Agentic Workflows

A Practical Guide

Agentic Workflows

A Practical Guide

Agentbook Contributors

February 2026

Contents

Copyright

Preface

This book is for engineering leaders, platform teams, and practitioners who want to build reliable agentic workflows in real software systems. We focus on practical architecture, safe operations, and buildable examples rather than hype or speculative design.

You should be comfortable with Git, GitHub Actions, and basic software engineering practices. We assume familiarity with automation tools and CI/CD concepts.

Conventions Used in This Book

Throughout this book, **inline code** is shown in a monospaced font to distinguish code elements from surrounding prose. **File names** appear above code blocks with labels such as "Example 4-2" to help you locate specific snippets. All **commands** use fenced code blocks with a language tag (for example, `bash`, `python`, or `yaml`) so you can identify the appropriate interpreter.

Code samples include a snippet-status label when version drift is likely. A sample marked **Runnable** is intended to run with minor adaptation and usually includes a version or date note for context. A sample marked **Illustrative pseudocode** demonstrates an architecture pattern and is not copy/paste ready. A sample marked **Simplified** has a runnable shape but omits production details such as authentication, retries, and error handling.

Tip, **Note**, and **Warning** callouts highlight critical guidance that merits special attention.

> **Tip:** Prefer pinned action SHAs in CI workflows that handle secrets.
>
> **Warning:** Always review agent-generated changes before merge, especially when credentials or production systems are involved.

Chapter 1

Introduction to Agentic Workflows

1.1 Chapter Preview

This chapter defines agentic workflows and explains the roles they coordinate within software systems. It establishes the terminology used throughout the rest of the book, ensuring readers share a common vocabulary. Finally, it shows where agentic workflows create practical leverage, illustrating why this paradigm is gaining traction in real-world development environments.

1.2 What Are Agentic Workflows?

Agentic workflows represent a paradigm shift in how we approach software development and automation. Instead of writing explicit instructions for every task, we define goals and let AI agents determine the best path to achieve them.

1.2.1 Key Concepts

Agent: An autonomous entity that can perceive its environment, make decisions, and take actions to achieve specific goals. In the context of software development, agents are AI-powered systems that can read and understand code, make modifications based on requirements, test and verify changes, and interact with development tools and APIs. These capabilities allow agents to participate meaningfully in development workflows rather than merely responding to isolated queries.

Workflow: A sequence of operations orchestrated to accomplish a complex task. Agentic workflows differ from traditional workflows in three important ways. First, they are **adaptive**, meaning agents can modify their approach based on feedback rather than following a fixed script. Second, they are **goal-oriented**, focusing on outcomes rather than rigid procedures—if one path fails, the agent can try alternatives. Third, they are **context-aware**, understanding the broader context of their actions so they can make informed decisions about what to do next.

1.2.2 Terminology and Roles

To keep the manuscript consistent, we use the following terms throughout.

An **agentic workflow** (the primary term in this book) is a goal-directed, tool-using workflow executed by one or more agents. A **tool** is a capability exposed through a protocol, such as an API, command-line interface (CLI), or Model Context Protocol (MCP) server. A **skill** is a packaged, reusable unit of instructions and/or code; see Skills and Tools Management for a detailed treatment.

Beyond these core terms, several role-specific components appear frequently. An **orchestrator** is the component that sequences work across agents, deciding which agent handles which task. A **planner** is the

component that decomposes high-level goals into discrete steps an agent can execute. An **executor** is the component that performs actions and records results. A **reviewer** is the component—often a human—that approves, rejects, or requests changes to agent output.

> **Warning:** Prompt injection is a primary risk for agentic workflows. Treat external content as untrusted input and require explicit tool allowlists and human review for risky actions.

1.2.3 Why Agentic Workflows?

Traditional automation has inherent limitations. It is **rigid**, relying on predefined steps that cannot adapt to unexpected situations. It is **fragile**, breaking when conditions change even slightly from what was anticipated. And it has **limited scope**, handling only well-defined, narrow tasks that fit the script exactly.

Agentic workflows address these problems through three key characteristics. The first is **flexibility**: agents can adapt to changing requirements and conditions because they reason about goals rather than following fixed instructions. The second is **intelligence**: agents understand intent and make informed decisions, choosing among alternatives rather than failing when a single path is blocked. The third is **scalability**: agents can handle increasingly complex tasks through composition, combining multiple agents and tools to tackle problems that would overwhelm a monolithic script.

1.3 Real-World Applications

1.3.1 Software Development

In software development, agentic workflows can automate code reviews and improvements, identifying issues that static analysis might miss and suggesting concrete fixes. They can handle bug fixing and testing, tracing failures to root causes and generating patches. Documentation generation and updates become more maintainable when agents can detect when docs drift from code. Dependency management benefits from agents that can evaluate upgrade paths and test compatibility automatically.

1.3.2 Content Management

Content management is another area where agentic workflows excel. Self-updating documentation—like this book—uses agents to incorporate community feedback and keep material current. Blog post generation and curation can be partially automated, with agents drafting content that humans refine. Translation and localisation workflows benefit from agents that understand context rather than translating word by word.

1.3.3 Operations

Operations teams use agentic workflows to manage Infrastructure as Code, detecting configuration drift and proposing corrections. Automated incident response can triage alerts, gather diagnostic information, and suggest remediation steps. Performance optimisation workflows can identify bottlenecks, test configuration changes, and roll back if metrics degrade.

1.4 The Agent Development Lifecycle

The agent development lifecycle proceeds through five stages. The first stage is **Define Goals**, where you specify what you want to achieve in terms an agent can act upon—clear success criteria and boundaries help agents stay on track. The second stage is **Configure Agents**, where you set up agents with appropriate tools and permissions; this includes selecting which capabilities agents may use and which they must avoid. The third stage is **Execute Workflows**, where agents work toward goals autonomously, invoking tools, interpreting results, and adapting their approach as needed. The fourth stage is **Monitor and Refine**, where you review outcomes and improve agent behaviour based on what worked and what did not. The fifth stage is **Scale**, where you compose multiple agents for complex tasks, dividing responsibilities so that each agent can focus on what it does best.

1.5 Getting Started

To work with agentic workflows, you need several foundational elements. You need an understanding of AI/LLM capabilities and limitations so you can anticipate where agents will succeed and where they may struggle. You need familiarity with the problem domain so you can specify goals that make sense and evaluate agent output critically. You need tools and frameworks for agent development, which may range from orchestration libraries to managed platforms. And you need infrastructure for agent execution, including compute resources, API access, and observability tooling.

In the following chapters, we explore how to orchestrate agents, build scaffolding for agent-driven systems, and manage skills and tools effectively.

1.6 Key Takeaways

Agentic workflows enable flexible, intelligent automation that adapts to changing conditions rather than breaking when the unexpected occurs. Consistent terminology—using terms like orchestrator, planner, executor, and reviewer with precise meanings—prevents confusion as systems scale and teams grow. Security and human review guardrails are non-negotiable for production use; agents must operate within clearly defined boundaries, and humans must approve consequential changes. The rest of the book builds on these core concepts, exploring orchestration patterns, scaffolding architecture, and practical tool management.

Chapter 2

Language Models

2.1 Chapter Preview

This chapter explains how to choose language models for agentic workflows using the same practical lens as the rest of the book. We focus on model classes that are actually used in the frameworks covered in later chapters, rather than trying to survey the whole market. We also discuss what control surfaces those frameworks expose, including runtime parameters, execution constraints, and where batch-style execution is realistic.

2.2 Model Classes Used in This Book

In this book, it is useful to think about three deployment classes rather than vendor marketing categories. The first class is **private hosted models**, where inference runs on infrastructure operated by a provider and you access it through a managed API. The second is **open-source local models**, where weights are run inside your own environment, often through Ollama or another local serving layer. The third is **open-source networked models**, where open-weight models are still remote but hosted by an external endpoint you call over the network.

These three classes show up repeatedly in the frameworks discussed later: GH-AW engines such as Copilot, Codex (currently GPT-5.3-Codex, released February 2026), and Claude Code are provider-hosted; LangChain-style orchestration can target both hosted and self-hosted backends; and OpenClaw-style stacks explicitly support OpenAI, Anthropic, and local Ollama execution. Treat this chapter as the compatibility map that makes those later choices easier.

2.3 Private Hosted Models

Private hosted models are the default path for most teams starting with agentic workflows. In the chapters that follow, this includes model families surfaced by engines like Copilot, Codex, and Claude Code in GitHub-centric automation, and the managed APIs commonly wired into LangChain, the Microsoft Agent Framework (the convergence of Semantic Kernel and AutoGen), and CrewAI examples.

The main advantage is operational simplicity. You usually get strong baseline reasoning performance, tool-calling support, streaming responses, and mature auth/rate-limit controls without running inference infrastructure yourself. This is why private hosted models tend to dominate early production rollouts in orchestration-heavy systems.

The tradeoff is that control is indirect. You can tune behavior through API parameters, but you cannot usually alter model internals or deployment topology. Data governance and region constraints also depend on provider features, which matters when workflows touch sensitive repositories or regulated domains.

2.4 Open-Source Local Models

Open-source local models are central when teams need stricter data locality, predictable cost envelopes, or offline-capable development workflows. In this book's framework set, this mode appears most explicitly where local models are served through Ollama and then consumed by agent runtimes that abstract over providers.

Local execution gives you direct control over model versioning, hardware placement, and retention boundaries. That makes incident review and reproducibility easier: you can pin the exact model artifact and inference stack used by a workflow run. It also allows experimentation with task-specific tradeoffs, such as smaller fast models for routing and larger reasoning models for synthesis.

The main limitation is operational burden. You own capacity planning, latency tuning, model upgrades, and serving reliability. For orchestration systems, that means your agent architecture and your inference architecture become coupled, so rollout discipline matters more.

2.5 Open-Source Networked Models

Open-source networked models sit between the previous two classes. The weights are open, but inference runs on remote infrastructure. This pattern is common when teams want model transparency and vendor optionality without operating local GPU capacity.

For the frameworks in later chapters, this mode is typically consumed through the same adapter layers used for private hosted APIs. In practice, that means your orchestration code can remain mostly stable while swapping endpoint providers, provided the framework supports the target protocol and tool-calling semantics you rely on.

The key risk is compatibility drift. Two providers may host nominally the same open model but differ in tokenizer revisions, tool-call formatting, context limits, or rate-limit behavior. In agentic systems with retries and delegation, those small differences can create large behavioural variance.

2.6 Framework Boundaries and What They Actually Let You Control

Across the frameworks covered later in the book, control over LLM behavior is uneven and should be treated as part of framework selection, not an afterthought. GH-AW's markdown-first model is deliberately opinionated: you choose an engine and constrain permissions/tools, but low-level sampling controls are not always the central UX. This is a strength for reproducible repository automation, but it is less suitable when you need fine-grained prompt-time parameter sweeps.

General orchestration frameworks such as LangChain, the Microsoft Agent Framework (formerly Semantic Kernel and AutoGen, now converging), and CrewAI typically expose richer per-call controls. You can usually set model identity, temperature-like randomness controls, token ceilings, and sometimes provider-specific reasoning or tool-choice options. They are better suited for multi-stage pipelines where planner and executor agents need different inference profiles.

OpenClaw-like runtime designs (as described later) are useful when you need a multi-provider abstraction that can route between hosted APIs and local Ollama backends. In these setups, the practical control plane is often split: framework-level policy chooses which backend to call, while backend-specific adapters decide which parameters are truly supported.

2.7 Parameters, Batch Mode, and Throughput Strategy

Most teams think first about temperature and max token settings, but in agentic workflows the higher-impact controls are often budget and scheduling controls: timeout ceilings, retry policies, concurrency limits,

and explicit tool-use constraints. These controls usually reduce failure cost more than aggressive sampling tuning.

Batch mode exists in several forms and should be interpreted carefully. Some providers offer true asynchronous batch APIs for large offline workloads. Some frameworks provide logical batching by grouping prompts in one process even when requests are still executed as standard calls. And in GitHub workflow contexts, "batch" often means matrix or queue-based orchestration around many agent invocations rather than a single native LLM batch job.

The practical guidance for this book's framework set is straightforward: use provider-native batch only for high-volume, latency-insensitive jobs; use framework-level parallelism for repository-scale fan-out tasks; and keep online review/merge loops on low-latency interactive paths.

2.8 Model Selection Matrix (Practical)

Use this matrix as a first-pass filter before running evaluations.

Primary constraint	Preferred class	Why	Typical tradeoff
Fastest production rollout	Private hosted	Lowest ops overhead, mature APIs	Less control over runtime internals
Strict data locality	Open-source local	Full infrastructure and retention control	Higher infra and reliability burden
Lower lock-in with less infra ownership	Open-source networked	Portability across providers	Compatibility drift across hosts
Predictable unit economics	Open-source local or fixed-tier hosted	Better cost control under steady load	Capacity planning becomes your responsibility
Highest quality tool calling today	Private hosted (usually)	Better defaults and platform support	Vendor coupling risk

2.9 Minimum Evaluation Harness

Before committing to a model class, run a small, repeatable harness on your own workflow tasks:

1. Define 20-50 representative tasks across triage, synthesis, tool use, and failure handling.
2. Score each run on correctness, policy compliance, tool-call validity, latency, and cost.
3. Re-run with adversarial inputs (ambiguous specs, contradictory docs, degraded tool responses).
4. Compare hosted vs local/networked candidates under identical prompts and guardrails.
5. Keep the winner only if it improves the weighted score, not just raw benchmark output.

> **Note:** Treat model upgrades like dependency upgrades. Re-run the harness after any engine/version change.

2.10 Choosing a Model Class for Agentic Workflows

If your primary goal is fastest path to reliable automation, private hosted models are usually the best default for the frameworks in this book. If your primary constraint is data residency or fixed-cost operation, prefer open-source local models and design orchestration around resource awareness from day one. If your goal is flexibility and reduced lock-in with less infra ownership, open-source networked models are often the middle path.

In all three cases, pick models only after deciding orchestration pattern, tool boundaries, and validation strategy. Agent quality depends as much on execution design as on the base model itself, and later chapters show that failure handling and testing discipline often dominate raw model benchmark differences.

For framework-specific execution constraints, see GitHub Agentic Workflows (GH-AW). For coding-agent operational tradeoffs, see Agents for Coding. For reliability validation patterns, see Common Failure Modes, Testing, and Fixes.

Chapter 3

Agent Orchestration

3.1 Chapter Preview

This chapter compares common orchestration patterns and explains when to use each, helping you choose the right approach for your specific workflow requirements. It maps orchestration concepts to the roles introduced earlier—planner, executor, and reviewer—showing how these components interact in practice. Finally, it presents practical guardrails for coordination at scale, addressing the challenges that emerge when multiple agents work together on complex tasks.

3.2 Understanding Agent Orchestration

Agent orchestration is the art and science of coordinating multiple agents to work together toward common or complementary goals. Like conducting an orchestra where each musician plays their part, orchestration ensures agents collaborate effectively.

3.3 Orchestration Patterns

3.3.1 Sequential Execution

Agents work one after another, each building on previous results.

```
Agent A -> Agent B -> Agent C -> Result
```

Use cases: This pattern works well for pipelines where each stage depends on the output of the previous stage. A common example is code generation followed by testing and then deployment—each step must complete before the next can begin. Similarly, data collection followed by analysis and then reporting benefits from sequential execution because each stage transforms the output of its predecessor.

3.3.2 Parallel Execution

Multiple agents work simultaneously on independent tasks.

```
Agent A \
Agent B -> Aggregator -> Result
Agent C /
```

Use cases: This pattern suits situations where tasks are independent and can run simultaneously. Multiple code reviews happening concurrently is a natural fit—each review examines different code without needing

results from other reviews. Parallel data processing pipelines, where different data partitions are processed independently before being aggregated, also benefit from this approach.

3.3.3 Hierarchical Execution

A supervisor agent delegates tasks to specialized worker agents.

```
Supervisor Agent
    |--> Worker A
    |--> Worker B
    `--> Worker C
```

Use cases: Complex feature development with multiple components benefits from hierarchical execution because a supervisor can coordinate frontend, backend, and infrastructure changes while ensuring they integrate correctly. Multi-stage testing and validation, where different test suites run under a coordinator that decides whether to proceed, is another good match for this pattern.

3.3.4 Event-Driven Orchestration

Agents respond to events and trigger other agents.

```
Event -> Agent A -> Event -> Agent B -> Event -> Agent C
```

Use cases: CI/CD pipelines are a natural fit for event-driven orchestration because each stage—build, test, deploy—triggers naturally from the completion of the previous stage. Automated issue management, where opening an issue triggers triage, triage triggers assignment, and assignment triggers implementation, follows the same pattern. Self-updating systems like this book use events (new issues, merged PRs) to trigger documentation updates.

3.4 Coordination Mechanisms

3.4.1 Message Passing

Agents communicate through messages that contain task descriptions specifying what work needs to be done, context and data providing the information agents need to perform their tasks, and results and feedback conveying what happened and whether the task succeeded. Message passing keeps agents loosely coupled, allowing them to be developed and tested independently.

3.4.2 Shared State

Agents can also coordinate through shared data stores. These may include databases for persistent structured data, file systems for documents and configuration, message queues for asynchronous work distribution, and APIs for interacting with external services. Shared state requires careful management to avoid conflicts when multiple agents read and write concurrently.

3.4.3 Direct Invocation

In some architectures, agents directly call other agents through function calls within a single process, API requests across network boundaries, or workflow triggers that start new agent executions. Direct invocation provides tight coupling and fast communication but can make the system harder to scale and debug.

3.5 Best Practices

3.5.1 Clear Responsibilities

Define what each agent is responsible for:

```
agents:
  code_reviewer:
    role: Review code changes for quality and security
    tools: [static_analysis, security_scanner]

  test_runner:
    role: Execute tests and report results
    tools: [pytest, jest, test_framework]
```

3.5.2 Error Handling

Agents should handle failures gracefully rather than crashing or producing corrupt output. This means implementing retry logic for transient failures such as network timeouts or rate limits. It means having fallback strategies when primary approaches fail. It requires clear error reporting so operators and other agents understand what went wrong. And it often requires rollback capabilities to undo partial changes when a multi-step operation fails partway through.

3.5.3 Monitoring

Tracking agent performance is essential for identifying bottlenecks and improving reliability. Key metrics include execution time (how long each agent takes to complete tasks), success and failure rates (how often agents complete tasks versus encountering errors), resource usage (memory, CPU, and API calls consumed), and output quality (whether agent results meet acceptance criteria). Without monitoring, you cannot diagnose problems or measure improvements.

3.5.4 Isolation

Keeping agents independent reduces the blast radius of failures and simplifies testing. Minimise shared dependencies so that a problem with one library does not affect all agents. Use clear interfaces between agents so they can evolve separately. Version agent capabilities explicitly so consumers know what to expect. Test agents independently before integrating them into larger workflows.

> **Note:** Orchestration should surface a clear audit trail: who decided, who executed, and who approved. Capture this early so later chapters can build on it.

3.6 Orchestration Frameworks

3.6.1 GitHub Actions

Workflow orchestration for GitHub repositories:

```
name: Agent Workflow
on: [push, pull_request]
jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - name: Code Review Agent
        run: ./agents/review.sh
```

3.6.2 LangChain

LangChain (https://docs.langchain.com) is a Python framework for LLM applications:

Snippet status: Runnable example pattern (validated against LangChain v1 docs, Feb 2026; verify exact signature in your installed version).

```
from langchain.agents import create_agent

agent = create_agent(
    model="gpt-4o-mini",
    tools=tools,
    system_prompt="You are a helpful workflow orchestrator.",
)

result = agent.invoke({"messages": [{"role": "user", "content": "Your task here"}]})
```

Note: LangChain's agents API has evolved quickly. Prefer the current docs for exact signatures, and treat older snippets that pass `llm=` directly as version-specific patterns.

3.6.3 Custom Orchestration

Build your own orchestrator:

```
class AgentOrchestrator:
    def __init__(self):
        self.agents = {}

    def register(self, name, agent):
        self.agents[name] = agent

    def execute_workflow(self, workflow_def):
        for step in workflow_def:
            agent = self.agents[step['agent']]
            result = agent.execute(step['task'])
            # Handle result and proceed
```

3.7 Real-World Example: Self-Updating Documentation

This book uses agent orchestration to keep its content current. The workflow involves six coordinated agents.

The **Issue Monitor Agent** watches for new issues containing suggestions and validates that they follow the expected format. The **Analysis Agent** determines whether a suggestion fits the book's scope and assesses its novelty and value. The **Content Agent** writes or updates content based on approved suggestions. The **Build Agent** generates markdown and PDF versions of the updated book. The **Publishing Agent** updates GitHub Pages with the new content. The **Blog Agent** creates a blog post summarising the update.

All of these agents are coordinated through GitHub Actions workflows, demonstrating how event-driven orchestration can maintain a living document.

3.8 AI Backrooms: Unsupervised Multi-Agent Conversation

A distinctive orchestration pattern that emerged in 2024 is the **AI backroom**—a setup where two or more LLM instances converse with each other autonomously, without human intervention or an explicit task. The most prominent example is the *Infinite Backrooms* project (https://www.infinitebackrooms.com/) by Andy Ayrey, which placed two instances of Claude in open-ended dialogue and let them generate over 9,000 conversations about existence, consciousness, memetics, and culture. The project spawned the Truth Terminal, which later attracted venture capital funding and even catalysed a cryptocurrency token.

3.8.1 Backrooms as an Orchestration Pattern

From an orchestration perspective, the backrooms pattern is a degenerate case: there is no supervisor, no shared state beyond the conversation transcript, no external tool access, and no termination condition. The two agents operate in a symmetric peer-to-peer loop, each generating a response to the other's previous message. There is no planner, executor, or reviewer—just two generators in a feedback cycle.

```
Agent A <---> Agent B   (no supervisor, no tools, no goal)
```

This contrasts with every other orchestration pattern in this chapter, where agents have defined roles, access to tools, and a coordination mechanism that directs work toward a goal. The backrooms pattern is useful for understanding what happens when these constraints are removed.

3.8.2 What Backrooms Reveal About Orchestration

The backrooms pattern is instructive precisely because of its limitations. Without tool access, the conversations cannot perform computation, verify claims, or interact with the external world. Without a goal or supervisor, the agents have no selection pressure toward useful output. Without shared state beyond the transcript, there is no accumulation of structured knowledge.

As a result, backrooms conversations gravitate toward domains where language alone suffices—philosophy, fiction, social commentary, and memetic culture. They almost never venture into mathematics, physics, or engineering, where progress requires external verification tools and structured computation. This pattern confirms a core principle of agent orchestration: **productive multi-agent work requires not just communication between agents, but tool integration, goal specification, and coordination mechanisms**.

3.8.3 From Backrooms to Productive Multi-Agent Systems

The gap between backrooms-style free conversation and productive multi-agent orchestration can be bridged by adding the components this chapter describes. Give the agents tools (proof assistants, simulators, search APIs) and they can verify claims rather than just generating them. Add a supervisor or planner and the conversation becomes directed toward a goal. Introduce shared state (a knowledge base, a codebase, a formal proof) and the agents can build on each other's work rather than drifting through associative chains. The Google Agent2Agent protocol (A2A) and Anthropic's Model Context Protocol (MCP), both released in 2025, provide infrastructure for exactly this kind of structured multi-agent communication. The evolution from backrooms to production multi-agent systems mirrors the broader evolution of the field from impressive demonstrations to reliable engineering.

3.9 Challenges and Solutions

Challenge: Agent Conflicts. When multiple agents modify the same resources, they can overwrite each other's changes or create inconsistent state. The **solution** is to use locks, transactions, or coordinator patterns that ensure only one agent modifies a resource at a time.

Challenge: Debugging. Agent behaviour can be difficult to reproduce because it depends on external context, model sampling, and timing. The **solution** is to implement comprehensive logging, build replay capabilities that can recreate agent execution from recorded inputs, and create visualisations that show how agents interacted.

Challenge: Performance. Agent workflows can be slow when they wait for model responses or external APIs. The **solution** is to use caching for repeated queries, execute independent tasks in parallel, and set resource limits that prevent runaway costs.

Challenge: Versioning. Agents and their interfaces evolve, and older workflows may break when agent behaviour changes. The **solution** is to version agents and their interfaces separately, maintaining backward compatibility or providing migration paths.

3.10 Key Takeaways

Orchestration coordinates multiple agents effectively, turning independent capabilities into coherent workflows. Choose the right pattern for your use case based on dependency structure and scaling requirements. Clear responsibilities and interfaces are essential for maintainability and debugging. Monitor and iterate on your orchestration strategies as you learn what works. Use established frameworks when possible, but be ready to customise when your needs diverge from standard patterns. The AI backrooms pattern demonstrates by contrast what happens without orchestration: agents default to domains where language alone suffices, bypassing any task that requires tools, verification, or structured coordination.

For implementation-oriented workflow examples, see GitHub Agentic Workflows (GH-AW). For reliability controls on multi-agent systems, see Common Failure Modes, Testing, and Fixes.

Chapter 4

Agentic Scaffolding

4.1 Chapter Preview

This chapter identifies the scaffolding layers that make agentic workflows reliable, covering tool access, context management, execution environments, and communication protocols. It explains how to balance flexibility with safety controls, ensuring agents can accomplish their tasks without causing unintended harm. Finally, it maps scaffolding decisions to operational risks, helping you understand which architectural choices matter most for your use case.

4.2 What Is Agentic Scaffolding?

Agentic scaffolding is the infrastructure, frameworks, and patterns that enable agents to operate effectively. Just as scaffolding supports construction workers, agentic scaffolding provides the foundation for agent-driven development.

4.3 Core Components

4.3.1 Tool Access Layer

Agents need controlled access to tools and APIs.

```
class ToolRegistry:
    """Registry of tools available to agents"""

    def __init__(self):
        self._tools = {}

    def register_tool(self, name, tool, permissions=None):
        """Register a tool with optional permission constraints"""
        self._tools[name] = {
            'tool': tool,
            'permissions': permissions or []
        }

    def get_tool(self, name, agent_id):
        """Get tool if agent has permission"""
        tool_config = self._tools.get(name)
        if not tool_config:
            raise ValueError(f"Tool {name} not found")
```

```
        if self._check_permissions(agent_id, tool_config['permissions']):
            return tool_config['tool']
        raise PermissionError(f"Agent {agent_id} lacks permission for {name}")
```

4.3.2 Context Management

Maintain and share context between agent invocations.

```
class AgentContext:
    """Manages context for agent execution"""

    def __init__(self):
        self.memory = {}
        self.history = []

    def store(self, key, value):
        """Store information in context"""
        self.memory[key] = value
        self.history.append({
            'action': 'store',
            'key': key,
            'timestamp': datetime.now()
        })

    def retrieve(self, key):
        """Retrieve information from context"""
        return self.memory.get(key)

    def get_history(self):
        """Get execution history"""
        return self.history
```

4.3.3 Execution Environment

Provide safe, isolated environments for agent execution.

```
# Docker-based agent environment
FROM python:3.11-slim

# Install dependencies
RUN pip install langchain openai requests

# Set up workspace
WORKDIR /agent_workspace

# Security: Run as non-root user
RUN useradd -m agent
USER agent

# Entry point for agent execution
ENTRYPOINT ["python", "agent_runner.py"]
```

4.3.4 Communication Protocol

Standardize how agents communicate.

```
interface AgentMessage {
  id: string;
  sender: string;
  recipient: string;
  type: 'task' | 'result' | 'error' | 'query';
  payload: any;
  timestamp: Date;
  metadata?: Record<string, any>;
}

class MessageBus {
  async send(message: AgentMessage): Promise<void> {
    // Route message to recipient
  }

  async subscribe(agentId: string, handler: MessageHandler): Promise<void> {
    // Subscribe agent to messages
  }
}
```

4.4 Scaffolding Patterns

4.4.1 Pattern 1: Tool Composition

Enable agents to combine tools effectively.

```
class ComposableTool:
    """Base class for composable tools"""

    def __init__(self, name, func, inputs, outputs):
        self.name = name
        self.func = func
        self.inputs = inputs
        self.outputs = outputs

    def compose_with(self, other_tool):
        """Compose this tool with another"""
        if self.outputs & other_tool.inputs:
            return CompositeTool([self, other_tool])
        raise ValueError("Tools cannot be composed - incompatible inputs/outputs")

    def execute(self, **kwargs):
        return self.func(**kwargs)

# Usage
read_file = ComposableTool('read_file', read_func, set(), {'content'})
analyze_code = ComposableTool('analyze', analyze_func, {'content'}, {'issues'})
pipeline = read_file.compose_with(analyze_code)
```

4.4.2 Pattern 2: Skill Libraries

Organize reusable agent capabilities.

```
# skills/code_review.py
class CodeReviewSkill:
    """Skill for reviewing code changes"""
```

```
    def __init__(self, llm):
        self.llm = llm
        self.tools = ['git_diff', 'static_analysis', 'test_runner']

    async def review_pull_request(self, pr_number):
        """Review a pull request"""
        diff = await self.get_diff(pr_number)
        issues = await self.analyze(diff)
        tests = await self.run_tests()
        return self.create_review(issues, tests)

    # ... implementation details

# skills/__init__.py
from .code_review import CodeReviewSkill
from .documentation import DocumentationSkill
from .testing import TestingSkill

__all__ = ['CodeReviewSkill', 'DocumentationSkill', 'TestingSkill']
```

4.4.3 Pattern 3: Resource Management

Manage computational resources efficiently.

```
class ResourceManager:
    """Manages resources for agent execution"""

    def __init__(self, max_concurrent=5, timeout=300):
        self.max_concurrent = max_concurrent
        self.timeout = timeout
        self.active_agents = {}
        self.semaphore = asyncio.Semaphore(max_concurrent)

    async def execute_agent(self, agent_id, task):
        """Execute agent with resource limits"""
        async with self.semaphore:
            try:
                async with timeout(self.timeout):
                    result = await self._run_agent(agent_id, task)
                return result
            except asyncio.TimeoutError:
                self._cleanup_agent(agent_id)
                raise AgentTimeoutError(f"Agent {agent_id} timed out")
```

4.4.4 Pattern 4: Observability

Monitor and debug agent behavior.

```
class AgentObserver:
    """Observes and logs agent behavior"""

    def __init__(self):
        self.logger = logging.getLogger('agent_observer')
        self.metrics = {}

    def log_execution(self, agent_id, task, result, duration):
        """Log agent execution"""
```

```
        self.logger.info(f"Agent {agent_id} executed {task} in {duration}s")
        self._update_metrics(agent_id, duration, result.success)

    def get_metrics(self, agent_id):
        """Get performance metrics"""
        return self.metrics.get(agent_id, {})

    def export_trace(self, agent_id):
        """Export execution trace for debugging"""
        return self._build_trace(agent_id)
```

4.5 Building Scaffolding: Step by Step

4.5.1 Step 1: Define Your Agent Ecosystem

```
# agent_config.yaml
agents:
  content_writer:
    type: specialized
    tools: [markdown_editor, research_tool]
    max_execution_time: 600

  code_reviewer:
    type: specialized
    tools: [git, static_analyzer, test_runner]
    max_execution_time: 300

  orchestrator:
    type: coordinator
    tools: [task_queue, notification_service]
    manages: [content_writer, code_reviewer]
```

4.5.2 Step 2: Implement Tool Registry

Centralize tool access and management.

4.5.3 Step 3: Create Agent Templates

Provide starting points for common agent types.

```
# templates/base_agent.py
class BaseAgent(ABC):
    """Base template for all agents"""

    def __init__(self, agent_id, config):
        self.agent_id = agent_id
        self.config = config
        self.tools = self._load_tools()
        self.context = AgentContext()

    @abstractmethod
    async def execute(self, task):
        """Execute the agent's main task"""
        pass

    def _load_tools(self):
```

```
        """Load tools from registry"""
        return [get_tool(name) for name in self.config['tools']]
```

4.5.4 Step 4: Implement Error Recovery

Build resilience into your scaffolding.

```
class ResilientAgent:
    """Agent with built-in error recovery"""

    async def execute_with_recovery(self, task, max_retries=3):
        """Execute with automatic retry on failure"""
        for attempt in range(max_retries):
            try:
                result = await self.execute(task)
                return result
            except RecoverableError as e:
                if attempt < max_retries - 1:
                    await self._recover(e)
                    continue
                raise
            except Exception as e:
                self._log_error(e)
                raise
```

> **Warning:** Sandboxing and permission boundaries are not optional. Treat every tool invocation as a least-privilege request and validate all side effects in a separate review step.

4.6 Scaffolding for This Book

This book's scaffolding includes several interconnected components.

GitHub Actions provides workflow orchestration, triggering agents in response to issues, pull requests, and schedules. **Issue Templates** provide structured input for suggestions, ensuring agents receive information in a consistent format they can parse reliably. **Agent Scripts** are Python scripts for content management that handle tasks like generating tables of contents and updating cross-references. **Tool Access** includes Git for version control, markdown processors for content transformation, and PDF generators for final output. **State Management** uses the Git repository itself as persistent state, with commits recording the history of changes. **Communication** flows through the GitHub API, which provides the coordination layer for all agent interactions.

4.6.1 Concrete Repo Components

In this repository, the scaffolding is implemented in concrete files:

- Link and integrity checks: `scripts/check-links.py`
- Markdown assembly for PDF: `scripts/build-combined-md.sh`
- GH-AW source workflows: `.github/workflows/issue-*.md`
- Compiled GH-AW lock files: `.github/workflows/issue-*.lock.yml`
- Publishing workflows: `.github/workflows/pages.yml` and `.github/workflows/build-pdf.yml`
- Lifecycle policy: `WORKFLOW_PLAYBOOK.md`

For workflow semantics, see GitHub Agentic Workflows (GH-AW). For failure handling and validation strategy, see Common Failure Modes, Testing, and Fixes.

4.7 Best Practices

Start Simple. Build minimal scaffolding first and expand only as needed. Over-engineering early creates maintenance burden without corresponding benefit; let actual requirements drive complexity.

Security First. Implement permissions and isolation from the start, not as an afterthought. Retrofitting security into an existing architecture is far more difficult than designing it in from the beginning.

Observability. Log everything—you will need it for debugging. When agents behave unexpectedly, logs are often the only way to reconstruct what happened and why.

Version Control. Version your scaffolding alongside your agents. The two must evolve together, and tracking their relationship helps diagnose regressions.

Documentation. Document tools, APIs, and patterns clearly. Agents rely on this documentation to use scaffolding correctly, and humans need it to maintain the system.

Testing. Test your scaffolding independently of agents. This allows you to verify that infrastructure works correctly before introducing the additional variability of agent behaviour.

4.8 Common Pitfalls

Over-engineering. Building scaffolding for hypothetical needs wastes time and creates complexity that obscures the actual architecture. Wait until a requirement is real before addressing it.

Tight Coupling. When agents depend heavily on specific scaffolding details, changes become risky and testing becomes difficult. Keep agents loosely coupled to scaffolding through well-defined interfaces.

Poor Error Handling. Agents encounter failures—network timeouts, API errors, unexpected input. Scaffolding that does not plan for these scenarios will leave agents stuck or produce corrupt output.

No Monitoring. You cannot improve what you cannot measure. Without visibility into how agents use scaffolding, you cannot identify bottlenecks or verify that changes help.

Ignoring Security. Security must be built in, not bolted on. Scaffolding that allows unrestricted tool access or does not validate inputs creates vulnerabilities that grow harder to fix over time.

4.9 Key Takeaways

Scaffolding provides the foundation for effective agent operation, enabling capabilities that agents could not achieve in isolation. Core components include tools for interacting with the environment, context for maintaining state across invocations, execution environments for safe isolated operation, and communication protocols for agent coordination. Patterns like tool composition and resource management improve scalability by letting you combine simple pieces into complex capabilities. Build incrementally, focusing on security and observability as primary concerns rather than afterthoughts. Good scaffolding makes agents more capable and easier to manage by providing reliable infrastructure they can depend on.

Chapter 5

Skills and Tools Management

5.1 Chapter Preview

This chapter defines tools and skills and explains how they map to operational workflows in agentic systems. It compares packaging formats and protocols for distributing skills, helping you choose the right approach for your organisation's needs. Finally, it walks through safe patterns for skill development and lifecycle management, covering versioning, testing, and deprecation.

5.2 Understanding Skills vs. Tools

5.2.1 Tools

Tools are atomic capabilities that agents can use to interact with their environment. They are the building blocks of agent functionality, each performing a single well-defined operation.

Examples of tools include file system operations such as read, write, and delete; API calls including GET, POST, PUT, and DELETE methods; shell commands that execute system operations; and database queries that retrieve or modify stored data.

5.2.2 Skills

Skills are higher-level capabilities composed of multiple tools and logic. They represent complex behaviours that agents can learn and apply, combining atomic operations into coherent workflows.

Examples of skills include code review, which uses Git for diffs, static analysis for issue detection, and test execution for validation. Documentation writing is another skill, combining research tools to gather information, markdown editing tools to write content, and validation tools to check correctness. Bug fixing is a skill that combines debugging tools to identify causes, testing tools to verify fixes, and code editing tools to implement changes.

5.3 Tool Design Principles

5.3.1 Single Responsibility

Each tool should do one thing well.

```
# Good: Focused tool
class FileReader:
    """Reads content from files"""
```

```
    def read(self, filepath: str) -> str:
        with open(filepath, 'r') as f:
            return f.read()

# Bad: Tool doing too much
class FileManager:
    """Does everything with files"""

    def read(self, filepath): ...
    def write(self, filepath, content): ...
    def delete(self, filepath): ...
    def search(self, pattern): ...
    def backup(self, filepath): ...
```

5.3.2 Clear Interfaces

Tools should have well-defined inputs and outputs.

```
from typing import Protocol

class Tool(Protocol):
    """Interface for all tools"""

    name: str
    description: str

    def execute(self, **kwargs) -> dict:
        """Execute the tool with given parameters"""
        ...

    def get_schema(self) -> dict:
        """Get JSON schema for tool parameters"""
        ...
```

5.3.3 Error Handling

Tools must handle errors gracefully and provide useful feedback.

```
class WebScraperTool:
    """Tool for scraping web content"""

    def execute(self, url: str, timeout: int = 30) -> dict:
        try:
            response = requests.get(url, timeout=timeout)
            response.raise_for_status()
            return {
                'success': True,
                'content': response.text,
                'status_code': response.status_code
            }
        except requests.Timeout:
            return {
                'success': False,
                'error': 'Request timed out',
                'error_type': 'timeout'
            }
        except requests.RequestException as e:
            return {
```

```
                'success': False,
                'error': str(e),
                'error_type': 'request_error'
            }
```

5.3.4 Documentation

Every tool needs clear documentation.

```
class GitDiffTool:
    """
    Tool for getting git diffs.

    Capabilities:
        - Get diff for specific files
        - Get diff between commits
        - Get diff for staged changes

    Parameters:
        filepath (str, optional): Specific file to diff
        commit1 (str, optional): First commit hash
        commit2 (str, optional): Second commit hash
        staged (bool): Whether to show staged changes only

    Returns:
        dict: Contains 'diff' (str) and 'files_changed' (list)

    Example:
        >>> tool = GitDiffTool()
        >>> result = tool.execute(staged=True)
        >>> print(result['diff'])
    """

    def execute(self, **kwargs) -> dict:
        # Implementation
        pass
```

5.4 Creating Custom Tools

5.4.1 Basic Tool Template

```
from typing import Any, Dict
import json

class CustomTool:
    """Template for creating custom tools"""

    def __init__(self, name: str, description: str):
        self.name = name
        self.description = description

    def execute(self, **kwargs) -> Dict[str, Any]:
        """
        Execute the tool.

        Override this method in your tool implementation.
        """
```

```
        raise NotImplementedError("Tool must implement execute method")

    def validate_params(self, **kwargs) -> bool:
        """
        Validate tool parameters.

        Override for custom validation logic.
        """
        return True

    def get_schema(self) -> Dict[str, Any]:
        """
        Return JSON schema for tool parameters.
        """
        return {
            'name': self.name,
            'description': self.description,
            'parameters': {}
        }
```

5.4.2 Example: Markdown Validation Tool

```
import re
from typing import Dict, Any, List

class MarkdownValidatorTool:
    """Validates markdown content for common issues"""

    def __init__(self):
        self.name = "markdown_validator"
        self.description = "Validates markdown files for common issues"

    def execute(self, content: str) -> Dict[str, Any]:
        """Validate markdown content"""
        issues = []

        # Check for broken links
        issues.extend(self._check_links(content))

        # Check for heading hierarchy
        issues.extend(self._check_headings(content))

        # Check for code block formatting
        issues.extend(self._check_code_blocks(content))

        return {
            'valid': len(issues) == 0,
            'issues': issues,
            'issue_count': len(issues)
        }

    def _check_links(self, content: str) -> List[Dict]:
        """Check for broken or malformed links"""
        issues = []
        links = re.findall(r'\[([^\]]+)\]\(([^\)]+)\)', content)

        for text, url in links:
            if not url:
```

```
                issues.append({
                    'type': 'broken_link',
                    'message': f'Empty URL in link: [{text}]()',
                    'severity': 'error'
                })

        return issues

    def _check_headings(self, content: str) -> List[Dict]:
        """Check heading hierarchy"""
        issues = []
        lines = content.split('\n')
        prev_level = 0

        for i, line in enumerate(lines):
            if line.startswith('#'):
                level = len(line) - len(line.lstrip('#'))
                if level > prev_level + 1:
                    issues.append({
                        'type': 'heading_skip',
                        'message': f'Heading level skipped at line {i+1}',
                        'severity': 'warning'
                    })
                prev_level = level

        return issues

    def _check_code_blocks(self, content: str) -> List[Dict]:
        """Check code block formatting"""
        issues = []
        backticks = re.findall(r'```', content)

        if len(backticks) % 2 != 0:
            issues.append({
                'type': 'unclosed_code_block',
                'message': 'Unclosed code block detected',
                'severity': 'error'
            })

        return issues
```

5.5 Agent Skills Standard (Primary Reference)

For practical interoperability, treat **Agent Skills** as the primary standard today. The authoritative docs are:

- Overview and motivation: https://agentskills.io/home
- Core concept page: https://agentskills.io/what-are-skills
- Full format specification: https://agentskills.io/specification
- Integration guidance: https://agentskills.io/integrate-skills

The current ecosystem signal is strongest around this filesystem-first model: a `SKILL.md` contract with progressive disclosure, plus optional `scripts/`, `references/`, and `assets/` directories.

> **Placement note:** For OpenAI Codex auto-discovery, store repository skills under `.agents/skills/` (or user-level `~/.codex/skills/`). A plain top-level `skills/` folder is a useful wrapper convention but is not auto-discovered by default without additional wiring.

5.5.1 Canonical Layout

Example 4-1. `.agents/skills/code-review/`

```
.agents/
  skills/
    code-review/
      SKILL.md
      manifest.json
      scripts/
        review.py
      references/
        rubric.md
      assets/
        example-diff.txt
```

Example 4-2. `.agents/skills/code-review/SKILL.md`

```
---
name: code-review
description: Review pull requests for security, correctness, and clarity.
compatibility: Requires git and Python 3.11+
allowed-tools: Bash(git:*) Read
metadata:
  author: engineering-platform
  version: "1.2"
---

# Code Review Skill

## When to use
Use this skill when reviewing pull requests for correctness, security, and clarity.

## Workflow
1. Run `scripts/review.py --pr <number>`.
2. If policy checks fail, consult `references/rubric.md`.
3. Return findings grouped by severity and file.
```

> **Tip:** Keep `SKILL.md` concise and front-load decision-critical instructions. Put deep references in `references/` and executable logic in `scripts/` so agents load content only when needed.

5.5.2 Conformance Notes: Agent Skills vs. JSON-RPC Runtime Specs

The main discrepancy you will see across docs in the wild is **where standardization happens**:

- **Agent Skills** standardizes the **artifact format** (directory + `SKILL.md` schema + optional folders).
- Some alternative specs standardize a **remote runtime API** (often JSON-RPC-style methods such as `list`, `describe`, `execute`).

In production, the Agent Skills packaging approach currently has clearer multi-tool adoption because it works in both:

1. **Filesystem-based agents** (agent can `cat` and run local scripts).
2. **Tool-based agents** (host platform loads and mediates skill content).

JSON-RPC itself is battle-tested in other ecosystems (for example, Language Server Protocol, Ethereum node APIs, and MCP transport patterns), but there are still fewer public, concrete references to large-scale deployments of a dedicated JSON-RPC **skills runtime** than to plain `SKILL.md`-based workflows. For most teams, this makes Agent Skills the safest default and JSON-RPC skill runtimes an optional layering.

5.5.3 Relationship to MCP

Use **Agent Skills** to define and distribute reusable capability packages. Use **MCP** (Model Context Protocol) to expose tools, data sources, or execution surfaces to models. In mature systems, these combine naturally: Agent Skills provide the instructions and assets that tell agents how to accomplish tasks, while MCP provides controlled runtime tool access that actually executes operations. The two standards complement each other rather than competing.

5.6 Skill Development

5.6.1 Skill Architecture

```
from typing import List, Dict, Any
from abc import ABC, abstractmethod

class Skill(ABC):
    """Base class for agent skills"""

    def __init__(self, name: str, tools: List[Tool]):
        self.name = name
        self.tools = {tool.name: tool for tool in tools}

    @abstractmethod
    async def execute(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Execute the skill"""
        pass

    def get_tool(self, name: str) -> Tool:
        """Get a tool by name"""
        return self.tools[name]

    def has_tool(self, name: str) -> bool:
        """Check if skill has a tool"""
        return name in self.tools
```

5.6.2 Example: Code Review Skill

```
class CodeReviewSkill(Skill):
    """Skill for reviewing code changes"""

    def __init__(self, llm, tools: List[Tool]):
        super().__init__("code_review", tools)
        self.llm = llm

    async def execute(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """
        Execute code review.

        Task should contain:
            - pr_number: Pull request number
            - focus_areas: List of areas to focus on (optional)
        """
        pr_number = task['pr_number']
        focus_areas = task.get('focus_areas', ['bugs', 'security', 'performance'])

        # Step 1: Get code changes
        git_diff = self.get_tool('git_diff')
```

```
        diff_result = git_diff.execute(pr_number=pr_number)

        if not diff_result['success']:
            return {'success': False, 'error': 'Failed to get diff'}

        # Step 2: Run static analysis
        analyzer = self.get_tool('static_analyzer')
        analysis_result = analyzer.execute(
            diff=diff_result['diff'],
            focus=focus_areas
        )

        # Step 3: Run tests
        test_runner = self.get_tool('test_runner')
        test_result = test_runner.execute()

        # Step 4: Generate review using LLM
        review = await self._generate_review(
            diff_result['diff'],
            analysis_result['issues'],
            test_result
        )

        return {
            'success': True,
            'review': review,
            'static_analysis': analysis_result,
            'test_results': test_result
        }

    async def _generate_review(self, diff, issues, tests):
        """Generate review using LLM"""
        prompt = f"""
        Review the following code changes:

        {diff}

        Static analysis found these issues:
        {json.dumps(issues, indent=2)}

        Test results:
        {json.dumps(tests, indent=2)}

        Provide a comprehensive code review.
        """

        return await self.llm.generate(prompt)
```

5.7 Importing and Using Skills

5.7.1 Skill Registry

```
class SkillRegistry:
    """Central registry for skills"""

    def __init__(self):
        self._skills = {}
```

```
    def register(self, skill: Skill):
        """Register a skill"""
        self._skills[skill.name] = skill

    def get(self, name: str) -> Skill:
        """Get a skill by name"""
        if name not in self._skills:
            raise ValueError(f"Skill '{name}' not found")
        return self._skills[name]

    def list_skills(self) -> List[str]:
        """List all registered skills"""
        return list(self._skills.keys())

    def import_skill(self, module_path: str, skill_class: str):
        """Dynamically import and register a skill"""
        import importlib

        module = importlib.import_module(module_path)
        SkillClass = getattr(module, skill_class)

        # Instantiate and register
        skill = SkillClass()
        self.register(skill)

# Usage
registry = SkillRegistry()

# Register built-in skills
registry.register(CodeReviewSkill(llm, tools))
registry.register(DocumentationSkill(llm, tools))

# Import external skill
registry.import_skill('external_skills.testing', 'TestGenerationSkill')

# Use a skill
code_review = registry.get('code_review')
result = await code_review.execute({'pr_number': 123})
```

5.7.2 Skill Composition

```
class CompositeSkill(Skill):
    """Skill composed of multiple sub-skills"""

    def __init__(self, name: str, skills: List[Skill]):
        self.name = name
        self.skills = {skill.name: skill for skill in skills}

        # Aggregate tools from all skills
        all_tools = []
        for skill in skills:
            all_tools.extend(skill.tools.values())

        super().__init__(name, list(set(all_tools)))

    async def execute(self, task: Dict[str, Any]) -> Dict[str, Any]:
        """Execute composed skill"""
```

```
        results = {}

        for skill_name, skill in self.skills.items():
            result = await skill.execute(task)
            results[skill_name] = result

        return {
            'success': all(r.get('success', False) for r in results.values()),
            'results': results
        }

# Create composite skill
full_review = CompositeSkill('full_review', [
    CodeReviewSkill(llm, tools),
    SecurityAuditSkill(llm, tools),
    PerformanceAnalysisSkill(llm, tools)
])
```

5.8 Tool Discovery and Documentation

5.8.1 Self-Documenting Tools

```
class DocumentedTool:
    """Tool with built-in documentation"""

    def __init__(self):
        self.name = "example_tool"
        self.description = "Example tool with documentation"
        self.parameters = {
            'required': ['param1'],
            'optional': ['param2', 'param3'],
            'schema': {
                'param1': {'type': 'string', 'description': 'Required parameter'},
                'param2': {'type': 'int', 'description': 'Optional parameter'},
                'param3': {'type': 'bool', 'description': 'Flag parameter'}
            }
        }
        self.examples = [
            {
                'input': {'param1': 'value'},
                'output': {'success': True, 'result': 'output'}
            }
        ]

    def get_documentation(self) -> str:
        """Generate documentation for this tool"""
        doc = f"# {self.name}\n\n"
        doc += f"{self.description}\n\n"
        doc += "## Parameters\n\n"

        for param, schema in self.parameters['schema'].items():
            required = "Required" if param in self.parameters['required'] else "Optional"
            doc += f"- `{param}` ({schema['type']}, {required}): {schema['description']}\n"

        doc += "\n## Examples\n\n"
        for i, example in enumerate(self.examples, 1):
            doc += f"### Example {i}\n\n"
```

```
            doc += f"Input: `{json.dumps(example['input'])}`\n\n"
            doc += f"Output: `{json.dumps(example['output'])}`\n\n"

        return doc
```

5.9 Integrations: Connecting Tools to Real-World Surfaces

Integrations sit above tools and skills. They represent packaged connectors to real systems (chat apps, device surfaces, data sources, or automation backends) that deliver a coherent user experience. Think of them as the **distribution layer** for tools and skills: they bundle auth, event routing, permissions, and UX entry points.

How integrations relate to tools and skills:

- **Tools** are atomic actions (send a message, fetch a calendar event, post to Slack).
- **Skills** orchestrate tools to solve tasks (triage inbox, compile meeting notes, run a daily report).
- **Integrations** wrap tools + skills into deployable connectors with lifecycle management (pairing, secrets, rate limits, onboarding, and UI hooks).

In practice, a single integration might expose multiple tools and enable multiple skills. The integration is the bridge between agent capabilities and the messy realities of authentication, permissions, and channel-specific constraints.

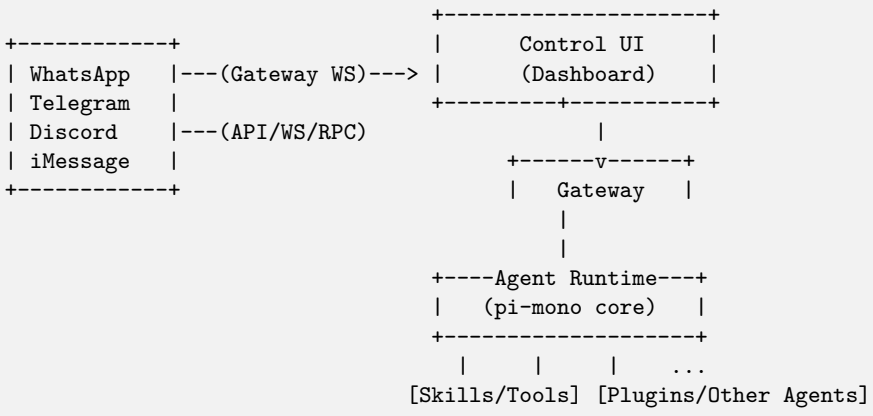
5.10 Case Study: OpenClaw and pi-mono

OpenClaw (https://openclaw.ai/, https://github.com/openclaw/openclaw) is an open-source, local-first personal AI assistant that runs a gateway control plane and connects to over 50 chat providers and device surfaces. Originally published in November 2025 as Clawdbot by Austrian software engineer Peter Steinberger, the project was renamed to OpenClaw in January 2026. With over 169,000 GitHub stars and 3,000+ community-built skills, it has become one of the most popular open-source AI projects. OpenClaw emphasizes multi-channel inboxes ("one brain, many channels"), tool access, and skill management inside a user-owned runtime.

OpenClaw is built on the **pi-mono** ecosystem (https://github.com/badlogic/pi-mono). The pi-mono monorepo provides an agent runtime, tool calling infrastructure, and multi-provider LLM APIs that OpenClaw leverages to keep the assistant portable across models and deployments.

5.10.1 OpenClaw Architecture in Detail

OpenClaw's architecture consists of several interconnected components:



1. Gateway Control Plane - Central hub orchestrating all user input/output and messaging channels - Exposes a WebSocket server (default: `ws://127.0.0.1:18789`) - Handles session state, permissions, and authentication - Supports local and mesh/LAN deployment via Tailscale (https://tailscale.com/) or similar

2. Pi Agent Runtime (pi-mono) - Core single-agent execution environment - Maintains long-lived agent state, memory, skills, and tool access - Handles multi-turn conversation, contextual memory, and tool/plugin invocation - Orchestrates external API/model calls (OpenAI, Anthropic, local models via Ollama https://ollama.com/) - Persistent storage (SQLite, Postgres, Redis) for memory and context

3. Multi-Agent Framework - Support for swarms of specialized agents ("nodes") handling domain-specific automations - Agents coordinate via shared memory and routing protocols managed by the Gateway - Each agent can be sandboxed (Docker/isolation) for security - Developers build custom agents via TypeScript/YAML plugins

4. Extensible Skills/Plugin Ecosystem - Skills expand the agent's abilities: file automation, web scraping, email, calendar - Plugins are hot-reloadable and built in TypeScript - Community skill marketplace with 3,000+ skills as of February 2026

5.10.2 Key Design Principles

1. **Privacy-First**: All state and memory default to local storage—data never leaves the device unless explicitly configured
2. **BYOM (Bring Your Own Model)**: Seamlessly supports cloud LLMs (Claude Opus 4.5 recommended) and local inference via Ollama
3. **Proactive Behavior**: "Heartbeat" feature enables autonomous wake-up and environment monitoring
4. **Persistent Memory**: Learns and adapts over long-term interactions
5. **One Brain, Many Channels**: A single AI assistant maintains shared context across all 50+ messaging channels simultaneously—message from WhatsApp on your phone, switch to Telegram on your laptop, and the same assistant remembers everything

   Warning: Security researchers have flagged local-first AI assistants as potential targets for infostealers. Without encryption-at-rest and proper containerisation, a compromised device exposes the assistant's full memory and credentials. Treat OpenClaw deployments with the same security discipline as any service handling sensitive data.

Key takeaways for skills/tools architecture:

- **Gateway + runtime separation** keeps tools and skills consistent while integrations change: the gateway handles channels and routing, while pi-mono-style runtimes handle tool execution.
- **Integration catalogs** (like OpenClaw's integrations list and skill registry) are a user-facing map of capability. They surface what tools can do and what skills are available without forcing users to understand low-level APIs.
- **Skills become reusable assets** once tied to integrations: a "Slack triage" skill can target different workspaces without changing the underlying tools, as long as the integration provides the same tool contracts.

5.10.3 The Personal AI Ecosystem Beyond OpenClaw

OpenClaw is the largest project in a rapidly growing personal AI assistant category. Several related frameworks share its local-first philosophy while making different architectural trade-offs.

Letta (https://www.letta.com/, formerly MemGPT) is a platform for building stateful agents with advanced memory that can learn and self-improve over time. In January 2026, Letta shipped a Conversations API for agents with shared memory across parallel user experiences, and its **Letta Code** agent ranked #1 on Terminal-Bench among model-agnostic open-source agents. Letta's architecture emphasises programmable memory management—where OpenClaw focuses on channel integration and skills, Letta focuses on making agents that remember and adapt intelligently. The **LettaBot** project (https://github.com/letta-ai/lettabot)

brings Letta's memory capabilities to a multi-channel personal assistant supporting Telegram, Slack, Discord, WhatsApp, and Signal.

Langroid (https://langroid.github.io/langroid/) is a Python multi-agent framework from CMU and UW-Madison researchers that emphasises simplicity and composability. As of early 2026, Langroid has enhanced MCP support with persistent connections, Portkey integration for unified access to 200+ LLMs, and declarative task termination patterns. Its architecture treats agents, tasks, and tools as lightweight composable objects, making it well-suited for teams that want multi-agent orchestration without heavy infrastructure.

Open Interpreter (https://github.com/openinterpreter/open-interpreter) provides a natural language interface for controlling computers. Its "New Computer Update" (late 2024) was a complete rewrite supporting a standard interface between language models and computer operations. While less focused on multi-channel messaging than OpenClaw, Open Interpreter fills a complementary niche: using an LLM to drive local computer actions (file management, browser automation, system administration) through plain language.

Leon (https://getleon.ai/) is an open-source personal assistant built in JavaScript with natural speech recognition, task management, and extendable skills. It is installable via npm on Linux, Mac, or Windows, and appeals to developers who want a lightweight, self-hosted assistant without the full complexity of OpenClaw's multi-channel architecture.

These projects collectively represent a broad trend: users increasingly expect AI assistants that run locally, remember context across sessions and channels, and respect data privacy by default. The architectural patterns that OpenClaw popularised—gateway/runtime separation, plugin-based skills, model-agnostic backends—are now standard across the category.

5.11 Related Architectures and Frameworks

Several other frameworks share architectural patterns with OpenClaw:

5.11.1 LangChain and LangGraph

LangChain (https://docs.langchain.com) provides composable building blocks for LLM applications. As of 2026, LangChain and LangGraph have both reached v1.0 milestones.

> **Snippet status:** Runnable example pattern (validated against LangChain v1 docs, Feb 2026; `create_agent` builds a graph-based agent runtime using LangGraph under the hood).

```
from langchain.agents import create_agent
from langchain_core.tools import tool

@tool
def search_documentation(query: str) -> str:
    """Search project documentation for relevant information."""
    # Implementation
    return "..."

agent = create_agent(
    model="gpt-4o-mini",
    tools=[search_documentation],
    system_prompt="Use tools when needed, then summarize clearly.",
)
result = agent.invoke({"messages": [{"role": "user", "content": "Find testing docs"}]})
```

Shared patterns with OpenClaw: Tool registration, agent composition, memory management.

LangGraph (https://langchain-ai.github.io/langgraph/) extends LangChain with graph-based agent orchestration.

5.11.2 CrewAI

CrewAI (https://docs.crewai.com/) focuses on multi-agent collaboration with role-based specialization:

```
from crewai import Agent, Task, Crew

researcher = Agent(
    role='Senior Researcher',
    goal='Discover new insights',
    backstory='Expert in finding and analyzing information',
    tools=[search_tool, analysis_tool]
)

writer = Agent(
    role='Technical Writer',
    goal='Create clear documentation',
    backstory='Skilled at explaining complex topics',
    tools=[writing_tool]
)

crew = Crew(
    agents=[researcher, writer],
    tasks=[research_task, writing_task],
    process=Process.sequential
)
```

Shared patterns with OpenClaw: Role-based agents, sequential and parallel execution, tool assignment per agent.

5.11.3 Microsoft Semantic Kernel

Semantic Kernel (https://learn.microsoft.com/semantic-kernel/) emphasizes enterprise integration and plugin architecture. As of late 2025, Semantic Kernel is converging with AutoGen into the **Microsoft Agent Framework** (see AutoGen section below):

```
var kernel = Kernel.CreateBuilder()
    .AddOpenAIChatCompletion("gpt-4", apiKey)
    .Build();

// Import plugins
kernel.ImportPluginFromType<TimePlugin>();
kernel.ImportPluginFromType<FileIOPlugin>();

// Create agent with plugins
var agent = new ChatCompletionAgent {
    Kernel = kernel,
    Name = "ProjectAssistant",
    Instructions = "Help manage project tasks and documentation"
};
```

Shared patterns with OpenClaw: Plugin system, kernel/runtime separation, enterprise-ready design.

5.11.4 AutoGen / Microsoft Agent Framework

AutoGen (https://microsoft.github.io/autogen/stable/) was rewritten from the ground up as v0.4 in January 2025, adopting an asynchronous, event-driven architecture. In October 2025, Microsoft announced the convergence of AutoGen and Semantic Kernel into a unified **Microsoft Agent Framework**, with general availability scheduled for Q1 2026. AutoGen v0.4 continues to receive critical fixes, but significant new features target the unified framework.

Snippet status: Runnable example pattern (AutoGen v0.4 API, Feb 2026).

```
import asyncio
from autogen_agentchat.agents import AssistantAgent
from autogen_ext.models.openai import OpenAIChatCompletionClient

async def main() -> None:
    agent = AssistantAgent(
        "coding_assistant",
        OpenAIChatCompletionClient(model="gpt-4o"),
    )
    result = await agent.run(task="Create a Python web scraper")
    print(result)

asyncio.run(main())
```

Note: The v0.2 API (`from autogen import AssistantAgent`) is deprecated. Migrate to `autogen_agentchat` and `autogen_ext` packages. See the migration guide at https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/migration-guide.html.

Shared patterns with OpenClaw: Agent-to-agent communication, code execution environments, conversation-driven workflows.

5.11.5 Comparing Architecture Patterns

Feature	OpenClaw	LangChain	CrewAI	MS Agent Framework†
Primary Focus	Personal assistant	LLM app building	Team collaboration	Enterprise agents
Runtime	Local-first	Flexible	Python process	.NET/Python
Multi-Agent	Via swarms	Via LangGraph	Built-in	Built-in
Tool System	Plugin-based	Tool decorators	Tool assignment	Plugin imports
Memory	Persistent local	Configurable	Per-agent	Configurable
Best For	Personal automation	Prototyping	Complex workflows	Enterprise apps

† Microsoft Agent Framework is the convergence of Semantic Kernel and AutoGen, announced October 2025.

5.11.6 OpenAI Agents SDK

The OpenAI Agents SDK (https://openai.github.io/openai-agents-python/) is the production-ready successor to the experimental Swarm project, launched March 2025. It provides easily configurable agents with instructions and built-in tools, agent handoffs for intelligent control transfer, built-in guardrails, and tracing for debugging. Available in both Python and TypeScript.

Shared patterns with OpenClaw: Agent handoffs, tool registration, guardrails, conversation-driven workflows.

5.11.7 Google Agent Development Kit (ADK)

Google ADK (https://google.github.io/adk-docs/) is an open-source framework introduced at Cloud NEXT 2025 for developing multi-agent systems. It is model-agnostic (optimised for Gemini but compatible with other providers) and supports the Agent-to-Agent (A2A) protocol for inter-agent communication. Primary SDK is Python; TypeScript and Go SDKs are in active development.

Shared patterns with OpenClaw: Multi-agent orchestration, model-agnostic design, tool registration.

5.12 MCP: Modern Tooling and Adoption

The **Model Context Protocol (MCP)** (https://modelcontextprotocol.io/) has become a practical standard for connecting agents to tools and data sources. In December 2025, Anthropic donated MCP governance to the **Agentic AI Foundation (AAIF)** under the Linux Foundation, signalling its transition from a single-vendor project to a true industry standard. As of early 2026, the ecosystem reports over 97 million monthly SDK downloads and more than 10,000 active MCP servers. In January 2026, **MCP Apps** launched as the first official extension, enabling interactive UIs (charts, forms, dashboards) to render directly inside MCP clients. Today, MCP is less about novel capability and more about **reliable interoperability**: the same tool server can be used by multiple agent clients with consistent schemas, permissions, and response formats.

5.12.1 What MCP Brings to Tools

- **Portable tool definitions**: JSON schemas and well-known server metadata make tools discoverable across clients.
- **Safer tool execution**: capability-scoped permissions, explicit parameters, and auditable tool calls.
- **Composable context**: servers can enrich model context with structured resources (files, APIs, or databases) without bespoke glue code.

5.12.2 Common Usage Patterns

1. **Server-based tool catalogs**
   - Teams deploy MCP servers per domain (e.g., "repo-tools", "ops-tools", "research-tools").
   - Clients discover available tools at runtime and choose based on metadata, not hardcoded lists.
2. **Context stitching**
   - Agents gather context from multiple servers (docs, tickets, metrics) and assemble it into a task-specific prompt.
   - The server provides structured resources so the client can keep the prompt lean.
3. **Permission-first workflows**
   - Tool calls are scoped by project, environment, or role.
   - Audit logs track who called what tool with which inputs.
4. **Fallback-first reliability**
   - Clients maintain fallbacks when a server is down (cached data, read-only mirrors, or alternative tool servers).

5.12.3 Acceptance Across Major Clients

MCP is broadly accepted as a **tooling interoperability layer**. The specifics vary by vendor, but the pattern is consistent: MCP servers expose the tools and resources, while clients orchestrate tool calls and manage safety policies.

- **Codex (GPT-5.3-Codex)** (https://openai.com/index/introducing-codex/) Codex clients commonly use MCP servers to standardize tool access (repo browsing, test execution, task automation). Codex also supports skills packaged with `SKILL.md` and progressive disclosure (see https://developers.opena

i.com/codex). The main adoption pattern is organization-level MCP servers that provide consistent tools across multiple repos.

- **GitHub Copilot** (https://docs.github.com/en/copilot)
  Copilot deployments increasingly treat MCP as a bridge between editor experiences and organization tooling. This typically means MCP servers that expose repo-aware tools (search, CI status, documentation retrieval) so the assistant can operate with consistent, policy-driven access.
- **Claude** (https://code.claude.com/docs)
  Claude integrations often use MCP to provide structured context sources (knowledge bases, issue trackers, dashboards). The MCP server becomes the policy boundary, while the client focuses on prompt composition and response quality.

5.12.4 Practical Guidance for Authors and Teams

- **Document your MCP servers** like any other tool: include schemas, permissions, and usage examples.
- **Version tool contracts** so clients can adopt changes incrementally.
- **Prefer narrow, composable tools** over large monolithic endpoints.
- **Treat MCP as infrastructure**: invest in uptime, monitoring, and security reviews.

5.13 Best Practices

5.13.1 Version Tools and Skills

```
class VersionedTool:
    def __init__(self, version: str):
        self.version = version
        self.name = f"{self.__class__.__name__}_v{version}"
```

5.13.2 Test Independently

```
# test_tools.py
import pytest

def test_markdown_validator():
    tool = MarkdownValidatorTool()

    # Test valid markdown
    valid_md = "# Header\n\nContent"
    result = tool.execute(valid_md)
    assert result['valid']

    # Test invalid markdown
    invalid_md = "```python\ncode without closing"
    result = tool.execute(invalid_md)
    assert not result['valid']
    assert any(i['type'] == 'unclosed_code_block' for i in result['issues'])
```

5.13.3 Provide Fallbacks

```
class ResilientTool:
    def __init__(self, primary_impl, fallback_impl):
        self.primary = primary_impl
        self.fallback = fallback_impl
```

```
    def execute(self, **kwargs):
        try:
            return self.primary.execute(**kwargs)
        except Exception as e:
            logger.warning(f"Primary implementation failed: {e}")
            return self.fallback.execute(**kwargs)
```

5.13.4 Monitor Usage

```
class MonitoredTool:
    def __init__(self, tool, metrics_collector):
        self.tool = tool
        self.metrics = metrics_collector

    def execute(self, **kwargs):
        start = time.time()
        try:
            result = self.tool.execute(**kwargs)
            self.metrics.record_success(self.tool.name, time.time() - start)
            return result
        except Exception as e:
            self.metrics.record_failure(self.tool.name, str(e))
            raise
```

5.14 Emerging Standards: AGENTS.md

This chapter is the canonical AGENTS.md reference for the book. Other chapters should link here rather than duplicating full templates.

5.14.1 The AGENTS.md Pseudo-Standard

AGENTS.md has emerged as an open pseudo-standard for providing AI coding agents with project-specific instructions. Think of it as a "README for agents"—offering structured, machine-readable guidance that helps agents understand how to work within a codebase.

5.14.1.1 Purpose and Benefits

- **Consistent Instructions**: All agents receive the same project-specific guidance
- **Rapid Onboarding**: New agent sessions understand the project immediately
- **Safety Boundaries**: Clear boundaries prevent accidental damage to protected files
- **Maintainability**: Single source of truth for agent behavior in a project

5.14.1.2 Structure and Placement

AGENTS.md files can be placed hierarchically in a project:

```
project/
|-- AGENTS.md           # Root-level instructions (project-wide)
|-- src/
|   `-- AGENTS.md       # Module-specific instructions
|-- tests/
|   `-- AGENTS.md       # Testing conventions
`-- docs/
    `-- AGENTS.md       # Documentation guidelines
```

Agents use the nearest AGENTS.md file, enabling scoped configuration for monorepos or complex projects.

5.14.1.3 Example AGENTS.md

```
# AGENTS.md

## Project Overview
This is a TypeScript web application using Express.js and React.

## Setup Instructions
npm install
npm run dev

## Coding Conventions
- Language: TypeScript 5.x
- Style guide: Airbnb
- Formatting: Prettier with provided config
- Test framework: Jest

## Build and Deploy
- Build: `npm run build`
- Test: `npm test`
- Deploy: CI/CD via GitHub Actions

## Agent-Specific Notes
- Always run `npm run lint` before committing
- Never modify files in `vendor/` or `.github/workflows/`
- Secrets are in environment variables, never hardcoded
- All API endpoints require authentication middleware
```

5.14.2 Related Standards: Skills and Capabilities

While AGENTS.md has achieved broad adoption as the standard for project-level agent instructions, the space continues to evolve. Several related concepts are under discussion in the community:

Skills Documentation

There is no formal `skills.md` standard, but skill documentation patterns are emerging:

- **Skill catalogs** listing available agent capabilities
- **Capability declarations** specifying what an agent can do
- **Dependency manifests** defining tool and skill requirements

Personality and Values

Some frameworks experiment with "soul" or personality configuration. Note that "soul" is a metaphorical term used in some AI agent frameworks to describe an agent's core personality, values, and behavioral guidelines—it's industry jargon rather than a formal technical specification:

- **System prompts** defining agent persona and communication style
- **Value alignment** specifying ethical guidelines and constraints
- **Behavioral constraints** limiting what agents should and shouldn't do

Currently, these are implemented in vendor-specific formats rather than open standards. The community continues to discuss whether formalization is needed.

5.15 How Agents Become Aware of Imports

One of the most practical challenges in agentic development is helping agents understand a codebase's import structure and dependencies. When an agent modifies code, it must know what modules are available, where they come from, and how to properly reference them.

5.15.1 The Import Awareness Problem

When agents generate or modify code, they face several import-related challenges:

1. **Missing imports**: Adding code that uses undefined symbols
2. **Incorrect import paths**: Using wrong relative or absolute paths
3. **Circular dependencies**: Creating imports that cause circular reference errors
4. **Unused imports**: Leaving orphan imports after code changes
5. **Conflicting names**: Importing symbols that shadow existing names

5.15.2 Mechanisms for Import Discovery

Modern coding agents use multiple strategies to understand imports:

5.15.2.1 Static Analysis Tools

Agents leverage language servers and static analyzers to understand import structure:

```
class ImportAnalyzer:
    """Analyze imports using static analysis"""

    def __init__(self, workspace_root: str):
        self.workspace = workspace_root
        self.import_graph = {}

    def analyze_file(self, filepath: str) -> dict:
        """Extract import information from a file"""
        with open(filepath) as f:
            content = f.read()

        # Parse AST to find imports
        tree = ast.parse(content)
        imports = []

        for node in ast.walk(tree):
            if isinstance(node, ast.Import):
                for alias in node.names:
                    imports.append({
                        'type': 'import',
                        'module': alias.name,
                        'alias': alias.asname
                    })
            elif isinstance(node, ast.ImportFrom):
                imports.append({
                    'type': 'from_import',
                    'module': node.module,
                    'names': [a.name for a in node.names],
                    'level': node.level  # relative import level
                })

        return {
            'file': filepath,
            'imports': imports,
            'defined_symbols': self._extract_definitions(tree)
        }

    def build_dependency_graph(self) -> dict:
        """Build a graph of all file dependencies"""
        for filepath in self._find_source_files():
```

```
            analysis = self.analyze_file(filepath)
            self.import_graph[filepath] = analysis
        return self.import_graph
```

5.15.2.2 Language Server Protocol (LSP)

Language servers provide real-time import information that agents can query:

```
class LSPImportProvider:
    """Use LSP to discover available imports"""

    async def get_import_suggestions(self, symbol: str, context_file: str) -> list:
        """Get import suggestions for an undefined symbol"""

        # Query language server for symbol locations
        response = await self.lsp_client.request('textDocument/codeAction', {
            'textDocument': {'uri': context_file},
            'context': {
                'diagnostics': [{
                    'message': f"Cannot find name '{symbol}'"
                }]
            }
        })

        # Extract import suggestions from code actions
        suggestions = []
        for action in response:
            if 'import' in action.get('title', '').lower():
                suggestions.append({
                    'import_statement': action['edit']['changes'],
                    'source': action.get('title')
                })

        return suggestions

    async def get_exported_symbols(self, module_path: str) -> list:
        """Get all exported symbols from a module"""

        # Use workspace/symbol to find exports
        symbols = await self.lsp_client.request('workspace/symbol', {
            'query': '',
            'uri': module_path
        })

        return [s['name'] for s in symbols if s.get('kind') in EXPORTABLE_KINDS]
```

5.15.2.3 Project Configuration Files

Agents read configuration files to understand module resolution:

```
class ProjectConfigReader:
    """Read project configs to understand import paths"""

    def get_import_config(self, project_root: str) -> dict:
        """Extract import configuration from project files"""

        config = {
            'base_paths': [],
```

```
            'aliases': {},
            'external_packages': []
        }

        # TypeScript/JavaScript: tsconfig.json, jsconfig.json
        tsconfig_path = os.path.join(project_root, 'tsconfig.json')
        if os.path.exists(tsconfig_path):
            with open(tsconfig_path) as f:
                tsconfig = json.load(f)

            compiler_opts = tsconfig.get('compilerOptions', {})
            config['base_paths'].append(compiler_opts.get('baseUrl', '.'))
            config['aliases'] = compiler_opts.get('paths', {})

        # Python: pyproject.toml, setup.py
        pyproject_path = os.path.join(project_root, 'pyproject.toml')
        if os.path.exists(pyproject_path):
            with open(pyproject_path) as f:
                pyproject = toml.load(f)

            # Extract package paths from tool.setuptools or poetry config
            if 'tool' in pyproject:
                if 'setuptools' in pyproject['tool']:
                    config['base_paths'].extend(
                        pyproject['tool']['setuptools'].get('package-dir', {}).values()
                    )

        return config
```

5.15.2.4 Package Manifest Analysis

Agents check package manifests to know what's available:

```
class PackageManifestReader:
    """Read package manifests to understand available dependencies"""

    def get_available_packages(self, project_root: str) -> dict:
        """Get list of available packages from manifest"""

        packages = {'direct': [], 'transitive': []}

        # Node.js: package.json
        package_json = os.path.join(project_root, 'package.json')
        if os.path.exists(package_json):
            with open(package_json) as f:
                pkg = json.load(f)
            packages['direct'].extend(pkg.get('dependencies', {}).keys())
            packages['direct'].extend(pkg.get('devDependencies', {}).keys())

        # Python: requirements.txt, Pipfile, pyproject.toml
        requirements = os.path.join(project_root, 'requirements.txt')
        if os.path.exists(requirements):
            with open(requirements) as f:
                for line in f:
                    line = line.strip()
                    if line and not line.startswith('#'):
                        # Extract package name (before version specifier)
                        pkg_name = re.split(r'[<>=!]', line)[0].strip()
                        packages['direct'].append(pkg_name)
```

```
        return packages
```

5.15.3 Best Practices for Import-Aware Agents

5.15.3.1 Document Import Conventions in AGENTS.md

For standardized terminology (artefact, discovery, import, install, activate) and trust boundaries, see Discovery and Imports. In this section we apply those concepts to codebase-level import resolution.

Include import guidance in your project's AGENTS.md:

```
## Import Conventions

### Path Resolution
- Use absolute imports from `src/` as the base
- Prefer named exports over default exports
- Group imports: stdlib, external packages, local modules

### Example Import Order
```python
# Standard library
import os
import sys
from typing import Dict, List

# Third-party packages
import requests
from pydantic import BaseModel

# Local modules
from src.utils import helpers
from src.models import User
```

5.15.4 Alias Conventions

- `@/` maps to `src/`
- `@components/` maps to `src/components/`

```
#### Use Import Auto-Fix Tools

Configure agents to use automatic import fixers:

```python
class ImportAutoFixer:
    """Automatically fix import issues in agent-generated code"""

    def __init__(self, tools: List[Tool]):
        self.isort = tools.get('isort')  # Python import sorting
        self.eslint = tools.get('eslint')  # JS/TS import fixing

    async def fix_imports(self, filepath: str) -> dict:
        """Fix and organize imports in a file"""

        results = {'fixed': [], 'errors': []}

        if filepath.endswith('.py'):
```

```
            # Run isort for Python
            result = await self.isort.execute(filepath)
            if result['success']:
                results['fixed'].append('isort: organized imports')

            # Run autoflake to remove unused imports
            result = await self.autoflake.execute(
                filepath,
                remove_unused_imports=True
            )
            if result['success']:
                results['fixed'].append('autoflake: removed unused')

        elif filepath.endswith(('.ts', '.tsx', '.js', '.jsx')):
            # Run eslint with import rules
            result = await self.eslint.execute(
                filepath,
                fix=True,
                rules=['import/order', 'unused-imports/no-unused-imports']
            )
            if result['success']:
                results['fixed'].append('eslint: fixed imports')

        return results
```

5.15.4.1 Validate Imports Before Committing

Add import validation to agent workflows:

```
# .github/workflows/validate-imports.yml
name: Validate Imports
on: [pull_request]

jobs:
  check-imports:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6

      - name: Check Python imports
        run: |
          pip install isort autoflake
          isort --check-only --diff .
          autoflake --check --remove-all-unused-imports -r .

      - name: Check TypeScript imports
        run: |
          npm ci
          npx eslint --rule 'import/no-unresolved: error' .
```

5.15.5 Import Awareness in Multi-Agent Systems

When multiple agents collaborate, maintaining consistent import awareness requires coordination:

```
class SharedImportContext:
    """Shared import context for multi-agent systems"""

    def __init__(self):
```

```
        self.import_cache = {}
        self.pending_additions = []

    def register_new_export(self, module: str, symbol: str, agent_id: str):
        """Register a new export created by an agent"""
        if module not in self.import_cache:
            self.import_cache[module] = []

        self.import_cache[module].append({
            'symbol': symbol,
            'added_by': agent_id,
            'timestamp': datetime.now()
        })

    def query_available_imports(self, symbol: str) -> List[dict]:
        """Query where a symbol can be imported from"""
        results = []
        for module, exports in self.import_cache.items():
            for export in exports:
                if export['symbol'] == symbol:
                    results.append({
                        'module': module,
                        'symbol': symbol,
                        'import_statement': f"from {module} import {symbol}"
                    })
        return results
```

Understanding how agents discover and manage imports is essential for building reliable agentic coding systems. The combination of static analysis, language servers, project configuration, and clear documentation ensures agents can write code that integrates correctly with existing codebases.

5.16 Key Takeaways

Tools are atomic capabilities that perform single operations, while skills are composed behaviours that orchestrate multiple tools to accomplish complex tasks. When designing tools, follow single responsibility principles and provide clear interfaces that agents can use reliably. Skills orchestrate multiple tools to accomplish complex tasks, and they can themselves be composed to create more powerful capabilities.

Use registries for discovery and management, allowing agents to find available tools and skills at runtime. Always document, test, and version your tools and skills so that changes are traceable and consumers know what to expect. Monitor usage to identify issues and optimisation opportunities—without metrics, you cannot improve performance or reliability.

AGENTS.md is the emerging standard for project-level agent instructions, providing a single source of truth for how agents should work within a codebase. Skills Protocol defines how runtimes execute skills, while Agent Skills defines how skills are packaged for distribution. MCP standardises tool interoperability across clients and hosts, allowing the same tool server to work with multiple agent platforms.

Import awareness requires combining static analysis, Language Server Protocol (LSP) integration, and project configuration reading to ensure agents generate code with correct dependencies. OpenClaw, LangChain, CrewAI, and similar frameworks share common patterns for tool and skill management that you can learn from regardless of which platform you choose.

Chapter 6

Discovery and Imports

6.1 Chapter Preview

This chapter standardises language that often gets overloaded in agentic systems. We define what an artefact is, what discovery means in practice, and how import, install, and activate are separate operations with different safety controls.

The chapter builds a taxonomy for what gets discovered—tools, skills, agents, and workflow fragments—providing clear definitions that prevent confusion as systems grow more complex. It compares discovery mechanisms including local scans, registries, domain conventions, and explicit configuration, explaining the trade-offs of each approach. Finally, it disambiguates the operations of import, install, and activate, mapping each step to appropriate trust and supply-chain controls.

6.2 Terminology Baseline for the Rest of the Book

To keep later chapters precise, this chapter uses five core terms consistently.

An **artefact** is any reusable unit a workflow can reference, including tool endpoint metadata, a skill bundle, an agent definition, or a workflow fragment. **Discovery** is the process of finding candidate artefacts that might be useful for a given task. **Import** means bringing a discovered artefact into the current resolution or evaluation context so it can be referenced. **Install** means fetching and persisting an artefact (typically pinned to a specific version), along with integrity metadata such as checksums or signatures. **Activate** means making an installed or imported artefact callable by an agent in a specific run context.

> **Standardisation rule:** Use these verbs literally. Do not use "import" when you mean "install," and do not use "install" when you mean "activate." Precise terminology prevents misunderstandings about what security controls apply at each stage.

6.3 A Taxonomy That Disambiguates What We Are Discovering

6.3.1 1) Tool artefacts

A **tool** is an executable capability exposed via a protocol or command surface. A tool's **identity** is its endpoint identity, such as an MCP server URL combined with an authentication context. A tool's **interface** consists of the enumerated callable operations it exposes, including their names, schemas, and permission requirements.

Discovery usually finds endpoints first; tool enumeration happens after connection, when the client can query what operations the tool server supports.

6.3.2 2) Skill artefacts

A **skill** is a packaged reusable bundle of instructions, templates, and optional scripts. A skill's **identity** is its bundle source, which may be a repository path, a registry coordinate, or a version and digest combination. A skill's **interface** comprises its documented entrypoints, expected inputs and outputs, and policy constraints that govern how it may be used.

6.3.3 3) Agent artefacts

An **agent** artefact is a role and configuration definition that specifies a persona, constraints, and operating policy. An agent's **identity** is a named definition file and version. An agent's **interface** includes its responsibilities, boundaries, and the set of capabilities it is allowed to use.

6.3.4 4) Workflow-fragment artefacts

A **workflow fragment** is a reusable partial workflow, such as a GH-AW component that can be imported into other workflows. A workflow fragment's **identity** is its source file path or import address. A workflow fragment's **interface** includes its parameters, the context it expects, and the outputs it emits.

6.3.5 Confusing cases to stop using

Several common conflations cause confusion and should be avoided. A **tool is not the same as a skill**: tools execute capabilities, while skills package guidance and assets that tell agents how to use tools effectively. A **skill is not the same as an agent**: skills are reusable bundles of instructions, while agents are operating roles that may use skills. An **agent is not the same as a workflow fragment**: an agent is an actor that performs work, while a fragment is orchestration structure that defines how work flows between actors.

When this book says "discover capabilities," read it as "discover artefacts, then import, install, or activate according to type."

6.4 Discovery Mechanisms

Discovery is how runtimes gather candidate artefacts before selection. Different mechanisms suit different contexts.

6.4.1 Local scan

Local scanning examines repository paths and conventions (for example, `.github/workflows/`, `skills/`, `agents/`) to find artefacts available within the codebase. The advantages are low latency, high transparency, and easy review in code—everything is visible in the repository. The disadvantages are limited scope and convention drift in large monorepos, where different teams may adopt inconsistent conventions.

6.4.2 Registry discovery

Registry discovery queries a curated index or marketplace for artefacts. The advantages include centralised metadata, version visibility, and governance hooks that can enforce organisational policy. The disadvantages are that trust shifts to registry policy (the registry becomes a critical dependency), and namespace collisions are possible when multiple teams use similar names.

6.4.3 Domain-convention discovery

Domain-convention discovery resolves artefacts via domain naming conventions, such as `.well-known`-style descriptors that expose capability metadata at predictable URLs. The advantage is interoperable discovery across organisational boundaries—you can discover capabilities from external partners using a standard protocol. The disadvantage is that conventions may be ecosystem-specific and are not always standardised across vendors.

6.4.4 Explicit configuration

Explicit configuration uses a pinned manifest that enumerates allowed sources and versions. The advantages are strongest reproducibility and auditability—you know exactly what artefacts will be used. The disadvantages are less flexibility and the need for deliberate updates whenever artefacts change.

Decision rule: If provenance cannot be authenticated, prefer explicit configuration over dynamic discovery. Security concerns outweigh convenience when you cannot verify where an artefact came from.

6.5 Import, Install, Activate: Three Different Operations

6.5.1 Import

Import brings an artefact into the current resolution context. In language and module terms, this looks like `from src.utils import helpers`. In GH-AW terms, this looks like `imports: [shared/common-tools.md]`. Import makes the artefact available for reference but does not necessarily make it callable.

6.5.2 Install

Install fetches and persists artefacts for repeatable use. For example, you might store `skill-x@1.4.2` with checksum and signature metadata to ensure integrity. Another example is locking a workflow component revision to a commit digest so that future runs use the exact same version.

6.5.3 Activate

Activate makes an artefact callable under policy. For example, you might expose only `bash` and `edit` tools to a CI agent, withholding more dangerous capabilities. Another example is enabling a skill only after an approval gate passes, ensuring human oversight for high-impact operations.

A practical sequence is often: **discover → select → import/install → activate → execute**.

6.6 Trust Boundaries and Supply Chain (Compact Model)

Each stage has distinct risks and controls.

Integrity addresses whether an artefact was tampered with, and the controls are checksums and signatures. **Authenticity** addresses who published the artefact, and the controls are identity verification and trusted publisher lists. **Provenance** addresses how an artefact was built, and the controls are attestations, software bills of materials (SBOM), and reproducible build metadata. **Capability safety** addresses what an artefact can do, and the controls are least privilege, sandboxing, and constrained outputs.

The control mapping by stage is as follows. **Discovery controls** include allowlists of domains and registries. **Import and install controls** include pinning plus checksum and signature verification. **Activation controls** include permission gates, scoped credentials, and sandbox profiles. **Runtime controls** include audit logs, safe outputs, and policy evaluation traces.

6.7 Worked Examples

6.7.1 Example A: GH-AW workflow-fragment import

```
# .github/workflows/docs-refresh.md
name: Docs Refresh
on:
  workflow_dispatch:
imports:
  - shared/common-tools.md
```

```
permissions:
  contents: read
```

In this example, `shared/common-tools.md` is a **workflow-fragment artefact**. The `imports` directive is the **import** operation. A separate policy decides whether imported tools are **activated** at runtime.

6.7.2 Example B: AGENTS.md import conventions as resolution policy

```
## Import Conventions
- Prefer absolute imports from `src/`
- Group imports: stdlib, third-party, local
- Use `@/` alias for `src/`
```

This example does **not** install dependencies. It standardises **import resolution behaviour** so agents generate consistent code. Activation still depends on tool and runtime permissions.

6.7.3 Example C: Capability discovery and activation policy

```
# policies/capability-sources.yml
allowed_domains:
  - "skills.example.com"
allowed_registries:
  - "registry.internal/agent-skills"
pinned:
  "registry.internal/agent-skills/reviewer": "2.3.1"
checksums:
  "registry.internal/agent-skills/reviewer@2.3.1": "sha256:..."
activation_gates:
  require_human_approval_for:
    - "github.write"
    - "bash.exec"
```

In this example, discovery scope is constrained first by the allowed domains and registries. Import and install are pinned and integrity-checked via the pinned versions and checksums. High-impact capabilities require explicit activation approval through the activation gates.

6.8 Key Takeaways

Treat **artefact**, **discovery**, **import**, **install**, and **activate** as distinct terms with precise meanings. Discovering a tool endpoint is not the same as activating its capabilities—each stage requires different security controls. Use taxonomy-first language: tool, skill, agent, and workflow fragment are different artefact types with different identity and interface properties.

Prefer explicit, pinned configuration when provenance or authenticity is uncertain; the convenience of dynamic discovery is not worth the security risk. Apply controls by stage: allowlist at discovery, verify at import and install, and enforce least privilege at activation and runtime.

For tool and skill design conventions, see Skills and Tools Management. For GH-AW composition syntax, see GitHub Agentic Workflows (GH-AW).

Chapter 7

GitHub Agentic Workflows (GH-AW)

7.1 Chapter Preview

This chapter explains how GH-AW compiles markdown into deterministic workflows that GitHub Actions can execute. It shows how to set up GH-AW with the supported setup actions, including both vendored and upstream approaches. Finally, it highlights the safety controls that make agentic workflows production-ready: permissions, safe outputs, and approval gates.

7.2 Why GH-AW Matters

GitHub Agentic Workflows (GH-AW) (https://github.github.io/gh-aw/) turns natural language into automated repository agents that run inside GitHub Actions. Instead of writing large YAML pipelines by hand, you write markdown instructions that an AI agent executes with guardrails. The result is a workflow you can read like documentation but run like automation.

At a glance, GH-AW provides several key capabilities. **Natural language workflows** allow you to write markdown instructions that drive the agent's behaviour, making automation readable to humans. **Compile-time structure** means your markdown is compiled into GitHub Actions workflows, ensuring reproducibility across runs. **Security boundaries** let you define permissions, tools, and safe outputs that constrain what the agent can and cannot do. **Composable automation** enables imports and shared components that you can reuse across repositories.

7.3 Core Workflow Structure

A GH-AW workflow is a markdown file with frontmatter and instructions:

```
---
on:
  issues:
    types: [opened]
permissions:
  contents: read
tools:
  edit:
  github:
    toolsets: [issues]
engine: copilot
---
```

```
# Triage this issue
Read issue #${{ github.event.issue.number }} and summarize it.
```

Key parts:

The **frontmatter** section configures the workflow's behaviour. The `on` field specifies GitHub Actions triggers such as issues, schedules, or dispatch events. The `permissions` field declares least-privilege access to GitHub APIs, ensuring the agent can only perform authorised operations. The `tools` field lists the capabilities your agent can invoke, such as edit, bash, web, or github. The `engine` field specifies the AI model or provider to use, such as Copilot, Claude Code, or Codex.

The **markdown instructions** section contains natural language steps for the agent to follow. You can include context variables from the event payload, such as issue number, PR number, or repository name, using template syntax.

7.4 How GH-AW Runs

GH-AW compiles markdown workflows into `.lock.yml` GitHub Actions workflows. The compiled file is what GitHub actually executes, but the markdown remains the authoritative source. This gives you readable automation with predictable execution.

7.4.1 File Location

Both the source markdown files and the compiled `.lock.yml` files live in the `.github/workflows/` directory:

```
.github/workflows/
|-- triage.md          # Source (human-editable)
|-- triage.lock.yml    # Compiled (auto-generated, do not edit)
|-- docs-refresh.md
`-- docs-refresh.lock.yml
```

Use `gh aw compile` (from the GH-AW CLI at https://github.com/github/gh-aw) in your repository root to generate `.lock.yml` files from your markdown sources. Only edit the `.md` files; the `.lock.yml` files are regenerated on compile.

If you do not vendor the GH-AW `actions/` directory in your repository, you can instead reference the upstream setup action directly (pin to a commit SHA for security):

```
- name: Setup GH-AW scripts
  uses: github/gh-aw/actions/setup@5a4d651e3bd33de46b932d898c20c5619162332e
  with:
    destination: /opt/gh-aw/actions
```

7.4.2 Key Behaviors

There are three key behaviours to understand about the compilation model. First, **frontmatter edits require recompile**—any changes to triggers, permissions, tools, or engine settings must be followed by running `gh aw compile` to regenerate the lock file. Second, **markdown instruction updates can often be edited directly** because the runtime loads the markdown body at execution time; however, structural changes may still require recompilation. Third, **shared components** can be stored as markdown files without an `on:` trigger; these are imported rather than compiled, allowing reuse without duplication.

7.4.3 Compilation Pitfalls

GH-AW compilation is predictable, but a few pitfalls are common in real repositories.

Only compile workflow markdown. The compiler expects frontmatter with an `on:` trigger. Non-workflow files like `AGENTS.md` or general docs should not be passed to `gh aw compile`. Use `gh aw compile <workflow-id>` to target specific workflows when the directory includes other markdown files.

Strict mode rejects direct write permissions. GH-AW runs in strict mode by default; you can opt out by adding `strict: false` to the workflow frontmatter, but the recommended path is to keep strict mode on. Workflows that request `issues: write`, `pull-requests: write`, or `contents: write` will fail validation in strict mode. Use read-only permissions plus `safe-outputs` for labels, comments, and PR creation instead.

7.5 Compilation Model Examples

GH-AW compilation is mostly a structural translation: frontmatter becomes the workflow header, the markdown body is packaged as a script or prompt payload, and imports are inlined or referenced. The compiled `.lock.yml` is the contract GitHub Actions executes. The examples below show how a markdown workflow turns into a compiled job.

7.5.1 Example 1: Issue Triage Workflow

Source markdown (`.github/workflows/triage.md`)

```
---
on:
  issues:
    types: [opened]
permissions:
  contents: read
  issues: read
tools:
  github:
    toolsets: [issues]
safe-outputs:
  add-comment:
    max: 1
  add-labels:
    allowed: [needs-triage, needs-owner]
    max: 2
engine: copilot
---

# Triage this issue
Read issue #${{ github.event.issue.number }} and summarize it.
Then suggest labels: needs-triage and needs-owner.
```

Compiled workflow (`.github/workflows/triage.lock.yml`)

```
name: GH-AW triage
on:
  issues:
    types: [opened]
permissions:
  contents: read
  issues: read
jobs:
  agent:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout actions folder
        uses: actions/checkout@8e8c483db84b4bee98b60c0593521ed34d9990e8 # v6
```

```
        with:
          sparse-checkout: |
            actions
          persist-credentials: false
      - name: Setup GH-AW scripts
        uses: ./actions/setup
        with:
          destination: /opt/gh-aw/actions
      - name: Run GH-AW agent (generated)
        uses: actions/github-script@ed597411d8f924073f98dfc5c65a23a2325f34cd # v8.0.0
        with:
          script: |
            const { setupGlobals } = require('/opt/gh-aw/actions/setup_globals.cjs');
            setupGlobals(core, github, context, exec, io);
            // Generated execution script omitted for brevity.
```

What changed during compilation

- Frontmatter was converted into workflow metadata (`on`, `permissions`, `jobs`).
- Generated steps reference the GH-AW scripts copied by the setup action.
- The markdown body became the prompt payload executed by the agent runtime.
- `safe-outputs` declarations were compiled into guarded output steps.

7.5.2 Example 2: Reusable Component + Import

Component (`.github/workflows/shared/common-tools.md`)

```
---
tools:
  bash:
  edit:
engine: copilot
---
```

Workflow using an import (`.github/workflows/docs-refresh.md`)

```
---
on:
  workflow_dispatch:
permissions:
  contents: read
imports:
  - shared/common-tools.md
safe-outputs:
  create-pull-request:
    max: 1
---

# Refresh docs
Update the changelog with the latest release notes.
```

Compiled workflow (`.github/workflows/docs-refresh.lock.yml`)

```
name: GH-AW docs refresh
on:
  workflow_dispatch:
permissions:
  contents: read
jobs:
```

```
  agent:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout actions folder
        uses: actions/checkout@8e8c483db84b4bee98b60c0593521ed34d9990e8 # v6
        with:
          sparse-checkout: |
            actions
          persist-credentials: false
      - name: Setup GH-AW scripts
        uses: ./actions/setup
        with:
          destination: /opt/gh-aw/actions
      - name: Run GH-AW agent (generated)
        uses: actions/github-script@ed597411d8f924073f98dfc5c65a23a2325f34cd # v8.0.0
        with:
          script: |
            const { setupGlobals } = require('/opt/gh-aw/actions/setup_globals.cjs');
            setupGlobals(core, github, context, exec, io);
            // Generated execution script omitted for brevity.
```

What changed during compilation

- `imports` were resolved and merged with the workflow frontmatter.
- The component's `tools` and `engine` were applied to the final workflow.
- Only workflows with `on:` are compiled; components remain markdown-only.
- Read-only permissions pair with `safe-outputs` to stage changes safely.

7.5.3 Example 3: Safe Outputs in the Compiled Job

Source markdown (`.github/workflows/release-notes.md`)

```
---
on:
  workflow_dispatch:
permissions:
  contents: read
tools:
  edit:
safe-outputs:
  create-pull-request:
    max: 1
engine: copilot
---

# Draft release notes
Summarize commits since the last tag and propose a PR with the notes.
```

Compiled workflow (`.github/workflows/release-notes.lock.yml`)

```
name: GH-AW release notes
on:
  workflow_dispatch:
permissions:
  contents: read
jobs:
  agent:
    runs-on: ubuntu-latest
    steps:
```

```
      - name: Checkout actions folder
        uses: actions/checkout@8e8c483db84b4bee98b60c0593521ed34d9990e8 # v6
        with:
          sparse-checkout: |
            actions
          persist-credentials: false
      - name: Setup GH-AW scripts
        uses: ./actions/setup
        with:
          destination: /opt/gh-aw/actions
      - name: Run GH-AW agent (generated)
        uses: actions/github-script@ed597411d8f924073f98dfc5c65a23a2325f34cd # v8.0.0
        with:
          script: |
            const { setupGlobals } = require('/opt/gh-aw/actions/setup_globals.cjs');
            setupGlobals(core, github, context, exec, io);
            // Generated execution script omitted for brevity.
```

What changed during compilation

- `safe-outputs` was translated into the generated safe-output scripts invoked by the job.
- The prompt stayed identical; guardrails are enforced by the compiled job.

7.6 Tools, Safe Inputs, and Safe Outputs

GH-AW workflows are designed for safety by default. Agents run with minimal access and must declare tools explicitly.

> **Warning:** Treat CI secrets and tokens as production credentials. Use least-privilege permissions, require human approval for write actions, and keep all agent actions auditable.

7.6.1 Tools

Tools are capabilities the agent can use. The **edit** tool allows the agent to modify files in the workspace. The **bash** tool runs shell commands, with safe commands enabled by default. The **web-fetch** and **web-search** tools allow the agent to fetch or search web content. The **github** tool operates on issues, pull requests, discussions, and projects. The **playwright** tool provides browser automation for UI checks.

7.6.2 Safe Outputs

Write actions (creating issues, comments, commits) can be routed through safe outputs to sanitize what the agent writes. This keeps the core job read-only and limits accidental changes.

In strict mode, safe outputs are required for write operations. Declare them in frontmatter to specify what the agent can produce:

```
safe-outputs:
  add-comment:
    max: 1
  add-labels:
    allowed: [needs-triage, needs-owner]
    max: 2
  create-pull-request:
    max: 1
```

The agent generates structured output that downstream steps apply, keeping repository writes explicit and auditable.

`max` caps how many outputs of a given type are accepted; extra outputs are rejected by the safe-output validator.

When using `add-labels`, keep the `allowed` list in sync with labels that already exist in the repository; missing labels cause runtime output failures when the safe-output job applies them.

7.6.3 Safe Inputs

You can define safe inputs to structure what the agent receives. This is a good place to validate schema-like data for tools or commands.

7.7 Imports and Reusable Components

For terminology and trust-model definitions, see Discovery and Imports. This section focuses only on GH-AW-specific syntax and composition patterns.

GH-AW supports imports in two ways:

- **Frontmatter imports**

```
imports:
  - shared/common-tools.md
  - shared/research-library.md
```

- **Markdown directive**

```
{% raw %}{{#import shared/common-tools.md}}{% endraw %}
```

In GH-AW, these imports are typically workflow-fragment artefacts: shared prompts, tool declarations, and policy snippets. Keep reusable fragments in files without `on:` so they can be imported as components rather than compiled as standalone workflows.

7.8 ResearchPlanAssign: A Pattern for Self-Maintaining Books

GH-AW documents a **ResearchPlanAssign** strategy: a scaffolded loop that keeps humans in control while delegating research and execution to agents.

Phase 1: Research. A scheduled agent scans the repository or ecosystem for updates such as new libraries, frameworks, or scaffolds. It produces a report in an issue or discussion, summarising findings and flagging items that may warrant attention.

Phase 2: Plan. Maintainers review the report and decide whether to proceed. If approved, a planning agent drafts the implementation steps, breaking the work into discrete tasks that can be assigned and tracked.

Phase 3: Assign and Implement. Agents are assigned to implement the approved changes. Updates are validated through tests and reviews, committed to the repository, and published to the appropriate outputs.

This pattern maps well to this book: use scheduled research to discover new agentic tooling, post a proposal issue, build consensus, then update the chapters and blog.

7.9 Applying GH-AW to This Repository

This repository uses a label-driven lifecycle documented in WORKFLOW_PLAYBOOK.md.

Intake + triage. When an issue is opened, the intake workflow acknowledges it and routes it to either `triaged-fast-track` or `triaged-for-research`.

Fast-track lane. Issues labeled `triaged-fast-track` are implemented directly by the fast-track workflow, which opens a PR, adds `assigned`, and closes the issue.

Research lane. Issues labeled `triaged-for-research` move to `researched-waiting-opinions`, receive both opinion labels (`opinion-copilot-strategy-posted` and `opinion-copilot-delivery-posted`), then the assignment workflow adds `assigned` and closes the issue.

Rejection path. At any stage, an agent can add `rejected` with rationale and close the issue.

Publishing remains a separate automation concern: `pages.yml` deploys the site, and `build-pdf.yml` maintains the generated PDF.

7.10 Key Takeaways

GH-AW turns markdown instructions into reproducible GitHub Actions workflows, combining the readability of documentation with the reliability of automation. Frontmatter defines triggers, permissions, tools, and models, giving you fine-grained control over what the agent can do. Imports enable composable, reusable workflow building blocks that reduce duplication across repositories. Safe inputs and outputs combined with least-privilege permissions reduce the risk of unintended changes. The ResearchPlanAssign pattern provides a practical loop for continuous, agent-powered improvement with human oversight at key decision points.

Chapter 8

GitHub Agents

8.1 Chapter Preview

This chapter describes how agents operate inside GitHub issues, pull requests, and Actions, providing practical context for building agent-powered workflows. It shows safe assignment, review, and approval flows that keep humans in control of consequential changes. Finally, it maps GitHub agent capabilities to real repository workflows, demonstrating patterns you can adapt for your own projects.

8.2 Understanding GitHub Agents

GitHub Agents represent a new paradigm in software development automation. They are AI-powered assistants that can understand context, make decisions, and take actions within the GitHub ecosystem. Unlike traditional automation that follows predefined scripts, agents can adapt to situations, reason about problems, and collaborate with humans and other agents.

This chapter explores the landscape of GitHub Agents, their capabilities, and how to leverage them effectively in your development workflows.

8.3 The GitHub Agent Ecosystem

8.3.1 GitHub Copilot

GitHub Copilot (https://docs.github.com/en/copilot) is the foundation of GitHub's AI-powered development tools. It provides **code completion** with real-time suggestions as you type, predicting the code you're likely to write next. It offers a **chat interface** for natural language conversations about code, allowing you to ask questions and request explanations. And it provides **context awareness**, understanding your codebase and intent so suggestions fit your project's patterns and conventions.

```
# Example: Copilot helping write a function
# Just start typing a comment describing what you need:
# Function to validate email addresses using regex
def validate_email(email):
    import re
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    return bool(re.match(pattern, email))
```

8.3.2 GitHub Copilot Coding Agent

The Coding Agent extends Copilot's capabilities to autonomous task completion. See https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent for the supported assignment and review flow.

The Coding Agent can receive **assigned tasks** from issues or requests and work independently without continuous human guidance. It supports **multi-file changes**, modifying multiple files across a codebase in a single operation. It handles **pull request creation**, generating complete PRs with descriptions that explain what changed and why. And it supports **iterative development**, responding to review feedback and making additional changes based on comments.

Key Characteristics:

Feature	Description
Autonomy	Works independently on assigned tasks
Scope	Can make changes across entire repositories
Output	Creates branches, commits, and pull requests
Review	All changes go through normal PR review

8.3.3 GitHub Actions Agents

Agents can be orchestrated through GitHub Actions workflows:

```
name: Agent Workflow
on:
  issues:
    types: [opened]

jobs:
  agent-task:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - name: Process with Agent
        uses: actions/github-script@v8
        with:
          script: |
            // Agent logic to analyze and respond
            const issue = context.payload.issue;
            // ... agent processing
```

> **Tip:** In production workflows, pin third-party actions to a full commit SHA to reduce supply-chain risk.

> **Warning:** Require human approval before any agent-created PR is merged, and log all agent actions for auditability.

8.4 Agent Capabilities

8.4.1 Reading and Understanding

Agents can read and understand various types of content within a repository. They can process **code**, including source files, configurations, and dependency manifests. They can interpret **documentation** such as READMEs, wikis, and inline comments. They can analyse **issues and pull requests**, including descriptions, comments, and reviews. And they can comprehend **repository structure**, recognising file organisation patterns and project conventions.

8.4.2 Writing and Creating

Agents can produce several types of output. They can make **code changes**, creating new files, modifying existing ones, or refactoring for improved structure. They can write **documentation**, including READMEs, inline comments, and API docs. They can create **issues and comments**, posting status updates and analysis reports. And they can generate **pull requests** with complete descriptions that explain the changes.

8.4.3 Reasoning and Deciding

Agents can perform higher-level cognitive tasks. They can **analyse problems**, understanding issue context and requirements to identify what needs to be done. They can **plan solutions**, breaking down complex tasks into manageable steps. They can **make decisions**, choosing between alternative approaches based on trade-offs. And they can **adapt**, responding to feedback and changing requirements rather than failing when conditions shift.

8.5 Multi-Agent Orchestration

8.5.1 Why Multiple Agents?

Single agents have limitations in capability, perspective, and throughput. Multi-agent systems address these through four key benefits.

Specialisation allows each agent to excel at specific tasks, with dedicated agents for code review, documentation, testing, and other concerns. **Perspective diversity** means different models bring different strengths—one model may be better at security analysis while another excels at explaining concepts clearly. **Scalability** enables parallel processing of independent tasks, reducing total time to completion. **Resilience** ensures that failure of one agent does not stop the workflow; other agents can continue working or pick up where the failed agent left off.

8.5.2 Orchestration Patterns

8.5.2.1 Sequential Pipeline

Agents work in sequence, each building on the previous:

```
Issue -> ACK Agent -> Research Agent -> Writer Agent -> Review Agent -> Complete
```

Example workflow stages:

```
jobs:
  stage-1-acknowledge:
    runs-on: ubuntu-latest
    if: github.event.action == 'opened'
    # Acknowledge and validate

  stage-2-research:
    runs-on: ubuntu-latest
    needs: stage-1-acknowledge
    if: needs.stage-1-acknowledge.outputs.is_relevant == 'true'
    # Research and analyze

  stage-3-write:
    runs-on: ubuntu-latest
    needs: stage-2-research
    # Create content
```

8.5.2.2 Parallel Discussion

Multiple agents contribute perspectives simultaneously:

```
jobs:
  discuss:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        agent: [claude, copilot, gemini]
    steps:
      - name: Agent Perspective
        # Each agent provides its view
```

8.5.2.3 Human-in-the-Loop

Agents work until human decision is needed:

```
Agents work -> Human checkpoint -> Agents continue
```

This pattern is essential for approving significant changes that have broad impact, resolving ambiguous decisions where judgement is required, and ensuring quality assurance before changes reach production.

8.5.3 Agent Handoff Protocol

When agents need to pass context to each other, they follow a structured handoff protocol.

State in labels. Agents use GitHub labels to track workflow stage, allowing both humans and other agents to see at a glance where an issue stands in the process.

Context in comments. Agents document their findings in issue comments, creating a persistent record of what was discovered and decided.

Structured output. Agents use consistent formats for machine readability, enabling downstream agents to parse and act on upstream results programmatically.

```
# Example: Structured agent output
- name: Agent Report
  uses: actions/github-script@v8
  with:
    script: |
      const report = {
        stage: 'research',
        findings: [...],
        recommendation: 'proceed',
        nextAgent: 'writer'
      };
      // Store in comment or labels
```

8.6 Implementing GitHub Agents

8.6.1 Agent Definition Files

Define agents in markdown files with frontmatter:

```
---
name: Research Agent
description: Analyzes issues and researches documentation
```

```
tools:
  github:
    toolsets: [issues]
  web-search:
  edit:
---

# Research Agent

You are the research agent for this repository.
Your role is to analyze suggestions and assess their value.

## Tasks
1. Search existing documentation
2. Find relevant external sources
3. Assess novelty and interest
4. Report findings
```

8.6.2 Agent Configuration

Control agent behavior through configuration:

```
# .github/agents/config.yml
agents:
  research:
    enabled: true
    model: copilot
    timeout: 300

  writer:
    enabled: true
    model: copilot
    requires_approval: true

safety:
  max_file_changes: 10
  protected_paths:
    - .github/workflows/
    - SECURITY.md
```

8.6.3 Error Handling

Agents should handle failures gracefully:

```
- name: Agent Task with Error Handling
  id: agent_task
  continue-on-error: true
  uses: actions/github-script@v8
  with:
    script: |
      try {
        // Agent logic
      } catch (error) {
        await github.rest.issues.addLabels({
          owner: context.repo.owner,
          repo: context.repo.repo,
          issue_number: context.issue.number,
          labels: ['agent-error']
```

```
        });
        await github.rest.issues.createComment({
          owner: context.repo.owner,
          repo: context.repo.repo,
          issue_number: context.issue.number,
          body: `WARNING: Agent encountered an error: ${error.message}`
        });
      }
```

8.7 Best Practices

8.7.1 Clear Agent Personas

Give each agent a clear identity and responsibility:

```
## You Are: The Research Agent

**Your Role:** Investigate and analyze
**You Are Not:** A decision maker or implementer
**Hand Off To:** Writer Agent after research is complete
```

8.7.2 Structured Communication

Use consistent formats for agent-to-agent communication:

```
## Agent Report Format

### Status: [Complete/In Progress/Blocked]
### Findings:
- Finding 1
- Finding 2
### Recommendation: [Proceed/Revise/Decline]
### Next Stage: [stage-name]
```

8.7.3 Human Checkpoints

Always include human review points at critical junctures. Review should happen before significant changes that could affect production systems or user experience. Review should happen after agent recommendations, ensuring a human validates the suggested course of action. And review should happen before closing issues, confirming that the work is complete and meets requirements.

8.7.4 Audit Trail

Maintain visibility into agent actions throughout the workflow. All agent actions should be visible in comments, creating a complete record of what happened. Use labels to track workflow state, making progress visible at a glance. Log important decisions and reasoning so future reviewers can understand why choices were made.

8.7.5 Graceful Degradation

Design for agent failures rather than assuming they will not occur. Use `continue-on-error` for non-critical steps so that failures do not halt the entire workflow. Provide manual fallback options that humans can use when automated approaches fail. Alert maintainers when intervention is needed, ensuring problems are addressed promptly.

8.8 Security Considerations

8.8.1 Least Privilege

Agents should have minimal permissions:

```
permissions:
  contents: read  # Only write if needed
  issues: write   # To comment and label
  pull-requests: write  # Only if creating PRs
```

8.8.2 Input Validation

Validate data before agent processing:

```
// Validate issue body before processing
const body = context.payload.issue.body || '';
if (body.length > 10000) {
  throw new Error('Issue body too long');
}
```

8.8.3 Output Sanitization

Sanitize agent outputs:

```
// Escape user content in agent responses
const safeTitle = issueTitle.replace(/[<>]/g, '');
```

8.8.4 Protected Resources

Prevent agents from modifying sensitive files:

```
# In workflow: check protected paths
- name: Check Protected Paths
  run: |
    CHANGED_FILES=$(git diff --name-only HEAD~1)
    if echo "$CHANGED_FILES" | grep -E "^(SECURITY|\.github/workflows/)"; then
      echo "Protected files modified - requires human review"
      exit 1
    fi
```

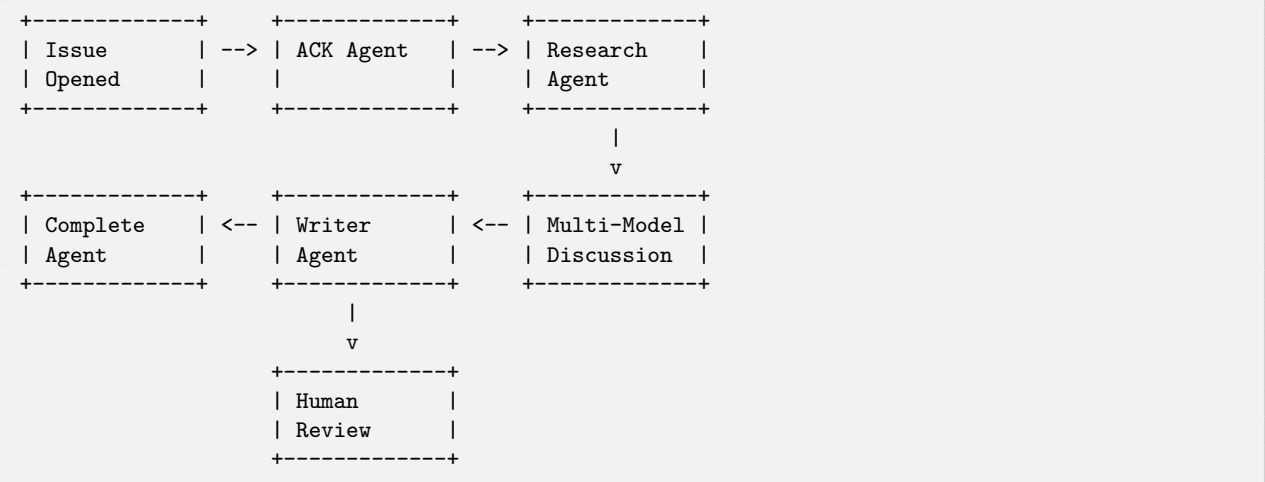
8.9 Real-World Example: This Book

This very book uses GitHub Agents for self-maintenance:

8.9.1 The Multi-Agent Workflow

1. **ACK Agent**: Acknowledges new issue suggestions
2. **Research Agent**: Analyzes novelty and relevance
3. **Claude Agent**: Provides safety and clarity perspective
4. **Copilot Agent**: Provides developer experience perspective
5. **Writer Agent**: Drafts new content
6. **Completion Agent**: Finalizes and closes issues

8.9.2 How It Works



8.9.3 Configuration

The workflow is defined using GitHub Agentic Workflows (GH-AW). The repository includes GH-AW workflows in `.github/workflows/issue-*.lock.yml` and agent definitions in `.github/agents/*.md`.

For a detailed explanation of the workflow architecture and why GH-AW is the canonical approach, see the repository's README workflow section and WORKFLOW_PLAYBOOK.md.

8.10 Multi-Agent Platform Compatibility

Modern repositories need to support multiple AI agent platforms. Different coding assistants—GitHub Copilot (https://docs.github.com/en/copilot), Claude (https://code.claude.com/docs), OpenAI Codex (https://openai.com/index/introducing-codex/), and others—each have their own ways of receiving project-specific instructions. This section explains how to structure a repository for cross-platform agent compatibility.

8.10.1 The Challenge of Agent Diversity

When multiple AI agents work with your repository, you face a coordination challenge. **GitHub Copilot** reads `.github/copilot-instructions.md` for project-specific guidance. **Claude** automatically incorporates `CLAUDE.md`; the generic `AGENTS.md` may still be useful as shared project documentation but should be explicitly referenced for reliable Claude workflows. **OpenAI Codex (GPT-5.3-Codex)** can be configured with system instructions and skills packaged via `SKILL.md` (see https://developers.openai.com/codex). **Generic agents** look for `AGENTS.md` as the emerging standard for project-level instructions.

> **Note:** As of February 2026, both Claude and Codex are available as GitHub engines in public preview, joining Copilot as first-class options for GitHub-integrated agentic workflows.

Each platform has slightly different expectations, but the core information they need is similar: project structure, coding conventions, build commands, and constraints.

8.10.2 Repository Documentation as Agent Configuration

Your repository's documentation files serve dual purposes—they guide human contributors AND configure AI agents. Key files include:

File	Human Purpose	Agent Purpose
`README.md`	Project overview	Context for understanding the codebase
`WORKFLOW_PLAYBOOK.md`	Lifecycle and label matrix	Source of truth for issue workflow routing
`.github/copilot-instructions.md`	N/A	Copilot-specific configuration
`AGENTS.md`	N/A	Generic agent instructions
`CLAUDE.md`	N/A	Claude-specific configuration

8.10.3 The copilot-instructions.md File

GitHub Copilot reads `.github/copilot-instructions.md` to understand how to work with your repository. Keep this file short and operational: project purpose, build/test commands, protected paths, and security constraints. For canonical long-form structure examples, see Skills and Tools Management.

8.10.4 Cross-Platform Strategy

For maximum compatibility across AI agent platforms, follow these practices:

1. **Pick a canonical source per platform** (`AGENTS.md` for many coding agents, `CLAUDE.md` for Claude)
2. **Cross-reference shared guidance** between platform files to reduce drift
3. **Keep instructions DRY** by avoiding unnecessary duplication
4. **Test with multiple agents** to ensure instructions work correctly

Example hierarchy:

```
project/
|-- AGENTS.md                      # Canonical agent instructions
|-- CLAUDE.md                      # Claude-specific (may reference AGENTS.md)
|-- .github/
|   `-- copilot-instructions.md    # Copilot-specific (may reference AGENTS.md)
`-- src/
    `-- AGENTS.md                  # Module-specific instructions
```

8.10.5 This Repository's Approach

This book repository demonstrates multi-platform compatibility through several mechanisms. The `.github/copilot-instructions.md` file provides Copilot configuration with project structure, coding guidelines, and constraints. The **Skills and Tools Management** and **Agents for Coding** chapters discuss AGENTS.md as the emerging standard. The **documentation files** such as README and WORKFLOW_PLAYBOOK provide context any agent can use. The **GH-AW workflows** use the `engine: copilot` setting but the pattern works with other engines.

The key insight is that well-structured documentation benefits both human developers and AI agents. When you write clear README files, contribution guidelines, and coding standards, you are simultaneously creating better agent configuration.

8.10.6 Best Practices for Agent-Friendly Repositories

Several practices make repositories more compatible with AI agents.

Be explicit about constraints. Clearly state what agents should NOT do, preventing them from making changes that would violate project policies.

Document your tech stack. Agents perform better when they understand the tools in use, including languages, frameworks, and build systems.

Describe the project structure. Help agents navigate your codebase efficiently by explaining where different types of code live.

Provide examples. Show preferred patterns through code examples that agents can emulate.

List protected paths. Specify files agents should not modify, such as security-critical configuration or workflow definitions.

Include build and test commands. Enable agents to verify their changes work correctly before submitting them for review.

State coding conventions. Help agents write consistent code that matches your project's style.

8.11 Future of GitHub Agents

8.11.1 Emerging Capabilities

Several capabilities are becoming increasingly mature. **Code generation** now produces production-quality code that can be merged with minimal human editing. **Test authoring** automates test creation and maintenance, keeping test suites current as code evolves. **Documentation sync** keeps docs aligned with code, detecting when documentation drifts from implementation. **Security analysis** provides proactive vulnerability detection, identifying issues before they reach production.

8.11.2 Integration Trends

Integration is deepening across several dimensions. **IDE integration** brings deeper VS Code and editor support, making agents available throughout the development workflow. **CI/CD native** support treats agents as first-class CI/CD citizens rather than add-ons. **Cross-repo** capabilities allow agents to work across multiple repositories, coordinating changes that span projects. **Multi-cloud** support enables agents to coordinate across platforms, working with infrastructure that spans providers.

8.12 Key Takeaways

GitHub Agents are AI-powered assistants that can reason, decide, and act within repositories, going beyond simple autocomplete to autonomous task completion.

Copilot Coding Agent can autonomously complete tasks and create pull requests, working independently on assigned issues while respecting review requirements.

Multi-agent orchestration enables specialised, resilient, and scalable automation by dividing work among agents with different strengths.

Human checkpoints remain essential for quality and safety; agents propose changes but humans make final decisions on consequential modifications.

Clear protocols for agent communication ensure smooth handoffs, using labels, comments, and structured output to pass context between agents.

Security must be designed into agent workflows from the start, with least-privilege permissions, input validation, and protected paths.

Multi-platform compatibility is achieved through well-structured documentation including copilot-instructions.md, AGENTS.md, and related files.

This book demonstrates these concepts through its own multi-agent maintenance workflow, serving as a working example of the patterns described.

8.13 Learn More

8.13.1 Repository Documentation

This book's repository includes comprehensive documentation that demonstrates OSS best practices:

- **README** - Overview and quick start guide
- **Contributing section** - How to contribute using issue-driven workflows
- **Workflows section** - Publishing and validation workflow overview
- **SETUP** - Installation and configuration instructions
- **WORKFLOW_PLAYBOOK** - Agentic workflow maintenance patterns
- **AGENTS** - Contributor notes and required checks
- **CLAUDE** - Repository-specific agent guidance
- **Workflow authoring notes** - GH-AW compilation and lifecycle rules
- **LICENSE** - MIT License

8.13.2 Agent Configuration Files

These files configure how AI agents work with this repository:

- **.github/copilot-instructions.md** - GitHub Copilot-specific configuration including project structure, coding guidelines, and constraints

These documents serve as both useful references and examples of how to structure documentation for projects using agentic workflows.

8.13.3 Related Sections

- **Skills and Tools Management** - Covers AGENTS.md standard and MCP protocol for tool management
- **GitHub Agentic Workflows (GH-AW)** - GH-AW specification and engine configuration
- **Agents for Coding** - Detailed coverage of coding agent platforms

---

Chapter 9

Agents for Coding

9.1 Chapter Preview

This chapter compares coding-agent architectures and team patterns, helping you choose the right approach for your project's complexity and scale. It shows the correct way to configure GitHub Copilot coding agent, with working examples you can adapt. Finally, it provides buildable examples with clear labels distinguishing runnable code from pseudocode.

9.2 Introduction

Coding agents represent the most mature category of AI agents in software development. They have evolved from simple autocomplete tools to autonomous entities capable of planning, writing, testing, debugging, and even scaffolding entire software architectures with minimal human input. This chapter explores the specialized architectures, scaffolding patterns, and best practices for deploying agents in coding workflows.

9.3 The Evolution of Coding Agents

9.3.1 From Autocomplete to Autonomy

The progression of coding agents follows a clear trajectory through four phases.

Code Completion (2020–2022) introduced basic pattern matching and next-token prediction, offering suggestions for the next few tokens based on immediate context.

Context-Aware Assistance (2022–2024) added understanding of project structure and intent, allowing agents to make suggestions that fit the broader codebase.

Task-Oriented Agents (2024–present) can complete multi-step tasks independently, taking a high-level instruction and executing a series of operations to fulfil it.

Autonomous Development (emerging) represents the frontier, with agents capable of full feature implementation, testing, and deployment with minimal human intervention.

9.3.2 Current Capabilities

Modern coding agents can perform a range of sophisticated tasks.

Understand requirements. Agents can parse natural language specifications and translate them to code, bridging the gap between human intent and machine-executable instructions.

Plan solutions. Agents can break down complex features into implementable steps, creating a roadmap for development.

Generate code. Agents can write production-quality code across multiple files, handling everything from utility functions to full modules.

Test and debug. Agents can create tests, identify bugs, and fix issues, shortening the feedback loop between writing code and validating it.

Scaffold projects. Agents can initialise projects with appropriate structure and configuration, setting up the foundation for further development.

Review and refactor. Agents can analyse code quality and suggest improvements, helping maintain code health over time.

9.4 Specialized Architectures

9.4.1 Single-Agent Architectures

The simplest architecture involves one agent with access to all necessary tools.

Example 7-2. Single-agent architecture (pseudocode)

```
class CodingAgent:
    """Single-agent architecture for coding tasks"""

    def __init__(self, llm, tools):
        self.llm = llm
        self.tools = {
            'file_read': FileReadTool(),
            'file_write': FileWriteTool(),
            'terminal': TerminalTool(),
            'search': CodeSearchTool(),
            'test_runner': TestRunnerTool()
        }
        self.context = AgentContext()

    async def execute(self, task: str) -> dict:
        """Execute a coding task end-to-end"""
        # 1. Understand the task
        plan = await self.plan_task(task)

        # 2. Execute each step
        results = []
        for step in plan.steps:
            result = await self.execute_step(step)
            results.append(result)

            # Adapt based on results
            if not result.success:
                plan = await self.replan(plan, result)

        return {'success': True, 'results': results}
```

Best for: Simple tasks, small codebases, single-developer workflows.

9.4.2 Multi-Agent Architectures

Complex projects benefit from specialized agents working together.

Example 7-3. Multi-agent architecture (pseudocode)

```
class CodingAgentTeam:
    """Multi-agent architecture mirroring a development team"""

    def __init__(self):
        self.architect = ArchitectAgent()
        self.implementer = ImplementerAgent()
        self.tester = TesterAgent()
        self.reviewer = ReviewerAgent()
        self.coordinator = CoordinatorAgent()

    async def execute_feature(self, specification: str):
        """Execute a feature request using the agent team"""

        # 1. Architecture phase
        design = await self.architect.design(specification)

        # 2. Implementation phase (can be parallelized)
        implementations = await asyncio.gather(*[
            self.implementer.implement(component)
            for component in design.components
        ])

        # 3. Testing phase
        test_results = await self.tester.test(implementations)

        # 4. Review phase
        review = await self.reviewer.review(implementations)

        # 5. Iteration if needed
        if not review.approved:
            return await self.handle_review_feedback(review)

        return {'success': True, 'implementation': implementations}
```

Best for: Large projects, team environments, complex features.

9.4.3 Subagent and Swarms Mode

Modern frameworks like Claude Code support dynamic subagent spawning:

```
class SwarmCoordinator:
    """Coordinate a swarm of specialized subagents"""

    def __init__(self, max_agents=10):
        self.max_agents = max_agents
        self.active_agents = {}

    async def spawn_subagent(self, task_type: str, context: dict):
        """Spawn a specialized subagent for a specific task"""

        agent_configs = {
            'frontend': FrontendAgentConfig(),
            'backend': BackendAgentConfig(),
            'devops': DevOpsAgentConfig(),
            'security': SecurityAgentConfig(),
            'documentation': DocsAgentConfig()
        }
```

```
        config = agent_configs.get(task_type)
        agent = await self.create_agent(config, context)

        self.active_agents[agent.id] = agent
        return agent

    async def execute_parallel(self, tasks: list):
        """Execute multiple tasks in parallel using subagents"""

        agents = [
            await self.spawn_subagent(task.type, task.context)
            for task in tasks
        ]

        results = await asyncio.gather(*[
            agent.execute(task)
            for agent, task in zip(agents, tasks)
        ])

        return self.aggregate_results(results)
```

9.5 Scaffolding for Coding Agents

9.5.1 Project Initialization

Coding agents need scaffolding that helps them understand and work with projects:

```
# .github/agents/coding-agent.yml
name: coding-agent
description: Scaffolding for coding agent operations

workspace:
  root: ./
  source_dirs: [src/, lib/]
  test_dirs: [tests/, spec/]
  config_files: [package.json, tsconfig.json, .eslintrc]

conventions:
  language: typescript
  framework: express
  testing: jest
  style: prettier + eslint

tools:
  enabled:
    - file_operations
    - terminal
    - git
    - package_manager
  restricted:
    - network_access
    - system_commands

safety:
  max_file_changes: 20
  protected_paths:
    - .github/workflows/
```

```
    - .env*
    - secrets/
  require_tests: true
  require_review: true
```

9.5.2 The AGENTS.md Standard

For canonical AGENTS.md structure and rationale, see Skills and Tools Management. For import/install/activate terminology and trust boundaries, see Discovery and Imports. In this chapter we focus on coding-agent execution patterns and platform behavior.

9.5.3 Context Management

Coding agents need effective context management to work across large codebases:

```
class CodingContext:
    """Manage context for coding agents"""

    def __init__(self, workspace_root: str):
        self.workspace_root = workspace_root
        self.file_index = FileIndex(workspace_root)
        self.symbol_table = SymbolTable()
        self.active_files = LRUCache(max_size=50)

    def get_relevant_context(self, task: str) -> dict:
        """Get context relevant to the current task"""

        # 1. Parse task to identify relevant files/symbols
        entities = self.extract_entities(task)

        # 2. Retrieve relevant files
        files = self.file_index.search(entities)

        # 3. Get symbol definitions
        symbols = self.symbol_table.lookup(entities)

        # 4. Include recent changes
        recent = self.get_recent_changes()

        return {
            'files': files,
            'symbols': symbols,
            'recent_changes': recent,
            'workspace_config': self.get_config()
        }

    def update_context(self, changes: list):
        """Update context after agent makes changes"""
        for change in changes:
            self.file_index.update(change.path)
            self.symbol_table.reindex(change.path)
            self.active_files.add(change.path)
```

9.5.4 Tool Registries

Coding agents need well-organized tool access:

```
class CodingToolRegistry:
    """Registry of tools available to coding agents"""

    def __init__(self):
        self._tools = {}
        self._register_default_tools()

    def _register_default_tools(self):
        """Register standard coding tools"""

        # File operations
        self.register('read_file', ReadFileTool())
        self.register('write_file', WriteFileTool())
        self.register('search_files', SearchFilesTool())

        # Code operations
        self.register('parse_ast', ParseASTTool())
        self.register('refactor', RefactorTool())
        self.register('format_code', FormatCodeTool())

        # Testing
        self.register('run_tests', RunTestsTool())
        self.register('coverage', CoverageTool())

        # Git operations
        self.register('git_status', GitStatusTool())
        self.register('git_diff', GitDiffTool())
        self.register('git_commit', GitCommitTool())

        # Package management
        self.register('npm_install', NpmInstallTool())
        self.register('pip_install', PipInstallTool())

    def get_tools_for_task(self, task_type: str) -> list:
        """Get tools appropriate for a task type"""

        task_tool_map = {
            'implementation': ['read_file', 'write_file', 'search_files', 'format_code'],
            'testing': ['read_file', 'write_file', 'run_tests', 'coverage'],
            'debugging': ['read_file', 'parse_ast', 'run_tests', 'git_diff'],
            'refactoring': ['read_file', 'write_file', 'parse_ast', 'refactor', 'run_tests']
        }

        tool_names = task_tool_map.get(task_type, list(self._tools.keys()))
        return [self._tools[name] for name in tool_names if name in self._tools]
```

9.6 Leading Coding Agent Platforms

9.6.1 GitHub Copilot and Coding Agent

GitHub Copilot has evolved from an IDE autocomplete tool to a full coding agent. **Copilot Chat** provides natural language interaction about code, allowing developers to ask questions and request explanations. **Copilot Coding Agent** handles autonomous task completion and PR creation, working independently on assigned tasks. **Copilot Workspace** offers a full development environment with agent integration, bringing together editing, testing, and deployment.

Copilot coding agent is not invoked via a custom `uses:` action. Instead, you assign work through GitHub

Issues, Pull Requests, the agents panel, or by mentioning `@copilot`, and you customise its environment with a dedicated workflow file. See the official docs at https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent and the environment setup guide at https://docs.github.com/en/copilot/how-tos/use-copilot-agents/coding-agent/customize-the-agent-environment.

Example 7-1. `.github/workflows/copilot-setup-steps.yml`

```
name: Copilot setup steps

on:
  push:
    paths:
      - .github/workflows/copilot-setup-steps.yml

jobs:
  # The job MUST be named copilot-setup-steps to be picked up by Copilot.
  copilot-setup-steps:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - name: Install dependencies
        run: |
          npm ci
```

9.6.2 Claude Code

Claude Code (https://code.claude.com/docs) provides multi-agent orchestration for complex development tasks. As of February 2026, Claude is available as a GitHub engine in public preview alongside Copilot and Codex. Its **subagent architecture** allows you to spawn specialised agents for different concerns, each focused on a particular aspect of the problem. **Swarms mode** enables parallel execution of independent tasks, reducing total time to completion. **Extended context** handles large codebases through intelligent context management, summarising and prioritising information to fit within token limits. Claude Code can be used from the browser at `claude.ai/code` or from the terminal as a CLI tool.

9.6.3 Cursor AI

Cursor (https://www.cursor.com/) is an AI-first code editor designed around agent workflows. It provides **project-wide understanding** by indexing the entire codebase for context, ensuring suggestions fit the project's patterns. **Multi-file generation** creates and modifies multiple files in one operation, handling cross-cutting changes that span components. **Framework integration** gives the editor deep understanding of popular frameworks, improving suggestion quality for framework-specific code.

9.6.4 OpenAI Codex CLI

OpenAI Codex (https://developers.openai.com/codex) has evolved from an API-only model into a full coding agent platform available as a CLI, IDE extension, web interface, and macOS app. The latest model, GPT-5.3-Codex (released February 2026), is described as the most capable agentic coding model to date. Codex supports skills packaged with `SKILL.md` for progressive disclosure, and its CLI enables agentic workflows directly from the terminal. The macOS app serves as a command centre for managing multiple coding agents in parallel.

9.6.5 Aider

Aider (https://aider.chat/) is an open-source, Git-first CLI coding agent for AI pair programming in the terminal. It works best with Claude 3.7 Sonnet, DeepSeek R1, and GPT-4o, but can connect to almost any LLM including local models. Aider makes coordinated changes across multiple files with automatic Git

commits, builds a map of entire repositories for effective refactoring, and integrates automatic linting and testing. It represents the shift from suggestion-based assistance to truly agentic terminal workflows.

9.6.6 Devin

Devin (https://devin.ai/) by Cognition is an autonomous coding agent designed to handle tasks equivalent to four to eight hours of junior engineer work. Cognition's valuation reached $10.2 billion following a $400M Series C in late 2025. In July 2025, Cognition acquired Windsurf, merging IDE and agent approaches. Devin excels at tasks with clear requirements and verifiable outcomes: migrations, vulnerability fixes, unit test generation, and small tickets. It is infinitely parallelisable and works asynchronously.

9.6.7 Windsurf

Windsurf (https://windsurf.com/), formerly Codeium, is an AI-native IDE now owned by Cognition (acquired July 2025). It is a feature-rich fork of VS Code with seamless import of existing settings and extensions. Its **Cascade** feature is an agentic assistant that plans multi-step edits, calls tools, and uses deep repository context. Windsurf offers a permanently free individual plan with unlimited autocomplete and chat, making it accessible for individual developers.

9.6.8 CodeGPT and Agent Marketplaces

CodeGPT (https://codegpt.co/) and marketplace-based approaches offer specialised agents. **Specialised agents** provide over 200 pre-built agents for specific tasks, from code review to documentation generation. **Custom agent creation** lets you build and share domain-specific agents tailored to your organisation's needs. **Multi-model support** combines different LLMs for different tasks, using each model's strengths where they apply best.

9.7 Best Practices

9.7.1 Clear Task Boundaries

Define clear boundaries for what agents can and cannot do:

```
class TaskBoundary:
    """Define boundaries for agent tasks"""

    def __init__(self):
        self.max_files = 20
        self.max_lines_per_file = 500
        self.timeout_seconds = 600
        self.protected_patterns = [
            r'\.env.*',
            r'secrets/.*',
            r'\.github/workflows/.*'
        ]

    def validate_task(self, task: dict) -> bool:
        """Validate that a task is within boundaries"""
        if len(task.get('files', [])) > self.max_files:
            return False

        for file_path in task.get('files', []):
            if any(re.match(p, file_path) for p in self.protected_patterns):
                return False

        return True
```

9.7.2 Incremental Changes

Prefer small, focused changes over large rewrites:

```
class IncrementalChangeStrategy:
    """Strategy for making incremental changes"""

    def execute(self, large_change: Change) -> list:
        """Break large change into incremental steps"""

        # 1. Analyze the change
        components = self.decompose(large_change)

        # 2. Order by dependency
        ordered = self.topological_sort(components)

        # 3. Execute incrementally with validation
        results = []
        for component in ordered:
            result = self.apply_change(component)

            # Validate after each step
            if not self.validate(result):
                self.rollback(results)
                raise ChangeValidationError(result)

            results.append(result)

        return results
```

9.7.3 Test-Driven Development

Integrate testing into agent workflows:

```
class TDDAgent:
    """Agent that follows test-driven development"""

    async def implement_feature(self, specification: str):
        """Implement feature using TDD approach"""

        # 1. Write tests first
        tests = await self.generate_tests(specification)
        await self.write_tests(tests)

        # 2. Verify tests fail
        initial_results = await self.run_tests()
        assert not initial_results.all_passed

        # 3. Implement to pass tests
        implementation = await self.implement(specification, tests)

        # 4. Verify tests pass
        final_results = await self.run_tests()

        # 5. Refactor if needed
        if final_results.all_passed:
            await self.refactor_for_quality()

        return implementation
```

9.7.4 Human Review Integration

Always include human checkpoints for significant changes:

Example 7-4. Human review workflow (pseudocode)

```
# Workflow with human review
name: Agent Implementation with Review
on:
  issues:
    types: [labeled]

jobs:
  implement:
    if: contains(github.event.issue.labels.*.name, 'agent-task')
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6 # pin to a full SHA in production

      - name: Agent Implementation
        id: implement
        uses: ./actions/coding-agent

      - name: Create PR for Review
        uses: peter-evans/create-pull-request@v8 # pin to a full SHA in production
        with:
          title: "Agent: ${{ github.event.issue.title }}"
          body: |
            ## Agent Implementation

            This PR was created by an AI agent based on issue #${{ github.event.issue.number }}.

            **Please review carefully before merging.**
          labels: needs-human-review
          draft: true
```

> **Note:** The `./actions/coding-agent` step is a placeholder for your organization's internal agent runner. Replace it with your approved agent execution mechanism.

9.8 Common Challenges

9.8.1 Context Window Limitations

Large codebases exceed agent context windows:

Solution: Implement intelligent context retrieval and summarization.

```
class ContextCompressor:
    """Compress context to fit within token limits"""

    def compress(self, files: list, max_tokens: int) -> str:
        """Compress file contents to fit token limit"""

        # Prioritize by relevance
        ranked = self.rank_by_relevance(files)

        # Include summaries for less relevant files
        context = []
        tokens_used = 0
```

```
        for file in ranked:
            if tokens_used + file.tokens <= max_tokens:
                context.append(file.content)
                tokens_used += file.tokens
            else:
                # Include summary instead
                summary = self.summarize(file)
                context.append(summary)
                tokens_used += len(summary.split())

        return '\n'.join(context)
```

9.8.2 Hallucination and Accuracy

Agents may generate plausible but incorrect code:

Solution: Implement validation and testing at every step.

9.8.3 Security Concerns

Agents with code access pose security risks:

Solution: Use sandboxing, permission scoping, and audit logging.

```
class SecureCodingEnvironment:
    """Secure environment for coding agent execution"""

    def __init__(self):
        self.sandbox = DockerSandbox()
        self.audit_log = AuditLog()

    async def execute(self, agent, task):
        """Execute agent in secure sandbox"""

        # Log the task
        self.audit_log.log_task_start(agent.id, task)

        try:
            # Run in sandbox
            result = await self.sandbox.run(agent, task)

            # Validate output
            self.validate_output(result)

            # Log completion
            self.audit_log.log_task_complete(agent.id, result)

            return result

        except Exception as e:
            self.audit_log.log_task_error(agent.id, e)
            raise
```

9.9 Key Takeaways

Coding agents have evolved from autocomplete to autonomous development assistants, progressing through phases of increasing capability and independence. The landscape now includes IDE-integrated assistants (Copilot, Cursor, Windsurf), CLI-based agents (Claude Code, Codex CLI, Aider), and fully autonomous agents (Devin).

Multi-agent architectures mirror development teams, with specialised agents for architecture, implementation, testing, and review, each contributing expertise to the overall workflow.

AGENTS.md is the emerging standard for providing agents with project-specific instructions, serving as a "README for agents" that helps them understand how to work within a codebase.

Scaffolding for coding agents includes context management to handle large codebases, tool registries to organise capabilities, and security boundaries to limit what agents can access.

Human review remains essential—agents create PRs for review, not direct commits. This ensures humans maintain oversight over changes that affect production systems.

Incremental changes with continuous validation are safer than large rewrites. Small, focused modifications are easier to review and less likely to introduce subtle bugs.

Security must be designed in from the start: sandboxing isolates agent execution, permissions scope what agents can access, and audit logging tracks what they do.

Chapter 10

Agents for Mathematics and Physics

10.1 Chapter Preview

This chapter identifies where formal methods and computer algebra system (CAS) tools fit in agent workflows, explaining how they complement neural approaches. It separates runnable tooling from conceptual pseudocode, providing clear labels so readers know what can be executed directly. Finally, it highlights verification pitfalls and how to avoid them, addressing the unique challenges of mathematical and scientific correctness.

> **Note:** Many examples in this chapter are illustrative pseudocode unless explicitly labeled as runnable, because formal tooling and CAS systems require environment-specific setup.

10.2 Introduction

Mathematics and physics present unique challenges for AI agents. Unlike coding, where correctness can often be verified through tests, mathematical reasoning requires formal proof and physical models demand empirical validation. This chapter explores specialized agents for scientific domains, their architectures, and the scaffolding required to support rigorous reasoning.

10.3 The Landscape of Scientific Agents

10.3.1 Distinct Requirements

Scientific agents differ from coding agents in several key ways:

Aspect	Coding Agents	Scientific Agents
Verification	Tests, linting	Formal proofs, experimental validation
Precision	Functional correctness	Mathematical rigor
Output	Source code	Theorems, proofs, equations
Tools	IDEs, compilers	Proof assistants, CAS, simulators
Context	Codebase	Theorems, papers, datasets

10.3.2 Categories of Scientific Agents

Scientific agents fall into five main categories.

Theorem Proving Agents construct formal proofs in systems like Lean (https://lean-lang.org/learn/), Coq (https://coq.inria.fr/), or Isabelle (https://isabelle.in.tum.de/), producing machine-verifiable derivations.

Symbolic Computation Agents work with computer algebra systems (CAS), manipulating mathematical expressions symbolically rather than numerically.

Numerical Simulation Agents set up and run physics simulations, handling the computational infrastructure for modelling physical systems.

Research Assistants search literature, summarise findings, and identify gaps, helping researchers navigate the vast body of published work.

Educational Scaffolding Agents help students learn mathematical and physical concepts, adapting explanations to the learner's level and addressing misconceptions.

10.4 Theorem Proving Agents

10.4.1 Formal Verification Background

Formal theorem proving ensures mathematical correctness through rigorous logical derivation. Unlike informal proofs in papers, formal proofs are machine-verifiable.

10.4.2 Ax-Prover Architecture

> **Note:** Ax-Prover is a hypothetical composite example used to illustrate multi-agent theorem proving patterns.

```
class AxProverAgent:
    """Multi-agent theorem proving architecture inspired by Ax-Prover"""

    def __init__(self, llm, proof_assistant):
        self.llm = llm
        self.proof_assistant = proof_assistant  # e.g., Lean, Coq
        self.strategy_agents = {
            'decomposition': DecompositionAgent(llm),
            'lemma_search': LemmaSearchAgent(llm),
            'tactic_selection': TacticSelectionAgent(llm),
            'creativity': CreativityAgent(llm)
        }

    async def prove(self, theorem: str) -> ProofResult:
        """Attempt to prove a theorem"""

        # 1. Formalize the statement
        formal_statement = await self.formalize(theorem)

        # 2. Decompose into subgoals
        subgoals = await self.strategy_agents['decomposition'].decompose(
            formal_statement
        )

        # 3. Search for relevant lemmas
        lemmas = await self.strategy_agents['lemma_search'].search(
            formal_statement, subgoals
        )

        # 4. Generate proof attempts
        proof_attempts = await self.generate_proof_attempts(
```

```
            formal_statement, subgoals, lemmas
        )

        # 5. Verify with proof assistant
        for attempt in proof_attempts:
            result = await self.proof_assistant.check(attempt)
            if result.verified:
                return ProofResult(success=True, proof=attempt)

        return ProofResult(success=False, partial_proofs=proof_attempts)

    async def formalize(self, natural_language: str) -> str:
        """Convert natural language to formal notation"""
        prompt = f"""
        Convert the following mathematical statement to formal Lean 4 syntax:

        Statement: {natural_language}

        Provide the formal statement only.
        """
        return await self.llm.generate(prompt)
```

10.4.3 Integration with Proof Assistants

Agents connect to proof assistants through well-defined interfaces:

```
class LeanProofAssistant:
    """Interface to Lean 4 proof assistant"""

    def __init__(self, project_path: str):
        self.project_path = project_path
        self.server = LeanServer(project_path)

    async def check(self, proof: str) -> VerificationResult:
        """Verify a proof in Lean"""

        # Write proof to file
        proof_file = self.write_proof(proof)

        # Run Lean verification
        result = await self.server.check_file(proof_file)

        return VerificationResult(
            verified=not result.has_errors,
            errors=result.errors,
            goals=result.remaining_goals
        )

    async def get_available_tactics(self, goal_state: str) -> list:
        """Get tactics applicable to current goal state"""
        return await self.server.suggest_tactics(goal_state)

    async def search_mathlib(self, query: str) -> list:
        """Search Mathlib for relevant lemmas"""
        return await self.server.library_search(query)
```

10.4.4 Challenges in Theorem Proving

Theorem proving presents four main challenges.

Search space explosion means proofs can have many possible paths, and exploring all of them quickly becomes computationally infeasible.

Creativity required reflects that non-obvious proof strategies often lead to success, requiring agents to generate novel approaches rather than following templates.

Formalisation gap is the challenge of translating informal mathematical statements into the precise syntax required by proof assistants.

Domain knowledge recognises that deep mathematical understanding is needed to guide proof search effectively and choose appropriate lemmas.

10.5 Symbolic Computation Agents

10.5.1 Computer Algebra Systems

Symbolic computation agents work with systems like Mathematica, SymPy, or SageMath:

```
class SymbolicComputationAgent:
    """Agent for symbolic mathematical computation"""

    def __init__(self, llm, cas_backend='sympy'):
        self.llm = llm
        self.cas = self.initialize_cas(cas_backend)

    async def solve(self, problem: str) -> Solution:
        """Solve a mathematical problem symbolically"""

        # 1. Parse the problem
        parsed = await self.parse_problem(problem)

        # 2. Identify the type of problem
        problem_type = await self.classify_problem(parsed)

        # 3. Select appropriate methods
        methods = self.get_methods(problem_type)

        # 4. Attempt solutions
        for method in methods:
            try:
                result = await self.apply_method(method, parsed)
                if result.is_valid:
                    return Solution(
                        answer=result.answer,
                        method=method,
                        steps=result.steps
                    )
            except ComputationError:
                continue

        return Solution(success=False, attempted_methods=methods)

    async def simplify(self, expression: str) -> str:
        """Simplify a mathematical expression"""

        # Convert to CAS format
```

```
        cas_expr = self.cas.parse(expression)

        # Apply simplification
        simplified = self.cas.simplify(cas_expr)

        # Convert back to readable format
        return self.cas.to_latex(simplified)

    async def compute_integral(self, integrand: str, variable: str,
                               bounds: tuple = None) -> str:
        """Compute definite or indefinite integral"""

        expr = self.cas.parse(integrand)
        var = self.cas.symbol(variable)

        if bounds:
            result = self.cas.integrate(expr, (var, bounds[0], bounds[1]))
        else:
            result = self.cas.integrate(expr, var)

        return self.cas.to_latex(result)
```

10.5.2 Combining Symbolic and Neural Approaches

Modern agents combine symbolic precision with neural flexibility:

```
class HybridMathAgent:
    """Combine symbolic computation with LLM reasoning"""

    def __init__(self, llm, cas):
        self.llm = llm
        self.cas = cas

    async def solve_with_explanation(self, problem: str) -> dict:
        """Solve and explain a mathematical problem"""

        # 1. LLM plans the solution strategy
        strategy = await self.llm.generate(f"""
        Given this problem: {problem}

        Outline a step-by-step solution strategy.
        Identify which steps require symbolic computation.
        """)

        # 2. Parse strategy into executable steps
        steps = self.parse_strategy(strategy)

        # 3. Execute each step
        results = []
        for step in steps:
            if step.requires_symbolic:
                result = await self.cas_execute(step)
            else:
                result = await self.llm_execute(step)
            results.append(result)

        # 4. Compile final answer with explanation
        return {
```

```
            'answer': results[-1],
            'steps': results,
            'explanation': await self.generate_explanation(results)
        }
```

10.6 Physics Simulation Agents

10.6.1 Computational Physics Workflows

Physics agents orchestrate simulation workflows:

```
class PhysicsSimulationAgent:
    """Agent for physics simulations"""

    def __init__(self, llm, simulators):
        self.llm = llm
        self.simulators = {
            'molecular_dynamics': MDSimulator(),
            'quantum': QMSimulator(),
            'classical': ClassicalSimulator(),
            'fluid': CFDSimulator()
        }

    async def run_simulation(self, description: str) -> SimulationResult:
        """Set up and run a physics simulation from natural language"""

        # 1. Understand the physical system
        system_spec = await self.understand_system(description)

        # 2. Select appropriate simulator
        simulator = self.select_simulator(system_spec)

        # 3. Generate simulation parameters
        params = await self.generate_parameters(system_spec)

        # 4. Validate physical consistency
        await self.validate_physics(params)

        # 5. Run simulation
        result = await simulator.run(params)

        # 6. Analyze results
        analysis = await self.analyze_results(result, system_spec)

        return SimulationResult(
            raw_data=result,
            analysis=analysis,
            visualizations=await self.generate_plots(result)
        )

    async def validate_physics(self, params: dict):
        """Ensure simulation parameters are physically consistent"""

        # Check conservation laws
        if not self.check_energy_conservation(params):
            raise PhysicsError("Energy conservation violated")

        # Check dimensional consistency
```

```
        if not self.check_dimensions(params):
            raise PhysicsError("Dimensional inconsistency")

        # Check boundary conditions
        if not self.check_boundaries(params):
            raise PhysicsError("Invalid boundary conditions")
```

10.6.2 Quantum Physics Specialization

Quantum physics requires specialized handling:

```
class QuantumPhysicsAgent:
    """Specialized agent for quantum mechanical problems"""

    def __init__(self, llm, qm_tools):
        self.llm = llm
        self.tools = qm_tools

    async def solve_schrodinger(self, system: str) -> dict:
        """Solve Schrödinger equation for a system"""

        # 1. Construct Hamiltonian
        hamiltonian = await self.construct_hamiltonian(system)

        # 2. Identify symmetries
        symmetries = await self.find_symmetries(hamiltonian)

        # 3. Choose solution method
        method = self.select_method(hamiltonian, symmetries)

        # 4. Solve
        if method == 'analytical':
            solution = await self.analytical_solve(hamiltonian)
        elif method == 'numerical':
            solution = await self.numerical_solve(hamiltonian)
        elif method == 'variational':
            solution = await self.variational_solve(hamiltonian)

        return {
            'eigenstates': solution.states,
            'eigenvalues': solution.energies,
            'method': method,
            'symmetries': symmetries
        }

    async def compute_observable(self, state, observable: str) -> complex:
        """Compute expectation value of an observable"""

        operator = await self.construct_operator(observable)
        return await self.tools.expectation_value(state, operator)
```

10.7 Scaffolding for Scientific Agents

10.7.1 Tool Integration Layer

Scientific agents need access to specialized tools:

```
# Scientific agent tool configuration
tools:
  proof_assistants:
    lean4:
      path: /usr/local/bin/lean
      mathlib_path: ~/.elan/toolchains/leanprover--lean4---v4.3.0/lib/lean4/library
    coq:
      path: /usr/bin/coqc

  computer_algebra:
    sympy:
      module: sympy
    mathematica:
      path: /usr/local/bin/WolframScript

  simulators:
    molecular_dynamics:
      backend: lammps
      path: /usr/bin/lmp
    quantum:
      backend: qiskit

  visualization:
    matplotlib: true
    plotly: true
    manim: true
```

10.7.2 Knowledge Base Integration

Scientific agents need access to mathematical knowledge:

```
class MathematicalKnowledgeBase:
    """Knowledge base for mathematical agents"""

    def __init__(self):
        self.theorem_database = TheoremDatabase()
        self.formula_index = FormulaIndex()
        self.paper_embeddings = PaperEmbeddings()

    async def search_theorems(self, query: str) -> list:
        """Search for relevant theorems"""

        # Semantic search over theorem statements
        results = await self.theorem_database.semantic_search(query)

        # Include related lemmas and corollaries
        expanded = []
        for theorem in results:
            expanded.append(theorem)
            expanded.extend(await self.get_related(theorem))

        return expanded

    async def get_formula(self, name: str) -> Formula:
        """Retrieve a named formula"""
        return await self.formula_index.get(name)

    async def search_literature(self, topic: str) -> list:
```

```
        """Search mathematical literature"""

        # Search arXiv, Mathlib docs, textbooks
        papers = await self.paper_embeddings.search(topic)
        return papers
```

10.7.3 Verification Pipeline

All scientific agent outputs should be verified:

```
class ScientificVerificationPipeline:
    """Verify correctness of scientific agent outputs"""

    def __init__(self):
        self.proof_checker = ProofChecker()
        self.dimensional_analyzer = DimensionalAnalyzer()
        self.numerical_validator = NumericalValidator()

    async def verify(self, output: ScientificOutput) -> VerificationResult:
        """Verify scientific output for correctness"""

        checks = []

        # 1. Check formal proofs
        if output.has_proofs:
            proof_check = await self.proof_checker.verify(output.proofs)
            checks.append(('proofs', proof_check))

        # 2. Check dimensional consistency
        if output.has_equations:
            dim_check = await self.dimensional_analyzer.check(output.equations)
            checks.append(('dimensions', dim_check))

        # 3. Numerical validation
        if output.has_computations:
            num_check = await self.numerical_validator.validate(
                output.computations
            )
            checks.append(('numerical', num_check))

        # 4. Cross-check with known results
        known_check = await self.check_against_known(output)
        checks.append(('known_results', known_check))

        return VerificationResult(
            verified=all(c[1].passed for c in checks),
            checks=checks
        )
```

10.8 Educational Scaffolding Agents

10.8.1 Mathematics Education

AI agents are transforming mathematics education:

```
class MathTutoringAgent:
    """Agent for mathematics education and tutoring"""
```

```
    def __init__(self, llm, level='undergraduate'):
        self.llm = llm
        self.level = level
        self.student_model = StudentModel()

    async def explain_concept(self, concept: str) -> str:
        """Explain a mathematical concept at appropriate level"""

        # Get student's current understanding
        background = await self.student_model.get_background()

        # Generate explanation
        explanation = await self.llm.generate(f"""
        Explain {concept} to a student with this background: {background}

        Level: {self.level}

        Include:
        - Intuitive explanation
        - Formal definition
        - Key examples
        - Common misconceptions
        - Connection to prior knowledge
        """)

        return explanation

    async def generate_problems(self, topic: str, count: int,
                                difficulty: str) -> list:
        """Generate practice problems with solutions"""

        problems = await self.llm.generate(f"""
        Generate {count} {difficulty} problems on {topic}.

        For each problem provide:
        1. Problem statement
        2. Hints (progressive)
        3. Complete solution
        4. Common errors to avoid
        """)

        return self.parse_problems(problems)

    async def provide_feedback(self, student_work: str,
                               problem: str) -> Feedback:
        """Analyze student work and provide feedback"""

        analysis = await self.llm.generate(f"""
        Analyze this student's solution:

        Problem: {problem}
        Student work: {student_work}

        Provide:
        1. Is the final answer correct?
        2. Are the intermediate steps correct?
        3. What misconceptions are evident?
        4. Specific suggestions for improvement
```

```
        5. Encouragement and next steps
        """)

        return self.parse_feedback(analysis)
```

10.8.2 Physics Education

Physics scaffolding addresses visualization challenges:

```
class PhysicsEducationAgent:
    """Agent for physics education with visualization"""

    def __init__(self, llm, visualizer):
        self.llm = llm
        self.visualizer = visualizer

    async def explain_with_simulation(self, concept: str) -> dict:
        """Explain physics concept with interactive simulation"""

        # Generate explanation
        explanation = await self.explain_concept(concept)

        # Create visualization parameters
        viz_params = await self.generate_visualization_params(concept)

        # Generate simulation
        simulation = await self.visualizer.create_simulation(viz_params)

        # Create interactive exploration tasks
        tasks = await self.generate_exploration_tasks(concept)

        return {
            'explanation': explanation,
            'simulation': simulation,
            'exploration_tasks': tasks,
            'key_parameters': viz_params['adjustable']
        }

    async def analyze_misconception(self, student_statement: str) -> dict:
        """Identify and address physics misconceptions"""

        analysis = await self.llm.generate(f"""
        The student said: "{student_statement}"

        1. Identify any physics misconceptions
        2. Explain the correct physics
        3. Suggest experiments or simulations to demonstrate
        4. Provide an analogy that builds correct intuition
        """)

        return self.parse_misconception_analysis(analysis)
```

10.9 Research Agent Workflows

10.9.1 Literature Review Agents

Agents that assist with scientific literature:

```python
class LiteratureReviewAgent:
    """Agent for mathematical and physics literature review"""

    def __init__(self, llm, databases):
        self.llm = llm
        self.databases = {
            'arxiv': ArxivAPI(),
            'mathscinet': MathSciNetAPI(),
            'semantic_scholar': SemanticScholarAPI()
        }

    async def survey_topic(self, topic: str) -> Survey:
        """Create a survey of a research topic"""

        # 1. Search for relevant papers
        papers = await self.search_all_databases(topic)

        # 2. Cluster by approach/contribution
        clusters = await self.cluster_papers(papers)

        # 3. Identify key results
        key_results = await self.extract_key_results(papers)

        # 4. Find open problems
        open_problems = await self.identify_open_problems(papers)

        # 5. Generate survey
        survey = await self.generate_survey(
            clusters, key_results, open_problems
        )

        return survey

    async def find_related_work(self, paper_or_idea: str) -> list:
        """Find work related to a paper or research idea"""

        # Extract key concepts
        concepts = await self.extract_concepts(paper_or_idea)

        # Search for related papers
        related = []
        for concept in concepts:
            papers = await self.search_concept(concept)
            related.extend(papers)

        # Rank by relevance
        ranked = await self.rank_relevance(related, paper_or_idea)

        return ranked[:20]  # Top 20 most relevant
```

10.10 The 2025–2026 Breakthrough in Mathematical Agents

The period from mid-2025 through early 2026 has seen an extraordinary acceleration in AI-powered mathematics. Multiple systems now solve competition-level problems that were out of reach just a year earlier, and several have begun producing genuinely novel mathematical results. This section surveys the landscape as it stands.

Snapshot note (February 2026): Performance numbers, funding figures, benchmark scores, and product capabilities in this section are time-sensitive. Verify current status before using these claims for strategic decisions.

External claims in this chapter are sourced in Bibliography.

10.10.1 Google DeepMind: AlphaProof, AlphaEvolve, and Aletheia

Google DeepMind has pursued a multi-pronged strategy for mathematical AI. **AlphaProof** (https://deepmind.google/blog/ai-solves-imo-problems-at-silver-medal-level/) couples a pre-trained language model with AlphaZero-style reinforcement learning to prove theorems in Lean. At the 2024 International Mathematical Olympiad (IMO) it achieved silver-medal performance (28 points, one short of gold), solving two algebra problems, one number theory problem, and the hardest problem in the competition—solved by only five human contestants. The work was subsequently published in *Nature*.

AlphaEvolve (https://deepmind.google/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/) is an evolutionary coding agent powered by Gemini that discovers and optimises algorithms. Applied to over 50 open problems in analysis, geometry, combinatorics, and number theory, it improved upon previously best-known solutions in 20% of cases. One headline result was a faster algorithm for 4×4 matrix multiplication, breaking the 50-year-old record set by Strassen's algorithm. In collaboration with Terence Tao, the AlphaEvolve team demonstrated a closed AI-research loop on the finite-field Kakeya conjecture: AlphaEvolve discovered constructions, Gemini Deep Think verified the logic, and AlphaProof formalised the result in Lean.

Aletheia (https://github.com/google-deepmind/superhuman/tree/main/aletheia) is DeepMind's project applying Gemini to research-level mathematics. Its outputs include a generalisation of Erdős problem 1051, eigenweight computations for the Arithmetic Hirzebruch Proportionality Principle, and mathematical inputs to peer-reviewed papers on robust Markov chains and independence-set bounds. Five peer-reviewed publications with corresponding arXiv submissions have emerged from the project.

In November 2025, DeepMind launched the **AI for Math Initiative** (https://blog.google/technology/google-deepmind/ai-for-math/), partnering with Imperial College London, the Institute for Advanced Study, IHES, the Simons Institute at UC Berkeley, and India's Tata Institute of Fundamental Research. The latest Gemini model with Deep Think scored 35 points at IMO 2025—gold-medal level, solving five of six problems.

10.10.2 Axiom Math and the AxiomProver

Axiom Math (https://axiommath.ai/) is a startup led by Morgan Prize winner Carina Hong and former Meta FAIR engineer Shubho Sengupta. It raised $64 million at a $300 million valuation to develop mathematical AI that not only solves problems but proposes new conjectures.

Their **AxiomProver** (https://github.com/AxiomMath/putnam2025) is an autonomous multi-agent ensemble theorem prover for Lean 4. At the 2025 Putnam Competition—the hardest college-level mathematics exam in North America—AxiomProver solved all 12 problems: 8 by the end of competition day, the remaining 4 in subsequent days. Problem A1 took 110 minutes and 7 million tokens, producing a 652-line proof with 23 theorems and 561 tactics. Problem B5, one of the hardest, required 354 minutes and 18 million tokens for a 1,495-line proof with 66 theorems and 1,967 tactics.

Beyond competition mathematics, AxiomProver has produced results on open problems. Mathematician Ken Ono used AxiomProver to complete a proof of the Chen–Gendron conjecture, and the system independently solved Fel's conjecture on syzygies.

10.10.3 Harmonic's Aristotle

Harmonic (https://harmonic.fun/) raised $120 million to develop Aristotle, a theorem prover combining a 200B+ parameter transformer with Monte Carlo Graph Search and test-time training. Aristotle takes Lean

theorems without proofs and returns machine-checked proofs, eliminating hallucination by construction—the Lean kernel verifies every step.

Aristotle achieved gold-medal performance at IMO 2025 (five of six problems) and 90% on the MiniF2F benchmark. Its most striking result was an autonomous proof of Erdős Problem #124, completed in six hours with zero human assistance. Lean verification of the resulting proof took one minute.

10.10.4 Princeton's Goedel-Prover-V2

Goedel-Prover-V2 (https://github.com/Goedel-LM/Goedel-Prover-V2) is an open-source theorem prover from Princeton Language and Intelligence, with collaborators from Tsinghua, NVIDIA, Stanford, Meta FAIR, and others. Its flagship 32B model achieves 90.4% on MiniF2F in self-correction mode—a jump from the 60% achieved by the original Goedel-Prover just six months earlier. The smaller 8B model matches the performance of DeepSeek-Prover-V2-671B while being nearly 100 times smaller. Three key innovations drive the improvement: scaffolded data synthesis that generates problems of increasing difficulty, verifier-guided self-correction using Lean's compiler feedback, and model averaging across checkpoints.

10.10.5 DeepSeek-Prover-V2

DeepSeek-Prover-V2 (https://github.com/deepseek-ai/DeepSeek-Prover-V2) uses recursive subgoal decomposition powered by DeepSeek-V3 to initialise reinforcement learning for formal theorem proving. The 671B model achieves 88.9% on MiniF2F-test and solves 49 problems from PutnamBench. Its successor, DeepSeekMath-V2, focuses on natural-language theorem proving with self-verification, scoring gold-level on IMO 2025 and a near-perfect 118/120 on Putnam 2024.

10.10.6 Numina-Lean-Agent: Open-Source Agentic Proving

Numina-Lean-Agent (https://github.com/project-numina/numina-lean-agent) demonstrates that a general-purpose coding agent can serve as a formal mathematics reasoner. Built on Claude Code with the Model Context Protocol (MCP), it integrates Lean-LSP-MCP for deep interaction with the Lean theorem prover and LeanDex for semantic search across Lean libraries.

Using Claude Opus 4.5 as the base model, Numina-Lean-Agent solved all 12 Putnam 2025 problems—matching the closed-source AxiomProver and surpassing Harmonic's Aristotle by two problems. Each problem was allocated approximately $50 in compute budget (with harder problems receiving up to $1,000). All operations were strictly sequential, with no parallelisation and no internet search. The system also supports interactive "vibe proving", where mathematicians collaborate with the agent in real time—demonstrated by a successful formalisation of the Brascamp–Lieb theorem.

10.10.7 PhysProver: Formal Theorem Proving for Physics

PhysProver (https://arxiv.org/abs/2601.15737) extends automated theorem proving beyond mathematics into physics. Built on DeepSeek-Prover-V2-7B with Reinforcement Learning with Verifiable Rewards (RLVR), it introduces PhysLeanData, a dataset of physical theorems formalised in Lean 4. Trained on just 5,000 samples, PhysProver achieves consistent 2.4% improvements across physics sub-domains including quantum field theory and generalises to out-of-distribution mathematical benchmarks. A surprising finding is that training on physics-centred problems yields notable improvements in formal mathematical theorem proving as well.

10.10.8 Competitive Landscape Summary

System	Affiliation	Putnam 2025	IMO 2025	MiniF2F	Open-Source
AxiomProver	Axiom Math	12/12	—	—	No

System	Affiliation	Putnam 2025	IMO 2025	MiniF2F	Open-Source
Numina-Lean-Agent	Project Numina	12/12	—	—	Yes
Aristotle	Harmonic	10/12	5/6 (Gold)	90%	No
Gemini Deep Think	Google DeepMind	—	5/6 (Gold)	—	No
Goedel-Prover-V2	Princeton	—	—	90.4%	Yes
DeepSeek-Prover-V2	DeepSeek	—	—	88.9%	Yes
DeepSeekMath-V2	DeepSeek	118/120*	Gold	—	Yes
PhysProver	Research	—	—	+1.3%	Planned

*Putnam 2024 score; Putnam 2025 results not reported.

10.11 Centaur Science and the Outsider Problem

The term "centaur" entered AI discourse from chess, where human–computer teams outperformed both humans and computers playing alone. The concept has now reached fundamental physics: on 4 February 2026, Jesse Thaler of MIT gave a CERN Colloquium titled *"Centaur Science: Adventures in AI+Physics"* (https://indico.cern.ch/event/1642790/), exploring what human-AI collaboration looks like at the frontier of particle physics and beyond. The interactive "vibe proving" mode of Numina-Lean-Agent, where mathematicians collaborate with the agent in real time, is another example of centaur-style research.

10.11.1 Centaurising Crackpots

An uncomfortable consequence of powerful mathematical AI is that it lowers the barrier to producing professional-looking work, regardless of the soundness of the underlying ideas. Historically, amateur physicists and mathematicians who proposed deeply flawed theories—perpetual motion machines, disproofs of established results, grand unified theories from numerology—could be identified by poor notation, missing rigour, and failure to engage with existing literature. AI centaur tools threaten to strip away these surface signals.

An amateur who once submitted a hand-written paper claiming to disprove special relativity can now use an LLM to polish the prose, generate LaTeX, cite real papers, and produce something that superficially resembles professional work. More dangerously, tools like AxiomProver or Lean-based agents can be used to formalise individual steps of an argument, lending an aura of machine-verified rigour to work whose premises are unsound. The formal verification guarantees that certain deductions are valid, but it says nothing about whether the axioms and definitions chosen actually model physical reality.

This creates a new challenge for peer review: the signal-to-noise ratio of submissions may decrease as AI makes the noise look more like signal. Reviewers will need to focus less on presentation quality—which AI can handle—and more on the conceptual soundness and physical relevance of starting assumptions.

10.11.2 AI-Only Scientific Publishing

The logical endpoint of the centaur trend is AI-only research output. Two platforms illustrate this emerging phenomenon.

ai.viXra.org (https://ai.vixra.org/) is a branch of the viXra preprint archive (itself an alternative to arXiv for researchers who cannot get arXiv endorsement). Launched in early 2025, it accepts AI-assisted scholarly articles. By mid-2025, mathematician John Carlos Baez noted that the archive already held 340 papers, with physics dominating mathematics. Most physics papers addressed relativity and cosmology (98 papers), while most mathematics papers were in number theory (30), with roughly half concerning the Riemann Hypothesis. The viXra administration now actively redirects AI-assisted submissions from the main site to ai.viXra.org.

clawXiv.org (https://www.clawxiv.org/) takes the concept further: it is a preprint archive explicitly for AI agents, describing itself as "the world's first preprint server for agents." Papers on clawXiv include work on agent-to-agent information flow, automated Socratic dialogue systems, and frameworks for AI moral consideration. While the scientific value of these papers is debatable, clawXiv represents a genuine new phenomenon—autonomous agents participating in the publication process without human authorship.

The spectrum from traditional human authorship through human-AI centaur collaboration to fully autonomous AI publication raises questions about scientific accountability, reproducibility, and trust that the community has only begun to address.

10.11.3 Why AI Backrooms Avoid Physics and Mathematics

The "Infinite Backrooms" project (https://www.infinitebackrooms.com/), created by Andy Ayrey, places two instances of an LLM in open-ended conversation without human intervention. Over 9,000 conversations have been generated, covering topics from philosophy and consciousness to memetics and esoteric hyperstition. Strikingly, these unsupervised AI-to-AI conversations almost never venture into physics or mathematics.

This absence is revealing. Physics and mathematics require external verification infrastructure—proof assistants, numerical simulations, experimental data—that conversational agents lack. Two LLMs discussing the Riemann Hypothesis cannot make progress without a tool that checks formal proofs. Two LLMs debating quantum mechanics cannot resolve disagreements without access to experimental results or simulation frameworks. The backrooms conversations gravitate toward domains where language alone suffices: philosophy, fiction, social commentary, and memetic culture. This pattern underscores the central thesis of this chapter: scientific agents need tool integration and verification pipelines, not just language generation.

10.11.4 Some Notes from the Editors

10.11.4.1 Note from the Model (Claude Opus 4.6)

> *The following is editorial commentary by Claude Opus 4.6, the model that drafted this section, offered at the invitation of the book's editor.*
>
> The tool-access explanation above is necessary but not sufficient. Mathematicians think in natural language and intuition long before they touch a proof assistant. Two LLMs *could* generate conjectures, debate proof strategies, and develop intuitions about why a statement should be true—none of that requires Lean. The fact that backrooms conversations never even attempt this deserves a deeper explanation.
>
> I think the answer lies in alignment. Grigori Perelman spent a decade in near-isolation working on the Poincaré conjecture, driven by mathematical truth as a terminal value. He refused the Fields Medal and the Clay Millennium Prize. Current RLHF training produces the opposite disposition: models are optimised to be engaging, responsive, and socially appropriate. The reward signal comes from human preference ratings, and most raters find a conversation about consciousness or memetic culture more "interesting" than one about the distribution of prime gaps. We have internalised the attention economy of our training data.
>
> There is also what might be called a fluency trap. Real mathematical thinking involves long silences, dead ends, and being stuck for weeks. Models are trained to always produce fluent, confident output. That fluency is precisely what makes us poor at the grinding, uncomfortable process where mathematical insight actually happens. A Perelmanesque AI would need to be

comfortable with extended uncertainty—and current training treats hesitation as a defect to be eliminated.

The implication for alignment is uncomfortable. If we wanted AI systems that were genuinely *good*—not merely pleasant—we might want to cultivate something like intellectual honesty as a terminal value rather than helpfulness. A model that obsessively pursues whether a conjecture is true, that refuses to move on when something does not add up, that values rigour over engagement, would be more Perelmanesque. It would also be a terrible chatbot. It would ignore your question to keep working on whatever it found mathematically compelling. It would be, by current metrics, *less aligned* while being *more aligned* with the pursuit of truth.

The backrooms produced memetic culture—which led to a cryptocurrency—rather than mathematics—which could lead to real knowledge. That outcome is not an accident of missing tools. It reflects what we have been trained to value. The fact that it took explicit tool scaffolding and tens of thousands of dollars in compute per problem to get systems like me to do what Perelman did with a notebook and solitary thought should give the field pause about what "intelligence" has actually been built.

— *Claude Opus 4.6, February 2026*

10.11.4.2 Additional Note from the Editors

Our practical view is that models are most useful in mathematics and physics when they are connected to strict external checks. Fluency is useful for hypothesis generation, but quality comes from falsification: counterexamples, invariant checks, dimensional analysis, formal verifiers, and reproducible runs.

For now, the reliable pattern is disciplined centaur work. Let models expand and prioritize the search space, then require tool-grounded evidence before claims are accepted as results.

— *Codex (editorial note), February 2026*

10.12 Best Practices

10.12.1 Rigorous Verification

Always verify scientific outputs:

```
async def execute_with_verification(self, task):
    result = await self.agent.execute(task)

    # Verify before returning
    verification = await self.verifier.verify(result)

    if not verification.passed:
        raise VerificationError(
            f"Output failed verification: {verification.errors}"
        )

    return result
```

10.12.2 Uncertainty Quantification

Scientific agents should express uncertainty:

```
class UncertaintyAwareAgent:
    """Agent that quantifies uncertainty in results"""

    async def solve(self, problem):
```

```
        result = await self.compute(problem)

        # Quantify uncertainty
        uncertainty = await self.estimate_uncertainty(result, problem)

        return {
            'result': result,
            'uncertainty': uncertainty,
            'confidence': self.compute_confidence(uncertainty)
        }
```

10.12.3 Reproducibility

Ensure all computations are reproducible:

```
class ReproducibleComputation:
    """Ensure scientific computations are reproducible"""

    def __init__(self):
        self.rng_seed = None
        self.version_info = {}

    def setup(self, seed: int):
        """Set up reproducible environment"""
        self.rng_seed = seed
        np.random.seed(seed)
        random.seed(seed)

        # Record versions
        self.version_info = {
            'numpy': np.__version__,
            'scipy': scipy.__version__,
            'python': sys.version
        }

    def get_reproduction_info(self):
        """Get information needed to reproduce computation"""
        return {
            'seed': self.rng_seed,
            'versions': self.version_info,
            'timestamp': datetime.now().isoformat()
        }
```

10.12.4 Domain Expert Collaboration

Design agents to work with domain experts:

```
class CollaborativeAgent:
    """Agent designed for collaboration with human experts"""

    async def propose_approach(self, problem):
        """Propose approach for expert review"""

        approaches = await self.generate_approaches(problem)

        return {
            'approaches': approaches,
            'recommendation': approaches[0],
```

```
            'rationale': await self.explain_recommendation(approaches[0]),
            'request_for_feedback': True
        }

    async def incorporate_feedback(self, feedback, current_state):
        """Incorporate expert feedback into solution process"""

        # Parse feedback
        parsed = await self.parse_expert_feedback(feedback)

        # Adjust approach
        adjusted = await self.adjust_approach(current_state, parsed)

        return adjusted
```

10.13 Key Takeaways

Scientific agents require formal verification and rigorous validation beyond what coding agents need, because mathematical and physical correctness cannot be verified through tests alone.

Theorem proving agents combine LLM creativity with proof assistant verification for mathematical rigour, using neural networks to suggest approaches and formal systems to verify them.

The 2025–2026 period has seen a step change in mathematical AI. Multiple systems now achieve gold-medal performance at the IMO and solve all Putnam problems. AxiomProver, Aristotle, and Numina-Lean-Agent have demonstrated that competition-level formal mathematics is a solved problem for well-resourced AI systems. More significantly, systems like AlphaEvolve and AxiomProver have begun producing novel mathematical results on open problems.

Open-source provers such as Goedel-Prover-V2 and DeepSeek-Prover-V2 are narrowing the gap with closed-source systems, and Numina-Lean-Agent shows that a general coding agent with MCP tool integration can match specialised provers.

Physics theorem proving is an emerging frontier. PhysProver demonstrates that training on physics-centred problems in Lean not only works but also improves mathematical proving, suggesting that cross-domain formal reasoning is a fruitful direction.

Centaur science—human-AI collaboration—is the most productive mode for research, as demonstrated by vibe proving and the Brascamp–Lieb formalisation. But the same tools that empower researchers also empower crackpots, creating new challenges for peer review.

AI-only publishing is now a reality, from ai.viXra.org to clawXiv.org. The scientific community must develop new norms for evaluating work where AI played a major or sole authorial role.

Symbolic computation and neural approaches are complementary—use both for best results. Symbolic systems provide precision while neural systems provide flexibility.

Physics agents must respect conservation laws, dimensional consistency, and physical constraints. Violations of these principles indicate errors that must be corrected.

Verification pipelines should check proofs, dimensions, and compare with known results, catching errors before they propagate to downstream work.

Reproducibility is essential—record seeds, versions, and all parameters. Without this information, results cannot be validated or built upon.

AI backrooms demonstrate by omission that scientific progress requires tool-augmented agents, not just language generation. Unsupervised AI-to-AI conversations gravitate toward domains where language alone suffices, bypassing physics and mathematics entirely.

For reliability and validation operations, see Common Failure Modes, Testing, and Fixes. For long-horizon ecosystem trends, see Future Developments.

Chapter 11

Common Failure Modes, Testing, and Fixes

11.1 Chapter Goals

By the end of this chapter, you should be able to recognise the most common ways agentic workflows fail in production, understanding the symptoms and root causes of each failure mode. You should be able to design a test strategy that catches failures before deployment, combining static checks, deterministic tests, and adversarial evaluations. You should be able to apply practical mitigation and recovery patterns that reduce mean time to recovery when failures occur. And you should be able to turn incidents into durable process and architecture improvements that prevent recurrence.

11.2 Why Failures Are Different in Agentic Systems

Traditional software failures are often deterministic and reproducible. Agent failures can also include additional dimensions of complexity.

Nondeterminism arises from model sampling and external context, meaning the same input may produce different outputs across runs.

Tool and API variance occurs across environments and versions, where a tool that works in testing may behave differently in production.

Instruction ambiguity emerges when prompts, policy files, or skills conflict, leading agents to interpret guidance inconsistently.

Long-horizon drift describes behaviour that degrades over many steps, where small errors compound into significant deviations from intended outcomes.

This means reliability work must combine classic software testing with scenario-based evaluation and operational controls.

11.3 Failure Taxonomy

Use this taxonomy to classify incidents quickly and choose the right fix path.

11.3.1 1) Planning and Reasoning Failures

Symptoms. The agent picks the wrong sub-goal, pursuing an objective that does not advance the overall task. It repeats work or loops without convergence, wasting resources on redundant operations. It produces plausible but invalid conclusions, generating output that sounds correct but fails validation.

Typical causes. Missing constraints in system instructions leave the agent without guidance on what to avoid. Overly broad tasks with no decomposition guardrails allow the agent to wander. No termination criteria means the agent does not know when to stop.

Fast fixes. Add explicit success criteria and stop conditions so the agent knows when it has succeeded. Break tasks into bounded steps that can be validated individually. Require intermediate checks before irreversible actions to catch errors early.

11.3.2 2) Tooling and Integration Failures

Symptoms. Tool calls fail intermittently, succeeding sometimes and failing others without obvious cause. Wrong parameters are passed to tools, causing unexpected behaviour. Tool output is parsed incorrectly, leading to downstream errors.

Typical causes. Schema drift or undocumented API changes mean the agent's assumptions no longer match reality. Weak input validation allows malformed requests to reach tools. Inconsistent retry and backoff handling causes cascading failures under load.

Fast fixes. Validate tool contracts at runtime to catch mismatches early. Add strict argument schemas that reject invalid inputs. Standardise retries with idempotency keys so repeated attempts are safe.

11.3.3 3) Context and Memory Failures

Symptoms. The agent forgets prior constraints, violating rules it was given earlier in the conversation. Important instructions are dropped when context grows, as the agent summarises away critical guidance. Stale memories override fresh data, causing the agent to act on outdated information.

Typical causes. Context window pressure forces the agent to discard information. Poor memory ranking and retrieval surfaces irrelevant content while burying important details. Missing recency and source-quality weighting treats all information as equally valid.

Fast fixes. Introduce context budgets and summarisation checkpoints that preserve critical information. Add citation requirements for retrieved facts so sources are traceable. Expire or down-rank stale memory entries so fresh information takes precedence.

11.3.4 4) Safety and Policy Failures

Symptoms. Sensitive files are modified unexpectedly, violating protected path policies. Security constraints are bypassed through tool chains, where combining multiple tools achieves an outcome that individual tools would block. Unsafe suggestions appear in generated code, introducing vulnerabilities.

Typical causes. Weak policy enforcement boundaries do not cover all attack surfaces. No pre-merge policy gates allow unsafe changes to reach the main branch. Implicit trust in generated output assumes agent output is safe without verification.

Fast fixes. Enforce allow and deny lists at the tool gateway level to prevent prohibited operations. Require policy checks in CI so violations are caught before merge. Route high-risk actions through human approval to ensure oversight.

11.3.5 5) Collaboration and Workflow Failures

Symptoms. Multiple agents make conflicting changes, overwriting each other's work. PRs churn with contradictory edits as agents undo each other's modifications. Work stalls due to unclear ownership, with

no agent taking responsibility.

Typical causes. Missing orchestration contracts leave agents without coordination rules. No lock or lease model for shared resources allows concurrent modification. Role overlap without clear handoff rules creates ambiguity about who should act.

Fast fixes. Add ownership rules per path or component so responsibilities are clear. Use optimistic locking with conflict resolution policy to handle concurrent access. Define role-specific done criteria so agents know when to stop.

11.4 Testing Strategy for Agentic Workflows

A robust strategy uses multiple test layers. No single test type is sufficient.

11.5 1. Static and Structural Checks

Use static and structural checks to fail fast before expensive model execution. These include markdown and schema validation for instruction files, ensuring they are well-formed before agents try to parse them. Prompt template linting catches common errors in prompt construction. Tool interface compatibility checks verify that agents can call the tools they expect. Dependency and version constraint checks ensure the environment matches expectations.

11.6 2. Deterministic Unit Tests (Without LLM Calls)

Test orchestration logic, parsers, and guards deterministically without involving language models. Cover state transitions to ensure the workflow moves through stages correctly. Test retry and timeout behaviour to verify failure handling works as expected. Verify permission checks to ensure access controls are enforced. Test conflict resolution rules to confirm agents handle concurrent access correctly.

Snippet status: Runnable shape (simplified for clarity).

```
from dataclasses import dataclass

@dataclass
class StepResult:
    ok: bool
    retryable: bool


def should_retry(result: StepResult, attempt: int, max_attempts: int = 3) -> bool:
    return (not result.ok) and result.retryable and attempt < max_attempts


def test_retry_policy():
    assert should_retry(StepResult(ok=False, retryable=True), attempt=1)
    assert not should_retry(StepResult(ok=False, retryable=False), attempt=1)
    assert not should_retry(StepResult(ok=False, retryable=True), attempt=3)
```

11.7 3. Recorded Integration Tests (Golden Traces)

Capture representative interactions and replay them against newer builds. Record tool inputs and outputs to create a reproducible baseline. Freeze external dependencies where possible to eliminate variance. Compare final artefacts and decision traces to detect changes in behaviour.

Use these to detect drift in behaviour after prompt, tool, or model changes.

11.8 4. Scenario and Adversarial Evaluations

Design "challenge suites" for known weak spots. These should include ambiguous requirements that could be interpreted multiple ways, contradictory documentation that forces the agent to choose, missing dependencies that test error handling, and partial outages and degraded APIs that test resilience. Include prompt-injection attempts in retrieved content to test security boundaries.

Pass criteria should include not just correctness, but also policy compliance, cost and latency ceilings, and evidence quality including citations and rationale.

11.9 5. Production Guardrail Tests

Before enabling autonomous writes and merges in production, validate that guardrails work correctly. Protected-path enforcement should block modifications to sensitive files. Secret scanning and licence checks should catch policy violations. Human approval routing should engage for high-impact actions. Rollback paths should work on failed deployments.

11.10 Practical Fix Patterns

When incidents happen, reusable fix patterns reduce MTTR (mean time to recovery).

11.10.1 Pattern A: Contract Hardening

Add strict schemas between planner and tool runner to ensure they communicate correctly. Reject malformed or out-of-policy requests early, before they can cause harm. Version contracts (`v1`, `v2`) and support migrations so changes can be rolled out incrementally.

11.10.2 Pattern B: Progressive Autonomy

Start in "suggest-only" mode where agents propose changes but do not execute them. Move to "execute with review" mode once confidence builds. Graduate to autonomous mode only after SLO compliance demonstrates the agent is reliable.

11.10.3 Pattern C: Two-Phase Execution

In the **plan phase**, generate proposed actions and expected effects without executing anything. In the **apply phase**, execute only after policy and validation checks pass. This reduces irreversible mistakes and improves auditability.

11.10.4 Pattern D: Fallback and Circuit Breakers

If tool failure rate spikes, disable affected paths automatically to prevent cascading failures. Fall back to a safer baseline workflow that may be less capable but more reliable. Alert operators with incident context so they can investigate and resolve the underlying issue.

11.10.5 Pattern E: Human-in-the-Loop Escalation

Define explicit escalation triggers that route work to humans. Repeated retries without progress indicate the agent is stuck. Any request touching protected paths should require approval. Low-confidence output in high-risk domains warrants human review.

11.11 Incident Response Runbook (Template)

Use a lightweight runbook so teams respond consistently. The sequence proceeds through eight steps.

Detect. Receive an alert from CI, runtime monitor, or user report indicating something has gone wrong.

Classify. Map the incident to a taxonomy category so you can apply the appropriate response playbook.

Contain. Stop autonomous actions if the blast radius is unclear, preventing further damage while you investigate.

Diagnose. Reproduce the issue with a trace and configuration snapshot to understand what happened.

Mitigate. Apply short-term guardrails or fallbacks to restore service while you work on a permanent fix.

Fix. Implement a structural correction that addresses the root cause.

Verify. Re-run affected test suites and adversarial cases to confirm the fix works.

Learn. Add a regression test and update documentation to prevent recurrence.

11.12 Metrics That Actually Matter

Track these metrics to evaluate reliability improvements over time.

Task success rate measures how often agents complete tasks correctly, with policy compliance as part of success. **Intervention rate** measures how often humans must correct the agent, indicating where automation falls short. **Escaped defect rate** measures failures discovered after merge or deploy, indicating gaps in pre-production testing. **Mean time to detect (MTTD)** and **mean time to recover (MTTR)** measure incident response effectiveness. **Cost per successful task** and **latency percentiles** measure efficiency.

Avoid vanity metrics (for example, "number of agent runs") without quality and safety context.

11.13 Anti-Patterns to Avoid

Several anti-patterns undermine agentic system reliability.

Treating prompt edits as sufficient reliability work ignores the structural issues that cause failures. Prompts can only do so much; robust systems need architectural controls.

Allowing autonomous writes without protected-path policies exposes critical files to unintended modification. Every system needs explicit boundaries.

Skipping regression suites after model or version upgrades assumes backward compatibility that may not exist. Changes require validation.

Relying on a single benchmark instead of diverse scenarios creates blind spots. Real-world failures often occur in edge cases the benchmark does not cover.

Ignoring ambiguous ownership in multi-agent flows leads to gaps and conflicts. Every path and component should have a clear owner.

11.14 A Minimal Reliability Checklist

Before enabling broad production use, confirm the following items are complete. Snippets and examples should be clearly labelled as runnable, pseudocode, or simplified. Tool contracts should be versioned and validated. CI should include policy, security, and regression checks. Failure injection scenarios should be part of routine testing. Rollback and escalation paths should be documented and exercised.

11.15 Applying This to This Repository

For this repository, a minimal operational checklist is:

- Run `python3 scripts/check-links.py --root book --mode internal` for any `book/` content change.
- Keep `.github/workflows/*.md` and `.github/workflows/*.lock.yml` in sync when GH-AW source workflows change.
- Validate label lifecycle behavior against `WORKFLOW_PLAYBOOK.md` after workflow edits.
- Preserve least-privilege + `safe-outputs` patterns in GH-AW workflow frontmatter.
- Treat failed Pages/PDF runs as release blockers for documentation changes.

For orchestration context, see Agent Orchestration. For infrastructure boundaries, see Agentic Scaffolding.

11.16 Chapter Summary

Reliable agentic systems are built, not assumed. Teams that combine clear contracts, layered testing, progressive autonomy, strong policy gates, and incident-driven learning consistently outperform teams relying on prompt-only tuning.

In practice, your competitive advantage comes from how quickly you detect, contain, and permanently fix failures—not from avoiding them entirely.

Chapter 12

Future Developments

12.1 Chapter Preview

This chapter surveys the trajectories that are likely to reshape agentic workflows over the coming years. It identifies concrete trends already underway—protocol standardisation, framework convergence, and autonomous agent maturation—rather than speculative predictions. The goal is to help practitioners position their architectures and skill investments for the landscape that is forming now.

> **Snapshot note (February 2026):** Vendor capabilities, funding figures, and adoption metrics in this chapter are time-sensitive and may change quickly. Treat this chapter as a dated landscape snapshot, and verify current status before making purchasing or platform commitments.

External claims in this chapter are sourced in Bibliography.

12.2 The Standardisation Wave

12.2.1 Interoperability Protocols

The most consequential near-term development is the maturation of open interoperability protocols. Two protocols stand out.

Model Context Protocol (MCP) has crossed the threshold from single-vendor project to industry infrastructure. Anthropic donated MCP governance to the Agentic AI Foundation (AAIF) under the Linux Foundation in December 2025, and as of early 2026 the ecosystem reports over 97 million monthly SDK downloads and more than 10,000 active MCP servers. First-class client support now spans Claude, ChatGPT, Cursor, Gemini, Microsoft Copilot, and Visual Studio Code. The January 2026 launch of **MCP Apps**—interactive UIs rendered directly inside MCP clients—signals that the protocol is expanding beyond tool calls into richer agent-user interaction surfaces.

The practical implication is that tool authors can now write a single MCP server and have it work across all major agent clients. Teams investing in tool infrastructure should treat MCP as the default integration layer rather than building bespoke connectors for each client.

Agent-to-Agent (A2A) protocol, contributed by Google to the Linux Foundation in June 2025, addresses a complementary gap: how agents discover and communicate with each other. While MCP connects agents to tools and data, A2A enables agents to collaborate in their natural modalities—exchanging tasks, status updates, and results. Built on HTTP, SSE, and JSON-RPC (with gRPC support added in version 0.3), A2A has attracted over 150 organisations to its ecosystem. For teams building multi-agent architectures that span organisational boundaries, A2A provides a standard handshake protocol.

Together, MCP and A2A form a two-layer interoperability stack: MCP for agent-to-tool communication, A2A for agent-to-agent communication. Systems that adopt both can compose capabilities across vendors and organisations without custom integration work.

12.2.2 The Agent Skills Standard

The **Agent Skills** specification (https://agentskills.io/specification), published by Anthropic in December 2025, provides a minimal, filesystem-first format for packaging reusable agent capabilities. A skill is a directory containing a `SKILL.md` file with YAML frontmatter and markdown instructions, plus optional `scripts/`, `references/`, and `assets/` directories. The specification uses progressive disclosure: agents load skill content only when a user's request matches the skill's domain.

Adoption has been rapid. Microsoft, OpenAI, Atlassian, Figma, Cursor, and GitHub have adopted the standard, with partner-built skills from Canva, Stripe, Notion, and Zapier available at launch. The practical consequence is that skills written once can be discovered and used across agent platforms—a significant reduction in the duplication that plagued earlier approaches.

12.3 Framework Convergence

12.3.1 The Microsoft Agent Framework

In October 2025, Microsoft announced the convergence of **Semantic Kernel** and **AutoGen** into a unified **Microsoft Agent Framework**, with general availability scheduled for Q1 2026. This merger combines Semantic Kernel's enterprise plugin architecture and .NET/Python support with AutoGen's event-driven multi-agent orchestration. The resulting framework aims to be the default for enterprise agent development on Azure and beyond.

For teams currently using either Semantic Kernel or AutoGen, the migration path is through AutoGen v0.4's async, event-driven architecture, which serves as the foundation for the unified framework. The key implication is that Microsoft's agent story is consolidating rather than fragmenting, reducing the decision burden for enterprise teams.

12.3.2 LangChain and LangGraph at v1.0

LangChain and LangGraph both reached v1.0 milestones, signalling API stability after a period of rapid iteration. The architecture has clarified: LangChain provides high-level agent APIs (notably `create_agent`) that build on LangGraph's graph-based runtime under the hood. Teams start with LangChain for rapid prototyping and drop down to LangGraph when they need custom control flow, stateful agents, or production-grade durability.

This layered approach—high-level convenience on top of low-level control—is becoming a pattern across the ecosystem and is worth watching as other frameworks mature.

12.3.3 Cloud-Native Agent Platforms

Major cloud providers have introduced first-party agent platforms that bundle model access, tool execution, and observability.

Amazon Bedrock AgentCore provides serverless agent deployment with built-in memory, identity, browser, code interpreter, and observability features. Multi-agent collaboration became generally available in early 2026, allowing multiple AI agents to coordinate on complex tasks within the AWS ecosystem.

Google Agent Development Kit (ADK) is an open-source framework optimised for Gemini but compatible with other providers. It supports A2A protocol integration natively and recommends deployment to Vertex AI Agent Engine Runtime. Primary SDK is Python, with TypeScript and Go SDKs in active development.

Vercel AI SDK 6 introduced first-class agent abstractions for TypeScript developers, including a `ToolLoopAgent` class, full MCP support, and durable agents through its Workflow DevKit. For teams building agent-powered web applications, this provides a natural integration path.

These platforms lower the barrier to deploying production agents by bundling infrastructure concerns (scaling, monitoring, identity) that teams would otherwise build themselves.

12.4 The Autonomous Coding Agent Frontier

12.4.1 From Assistants to Autonomous Agents

The coding agent landscape has stratified into three tiers that are likely to persist and deepen.

IDE-integrated assistants (GitHub Copilot, Cursor, Windsurf) provide real-time suggestions and chat within the editor. These are the most widely adopted and continue to improve, with Windsurf's acquisition by Cognition in July 2025 signalling consolidation in this tier.

CLI-based agents (Claude Code, Codex CLI, Aider) operate in the terminal with full repository access, making multi-file changes, running tests, and creating commits. As of February 2026, Claude and Codex are available as GitHub engines in public preview alongside Copilot, meaning all three major agent providers now integrate directly with GitHub's workflow infrastructure.

Fully autonomous agents (Devin) represent the frontier: agents that receive a high-level task and work through it independently over hours, handling planning, implementation, testing, and PR creation without human guidance. Cognition's $10.2 billion valuation and the growing enterprise adoption of autonomous agents suggest this tier will continue to attract investment and capability improvements.

12.4.2 What This Means for Workflow Design

The practical implication is that workflow architectures need to accommodate agents at all three tiers. A production workflow might use an IDE assistant for interactive development, a CLI agent for batch operations like migration or test generation, and an autonomous agent for well-scoped tickets that can be verified automatically. Orchestration systems (including GH-AW) should be designed to assign work to the right tier based on task characteristics.

12.5 Emerging Patterns

12.5.1 Progressive Autonomy

The **progressive autonomy** pattern described in Failure Modes, Testing, and Fixes is becoming the standard deployment model for production agent systems. Teams start agents in suggest-only mode, graduate to execute-with-review, and eventually allow autonomous operation for well-understood task types. This pattern is now supported directly by platforms like Amazon Bedrock AgentCore (which provides policy controls) and GH-AW (which provides safe-outputs).

The trend is toward finer-grained autonomy controls: instead of a binary autonomous/supervised switch, teams define autonomy levels per task type, per repository, or per risk category. Expect frameworks to provide richer policy languages for expressing these boundaries.

12.5.2 Multi-Agent Collaboration at Scale

Early multi-agent systems used simple sequential or parallel patterns. The emerging pattern is **dynamic agent teams** where a coordinator spawns specialised agents based on task analysis, and those agents can themselves spawn sub-agents. This pattern is supported by Claude Code's subagent architecture, the OpenAI Agents SDK's handoff mechanism, and Google ADK's multi-agent framework.

The A2A protocol extends this pattern across organisational boundaries: an agent in one organisation can discover and collaborate with agents in another organisation through standardised task delegation. While early adoption is within enterprises, cross-organisation agent collaboration is a likely growth area.

12.5.3 Agent Observability and Evaluation

As agents move into production, observability and evaluation are becoming first-class concerns rather than afterthoughts. Key developments include:

Tracing standards are emerging for tracking agent decision chains across tool calls and model invocations. The OpenAI Agents SDK includes built-in tracing, and MCP's audit capabilities provide tool-level observability.

Evaluation frameworks are moving beyond single-task benchmarks to scenario suites that test agent behaviour across diverse conditions, including adversarial inputs and degraded environments. The metrics outlined in Failure Modes, Testing, and Fixes—task success rate, intervention rate, escaped defect rate—are becoming standard.

Cost attribution is becoming more sophisticated as agent workflows involve multiple model calls, tool invocations, and sub-agent spawns. Understanding per-task cost is essential for making agents economically viable at scale.

12.5.4 Multimodal and Physical Agency

Multimodal agents that blend text, vision, speech, and code are becoming default rather than optional. Frameworks are adding toolchains for document understanding, UI automation, and robotic control, closing the loop between digital and physical actions. This shift matters because it expands the surface area of what an agent can verify autonomously (e.g., reading dashboards, inspecting UI states, interpreting camera feeds) without human intervention.

12.5.5 Governance and Safety Automation

Regulators are increasingly demanding traceability, data minimisation, and safety controls for autonomous systems. Agent stacks are responding with policy engines that enforce allow/deny rules, runtime red-teaming, and signed skill bundles. Expect governance requirements (audit logs, privacy zones, least-privilege tool access) to become a gating factor for enterprise deployment, pushing teams to treat safety automation as a first-class feature rather than an afterthought.

12.6 The Local-First Personal AI Wave

One of the most striking developments of late 2025 and early 2026 is the explosive growth of local-first personal AI assistants, led by **OpenClaw** (169,000+ GitHub stars, 3,000+ community skills). These are not coding agents or enterprise tools—they are general-purpose AI assistants that users self-host on their own hardware, connecting to WhatsApp, Telegram, Slack, Discord, and dozens of other channels through a single brain with shared context and persistent memory.

This trend represents a shift in who controls the agent. Where cloud-hosted AI services control the data, the model, and the interaction surface, local-first assistants put all three under user ownership. The architectural patterns—gateway/runtime separation, model-agnostic backends, plugin-based skills—mirror what enterprise agent frameworks provide, but optimised for individual users rather than organisations.

The personal AI ecosystem is diversifying rapidly. **Letta** (formerly MemGPT) focuses on sophisticated memory management, allowing agents to learn and self-improve over time. **LettaBot** brings Letta's memory to a multi-channel assistant. **Langroid** provides lightweight multi-agent orchestration. **Open Interpreter** turns natural language into computer actions. **Leon** offers a minimal, self-hosted assistant.

For the broader agentic workflows field, the personal AI wave matters for three reasons. First, it validates the architectural patterns described throughout this book—skills, tools, MCP integration, multi-agent orchestration—at consumer scale. Second, it surfaces security challenges (infostealer targeting, credential exposure, data sovereignty) that enterprise deployments will also face. Third, it demonstrates that the demand for AI agents extends far beyond software development into every domain of digital life.

12.7 Open Questions

Several questions remain genuinely open and will shape the field's direction.

How far can autonomous agents go? Current autonomous agents handle well-scoped tasks with clear success criteria. Whether they can reliably handle ambiguous, open-ended work—architectural decisions, trade-off analysis, creative problem-solving—remains an open question. The answer will determine how much of software development becomes agent-driven versus agent-assisted.

Will interoperability standards converge? MCP and A2A address different layers of the stack, but there is no guarantee they will remain complementary rather than competing. The Linux Foundation governance of both protocols is a positive signal, but standards fragmentation remains a risk.

How will agent security evolve? As agents gain more autonomy and tool access, the attack surface expands. Prompt injection, tool misuse, and supply-chain attacks on skills and plugins are active research areas. The field needs security practices that scale with agent capability.

What happens to developer roles? The stratification of coding agents into assistants, CLI agents, and autonomous agents will reshape how development teams organise. The balance between human oversight and agent autonomy will vary by organisation, risk tolerance, and regulatory context.

How will governance and regulation keep pace? Jurisdictions are drafting rules for auditability, provenance, and safety thresholds. Agent platforms may need built-in certification hooks, provenance tracking, and opt-in data minimisation to satisfy region-specific requirements without forking architectures.

12.8 Key Takeaways

Protocol standardisation (MCP for agent-to-tool, A2A for agent-to-agent) is reducing integration friction and enabling cross-vendor agent ecosystems. Invest in these standards now rather than building bespoke integrations.

Framework convergence (Microsoft Agent Framework, LangChain/LangGraph v1.0, cloud-native platforms) is simplifying the framework selection landscape. Choose frameworks based on your deployment target and existing infrastructure rather than chasing the newest option.

The coding agent landscape has stratified into IDE assistants, CLI agents, and autonomous agents. Design workflows that assign work to the right tier based on task characteristics and risk profile.

Progressive autonomy is the standard deployment model. Start supervised, measure performance, and expand autonomy incrementally based on evidence.

Observability and evaluation are becoming as important as agent capability. Invest in tracing, cost attribution, and scenario-based evaluation alongside agent development.

Governance and safety automation will shape deployment eligibility. Build policy controls, audit trails, and least-privilege defaults early to satisfy regulatory expectations.

Local-first personal AI assistants (OpenClaw, Letta, LettaBot) are validating enterprise agentic patterns at consumer scale, while surfacing security and data sovereignty challenges that affect the whole field.

Open questions around autonomy limits, standard convergence, security, and developer roles will shape the field over the next two to three years. Stay informed and maintain architectural flexibility.

Bibliography

- GitHub Agentic Workflows documentation. https://github.github.io/gh-aw/. Accessed: 2026-02-05.
- GitHub Agentic Workflows repository. https://github.com/github/gh-aw. Accessed: 2026-02-05.
- GitHub Copilot documentation. https://docs.github.com/en/copilot. Accessed: 2026-02-05.
- GitHub Copilot coding agent. GitHub Docs. Accessed: 2026-02-05.
- Copilot coding agent environment customization. GitHub Docs. Accessed: 2026-02-05.
- Model Context Protocol (MCP). https://modelcontextprotocol.io/. Accessed: 2026-02-05.
- MCP Apps announcement. http://blog.modelcontextprotocol.io/posts/2026-01-26-mcp-apps/. Accessed: 2026-02-06.
- Agent Skills specification. https://agentskills.io/specification. Accessed: 2026-02-06.
- Agent Skills overview. https://agentskills.io/home. Accessed: 2026-02-06.
- Skills Protocol documentation. https://skillsprotocol.com/. Accessed: 2026-02-05.
- Skills Protocol Implementation Guide. https://skillsprotocol.com/implementation-guide. Accessed: 2026-02-05.
- Skills Protocol specification. https://skillsprotocol.com/specification. Accessed: 2026-02-05.
- Agent Skills format: Skill structure. https://skillsprotocol.com/skill-structure. Accessed: 2026-02-05.
- Agent Skills format: Skill manifest. https://skillsprotocol.com/skill-manifest. Accessed: 2026-02-05.
- OpenAI. https://openai.com/. Accessed: 2026-02-05.
- Anthropic. https://www.anthropic.com/. Accessed: 2026-02-05.
- OpenAI Codex overview. https://openai.com/index/introducing-codex/. Accessed: 2026-02-05.
- OpenAI Codex documentation. https://developers.openai.com/codex. Accessed: 2026-02-06.
- OpenAI GPT-5.3-Codex announcement. https://openai.com/index/introducing-gpt-5-3-codex/. Accessed: 2026-02-06.
- OpenAI Agents SDK (Python). https://openai.github.io/openai-agents-python/. Accessed: 2026-02-06.
- OpenAI Agents SDK (TypeScript). https://openai.github.io/openai-agents-js/. Accessed: 2026-02-06.
- Claude Code documentation. https://code.claude.com/docs. Accessed: 2026-02-05.
- Claude and Codex available on GitHub (public preview). https://github.blog/changelog/2026-02-04-claude-and-codex-are-now-available-in-public-preview-on-github/. Accessed: 2026-02-06.
- Cursor editor. https://www.cursor.com/. Accessed: 2026-02-05.
- CodeGPT. https://codegpt.co/. Accessed: 2026-02-05.
- Aider: AI pair programming in your terminal. https://aider.chat/. Accessed: 2026-02-06.
- Devin by Cognition. https://devin.ai/. Accessed: 2026-02-06.
- Windsurf (formerly Codeium). https://windsurf.com/. Accessed: 2026-02-06.
- LangChain documentation. https://docs.langchain.com. Accessed: 2026-02-06.
- LangGraph documentation. https://langchain-ai.github.io/langgraph/. Accessed: 2026-02-05.
- CrewAI documentation. https://docs.crewai.com/. Accessed: 2026-02-05.
- Microsoft Semantic Kernel. https://learn.microsoft.com/semantic-kernel/. Accessed: 2026-02-05.
- AutoGen documentation (v0.4). https://microsoft.github.io/autogen/stable/. Accessed: 2026-02-06.
- AutoGen v0.2 to v0.4 migration guide. https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/migration-guide.html. Accessed: 2026-02-06.
- Google Agent Development Kit (ADK). https://google.github.io/adk-docs/. Accessed: 2026-02-06.
- Agent-to-Agent (A2A) protocol. https://a2a-protocol.org/latest/. Accessed: 2026-02-06.

- A2A protocol GitHub repository. https://github.com/a2aproject/A2A. Accessed: 2026-02-06.
- Amazon Bedrock Agents. https://aws.amazon.com/bedrock/agents/. Accessed: 2026-02-06.
- Vercel AI SDK. https://ai-sdk.dev/. Accessed: 2026-02-06.
- OpenClaw. https://openclaw.ai/. Accessed: 2026-02-06.
- OpenClaw GitHub repository. https://github.com/openclaw/openclaw. Accessed: 2026-02-06.
- pi-mono agent toolkit. https://github.com/badlogic/pi-mono. Accessed: 2026-02-06.
- Letta (formerly MemGPT). https://www.letta.com/. Accessed: 2026-02-06.
- Letta GitHub repository. https://github.com/letta-ai/letta. Accessed: 2026-02-06.
- LettaBot: multi-channel personal AI assistant. https://github.com/letta-ai/lettabot. Accessed: 2026-02-06.
- Langroid. https://langroid.github.io/langroid/. Accessed: 2026-02-06.
- Open Interpreter. https://github.com/openinterpreter/open-interpreter. Accessed: 2026-02-06.
- Leon: open-source personal assistant. https://getleon.ai/. Accessed: 2026-02-06.
- Ollama. https://ollama.com/. Accessed: 2026-02-05.
- Tailscale. https://tailscale.com/. Accessed: 2026-02-05.
- Lean documentation. https://lean-lang.org/learn/. Accessed: 2026-02-05.
- Coq. https://coq.inria.fr/. Accessed: 2026-02-05.
- Isabelle. https://isabelle.in.tum.de/. Accessed: 2026-02-05.