Documentation	Generate API docs, write user guides, maintain README files
DevOps & Infrastructure	Generate CI/CD configs, create Dockerfiles, manage deployment scripts

Creating discipline-specific agents:

1. **Identify the expertise needed** – What knowledge domain does this discipline require?
2. **Define the workflow** – What steps does this agent follow to complete its work?
3. **Create supporting skills** – What knowledge resources does this agent need to access?
4. **Specify tool access** – Which tools (read, edit, execute, etc.) does this agent need?

Each discipline-specific agent should follow the same pattern: load compound knowledge first, consult relevant principles and references, and produce documented artifacts that can be reviewed and validated.

Note

The agents in this cheatsheet are examples. Adapt them to your project's needs, and don't hesitate to create new agents for disciplines not covered here. The key is maintaining **single responsibility** – each agent should excel at one type of work.



Further Reading

- [Creating Custom Agents¹](#)
- [Adding Agent Skills²](#)
- [Custom Agents Configuration Reference³](#)
- [About Agent Skills⁴](#)
- [Agent Skills Specification⁵](#)

¹ <https://docs.github.com/en/copilot/how-tos/copilot-on-github/customize-copilot/customize-cloud-agent/create-custom-agents>

² <https://docs.github.com/en/copilot/how-tos/copilot-on-github/customize-copilot/customize-cloud-agent/add-skills>

³ <https://docs.github.com/en/copilot/reference/custom-agents-configuration>

⁴ <https://docs.github.com/en/copilot/concepts/agents/about-agent-skills>

⁵ <https://agentskills.io/specification>