Chapter 8: Agents for Mathematics and Physics

Contents

Chapter 1

Agentic Workflows: A Practical Guide

1.1 About This Book

This book is a living document that explores the fascinating world of agentic workflows, agent orchestration, and agentic scaffolding. It demonstrates modern AI-powered development practices by being self-maintaining - the book updates itself based on community suggestions and contributions through automated workflows.

1.2 What You'll Learn

- **Agentic Workflows**: Understanding how AI agents can automate complex tasks
- **Agent Orchestration**: Coordinating multiple agents to work together effectively
- **Agentic Scaffolding**: Building the infrastructure for agent-driven development
- **Skills and Tools**: How to use, import, and compose agent capabilities

1.3 Structure

1. Introduction to Agentic Workflows
2. Agent Orchestration
3. Agentic Scaffolding
4. Skills and Tools Management
5. GitHub Agentic Workflows (GH-AW)

1.4 Contributing

This book welcomes contributions! Open an issue with suggestions, and our automated workflows will process and integrate valuable feedback.

1.5 License

This work is open source and available for educational purposes.

Chapter 2

Chapter 1: Introduction to Agentic Workflows

2.1 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.

2.1.1 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 - Interact with development tools and APIs

Workflow: A sequence of operations orchestrated to accomplish a complex task. Agentic workflows differ from traditional workflows by being: - **Adaptive**: Agents can modify their approach based on feedback - **Goal-oriented**: Focus on outcomes rather than rigid procedures - **Context-aware**: Understanding the broader context of their actions

2.1.2 Why Agentic Workflows?

Traditional automation has limitations: - **Rigid**: Predefined steps that can't adapt to unexpected situations - **Fragile**: Breaking when conditions change - **Limited scope**: Only handling well-defined, narrow tasks

Agentic workflows solve these problems by: 1. **Flexibility**: Adapting to changing requirements and conditions 2. **Intelligence**: Understanding intent and making informed decisions 3. **Scalability**: Handling increasingly complex tasks through composition

2.2 Real-World Applications

2.2.1 Software Development

- Automated code reviews and improvements
- Bug fixing and testing
- Documentation generation and updates
- Dependency management

2.2.2 Content Management

- Self-updating documentation (like this book!)
- Blog post generation and curation
- Translation and localization

2.2.3 Operations

- Infrastructure as Code management
- Automated incident response
- Performance optimization

2.3 The Agent Development Lifecycle

1. **Define Goals**: Specify what you want to achieve
2. **Configure Agents**: Set up agents with appropriate tools and permissions
3. **Execute Workflows**: Agents work toward goals autonomously
4. **Monitor and Refine**: Review outcomes and improve agent behavior
5. **Scale**: Compose multiple agents for complex tasks

2.4 Getting Started

To work with agentic workflows, you need: - Understanding of AI/LLM capabilities and limitations - Familiarity with the problem domain - Tools and frameworks for agent development - Infrastructure for agent execution

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

2.5 Key Takeaways

- Agentic workflows enable flexible, intelligent automation
- Agents are autonomous, goal-oriented entities
- This approach scales better than traditional automation
- Understanding the fundamentals is crucial for effective implementation

Chapter 3

Chapter 2: Agent Orchestration

3.1 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.

3.2 Orchestration Patterns

3.2.1 Sequential Execution

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

```
Agent A → Agent B → Agent C → Result
```

Use cases: - Code generation → Testing → Deployment - Data collection → Analysis → Reporting

3.2.2 Parallel Execution

Multiple agents work simultaneously on independent tasks.

```
Agent A
Agent B → Aggregator → Result
Agent C
```

Use cases: - Multiple code reviews happening concurrently - Parallel data processing pipelines

3.2.3 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 - Multi-stage testing and validation

3.2.4 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 - Automated issue management - Self-updating systems (like this book!)

3.3 Coordination Mechanisms

3.3.1 Message Passing

Agents communicate through messages containing: - Task descriptions - Context and data - Results and feedback

3.3.2 Shared State

Agents access common data stores: - Database - File system - Message queues - APIs

3.3.3 Direct Invocation

Agents directly call other agents: - Function calls - API requests - Workflow triggers

3.4 Best Practices

3.4.1 1. Clear Responsibilities

Define what each agent is responsible for:

```yaml
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]
```

3.4.2 2. Error Handling

Agents should handle failures gracefully: - Retry logic for transient failures - Fallback strategies - Clear error reporting - Rollback capabilities

3.4.3 3. Monitoring

Track agent performance: - Execution time - Success/failure rates - Resource usage - Output quality

3.4.4 4. Isolation

Keep agents independent: - Minimize shared dependencies - Use clear interfaces - Version agent capabilities - Test agents independently

3.5 Orchestration Frameworks

3.5.1 GitHub Actions

Workflow orchestration for GitHub repositories:

```yaml
name: Agent Workflow
on: [push, pull_request]
jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Code Review Agent
        run: ./agents/review.sh
```

3.5.2 LangChain

Python framework for LLM applications:

```python
from langchain.agents import AgentExecutor
from langchain.agents import create_agent

agent = create_agent(
    llm=llm,
    tools=tools,
    prompt=prompt
)

executor = AgentExecutor(agent=agent, tools=tools)
result = executor.run("Your task here")
```

3.5.3 Custom Orchestration

Build your own orchestrator:

```python
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
```

3.6 Real-World Example: Self-Updating Documentation

This book uses agent orchestration:

1. **Issue Monitor Agent**: Watches for new issues with suggestions
2. **Analysis Agent**: Determines if suggestion fits the book's scope
3. **Content Agent**: Writes or updates content
4. **Build Agent**: Generates markdown and PDF versions
5. **Publishing Agent**: Updates GitHub Pages
6. **Blog Agent**: Creates blog post about the update

All coordinated through GitHub Actions workflows!

3.7 Challenges and Solutions

3.7.1 Challenge: Agent Conflicts

Solution: Use locks, transactions, or coordinator patterns

3.7.2 Challenge: Debugging

Solution: Comprehensive logging, replay capabilities, visualization

3.7.3 Challenge: Performance

Solution: Caching, parallel execution, resource limits

3.7.4 Challenge: Versioning

Solution: Version agents and interfaces separately

3.8 Key Takeaways

- Orchestration coordinates multiple agents effectively
- Choose the right pattern for your use case
- Clear responsibilities and interfaces are essential
- Monitor and iterate on your orchestration strategies
- Use established frameworks when possible, but be ready to customize

Chapter 4

Chapter 3: Agentic Scaffolding

4.1 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.

4.2 Core Components

4.2.1 1. Tool Access Layer

Agents need controlled access to tools and APIs.

```python
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}")
```

4.2.2 2. Context Management

Maintain and share context between agent invocations.

```python
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
```

4.2.3 3. Execution Environment

Provide safe, isolated environments for agent execution.

```dockerfile
# Docker-based agent environment
FROM python:3.9-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"]
```

4.2.4 4. Communication Protocol

Standardize how agents communicate.

```typescript
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
  }
}
```

4.3 Scaffolding Patterns

4.3.1 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)
```

4.3.2 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']
```

4.3.3 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")
```

4.3.4 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)
```

4.4 Building Scaffolding: Step by Step

4.4.1 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]
```

4.4.2 Step 2: Implement Tool Registry

Centralize tool access and management.

4.4.3 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']]
```

4.4.4 Step 4: Implement Error Recovery

Build resilience into your scaffolding.

```python
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
```

4.5 Scaffolding for This Book

This book's scaffolding includes:

1. **GitHub Actions**: Workflow orchestration
2. **Issue Templates**: Structured input for suggestions
3. **Agent Scripts**: Python scripts for content management
4. **Tool Access**: Git, markdown processors, PDF generators
5. **State Management**: Git repository as persistent state
6. **Communication**: GitHub API for coordination

4.6 Best Practices

1. **Start Simple**: Build minimal scaffolding first, expand as needed
2. **Security First**: Implement permissions and isolation from the start
3. **Observability**: Log everything, you'll need it for debugging
4. **Version Control**: Version your scaffolding alongside your agents
5. **Documentation**: Document tools, APIs, and patterns clearly
6. **Testing**: Test your scaffolding independently of agents

4.7 Common Pitfalls

Over-engineering: Don't build scaffolding for hypothetical needs **Tight Coupling**: Keep agents loosely coupled to scaffolding **Poor Error Handling**: Always plan for failure scenarios **No Monitoring**: You can't improve what you can't measure **Ignoring Security**: Security must be built in, not bolted on

4.8 Key Takeaways

- Scaffolding provides the foundation for effective agent operation
- Core components include tools, context, environment, and communication
- Patterns like composition and resource management improve scalability

- Build incrementally, focusing on security and observability
- Good scaffolding makes agents more capable and easier to manage

Chapter 5

Chapter 4: Skills and Tools Management

5.1 Understanding Skills vs. Tools

5.1.1 Tools

Tools are atomic capabilities that agents can use to interact with their environment. They are the building blocks of agent functionality.

Examples: - File system operations (read, write, delete) - API calls (GET, POST, PUT, DELETE) - Shell commands - Database queries

5.1.2 Skills

Skills are higher-level capabilities composed of multiple tools and logic. They represent complex behaviors that agents can learn and apply.

Examples: - Code review (using git, static analysis, test execution) - Documentation writing (using research, markdown editing, validation) - Bug fixing (using debugging, testing, code editing)

5.2 Tool Design Principles

5.2.1 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"""
```

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

5.2.2 2. Clear Interfaces

Tools should have well-defined inputs and outputs.

```python
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"""
        ...
```

5.2.3 3. Error Handling

Tools must handle errors gracefully and provide useful feedback.

```python
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'
            }
```

5.2.4 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
```

5.3 Creating Custom Tools

5.3.1 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:
```

```python
        """
        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': {}
        }
```

5.3.2 Example: Markdown Validation Tool

```python
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
```

5.4 Skill Development

5.4.1 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
```

5.4.2 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)
```

5.5 Importing and Using Skills

5.5.1 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())
```

```python
    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})
```

5.5.2 Skill Composition

```python
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
        }
```

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

5.6 Tool Discovery and Documentation

5.6.1 Self-Documenting Tools

```python
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
```

5.7 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.

5.8 Case Study: OpenClaw and pi-mono

OpenClaw is a personal, local-first AI assistant that runs a gateway control plane and connects to many chat providers and device surfaces. It emphasizes multi-channel inboxes, tool access, and skill management inside a user-owned runtime.

OpenClaw is built on the **pi-mono** ecosystem. The pi-mono monorepo provides an agent runtime, tool calling infrastructure, and multi-provider LLM APIs that OpenClaw can leverage to keep the assistant portable across models and deployments.

5.8.1 OpenClaw Architecture in Detail

OpenClaw's architecture consists of several interconnected components:

```
                                +--------------------+
+------------+                  |     Control UI     |
| WhatsApp   |---(Gateway WS)---> |     (Dashboard)    |
| Telegram   |                  +---------+----------+
| Discord    |---(API/WS/RPC)             |
| iMessage   |                      +------v------+
+------------+                      |   Gateway   |
                                        |
                                        |
                                +----Agent Runtime---+
                                |   (pi-mono core)   |
                                +--------------------+
                                  |     |     |    ...
                               [Skills/Tools] [Plugins/Other Agents]
```

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 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) - 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 for sharing and discovery

5.8.2 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 and local inference
3. **Proactive Behavior**: "Heartbeat" feature enables autonomous wake-up and environment monitoring
4. **Persistent Memory**: Learns and adapts over long-term interactions

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.

5.9 Related Architectures and Frameworks

Several other frameworks share architectural patterns with OpenClaw:

5.9.1 LangChain and LangGraph

LangChain provides composable building blocks for LLM applications:

```python
from langchain.agents import AgentExecutor, create_tool_calling_agent
from langchain_core.tools import tool

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

# Create agent with tools
agent = create_tool_calling_agent(llm, tools=[search_documentation], prompt=prompt)
executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

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

5.9.2 CrewAI

CrewAI focuses on multi-agent collaboration with role-based specialization:

```python
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.

5.9.3 Microsoft Semantic Kernel

Semantic Kernel emphasizes enterprise integration and plugin architecture:

```
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.

5.9.4 AutoGen

AutoGen specializes in conversational multi-agent systems:

```
from autogen import AssistantAgent, UserProxyAgent

assistant = AssistantAgent(
    name="coding_assistant",
    llm_config={"model": "gpt-4"},
    system_message="You are a helpful coding assistant."
)

user_proxy = UserProxyAgent(
    name="user",
    human_input_mode="NEVER",
    code_execution_config={"work_dir": "coding"}
)

# Agents collaborate through conversation
user_proxy.initiate_chat(assistant, message="Create a Python web scraper")
```

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

5.9.5 Comparing Architecture Patterns

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

5.10 MCP: Modern Tooling and Adoption

The **Model Context Protocol (MCP)** has become a practical standard for connecting agents to tools and data sources. 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.

5.10.1 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.

5.10.2 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).

5.10.3 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**
  Codex clients commonly use MCP servers to standardize tool access (repo browsing, test execution, task automation). MCP reduces per-project wiring by centralizing tool definitions and auth. The main adoption pattern is organization-level MCP servers that provide consistent tools across multiple repos.
- **GitHub 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**
  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.

5.10.4 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.

5.11 Best Practices

5.11.1 1. Version Tools and Skills

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

5.11.2 2. Test Independently

```python
# 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'])
```

5.11.3 3. Provide Fallbacks

```python
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)
```

5.11.4 4. Monitor Usage

```python
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
```

5.12 Emerging Standards: AGENTS.md

5.12.1 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.

5.12.1.1 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

5.12.1.2 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.

5.12.1.3 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
```

5.12.2 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.

5.13 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.

5.13.1 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

5.13.2 Mechanisms for Import Discovery

Modern coding agents use multiple strategies to understand imports:

5.13.2.1 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
```

5.13.2.2 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]
```

5.13.2.3 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
```

5.13.2.4 4. Package Manifest Analysis

Agents check package manifests to know what's available:

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

```python
    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
```

5.13.3 Best Practices for Import-Aware Agents

5.13.3.1 1. Document Import Conventions in AGENTS.md

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

~~~markdown
## 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
~~~

5.13.4 Alias Conventions

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

~~~~
#### 2. 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
~~~~

5.13.4.1 3. Validate Imports Before Committing

Add import validation to agent workflows:

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

```yaml
jobs:
  check-imports:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - 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' .
```

5.13.5 Import Awareness in Multi-Agent Systems

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

```python
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.

5.14 Key Takeaways

- Tools are atomic capabilities; skills are composed behaviors
- Design tools with single responsibility and clear interfaces
- Skills orchestrate multiple tools to accomplish complex tasks
- Use registries for discovery and management
- Skills can be composed to create more powerful capabilities
- Always document, test, and version your tools and skills
- Monitor usage to identify issues and optimization opportunities
- AGENTS.md is the emerging standard for project-level agent instructions
- Import awareness requires combining static analysis, LSP, and project configuration
- OpenClaw, LangChain, CrewAI, and similar frameworks share common patterns for tool and skill management

Chapter 6

Chapter 5: GitHub Agentic Workflows (GH-AW)

6.1 Why GH-AW Matters

GitHub Agentic Workflows (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:

- **Natural language workflows**: Markdown instructions drive the agent's behavior.
- **Compile-time structure**: Markdown is compiled into GitHub Actions workflows for reproducibility.
- **Security boundaries**: Permissions, tools, and safe outputs define what the agent can and cannot do.
- **Composable automation**: Imports and shared components enable reuse across repositories.

6.2 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:

1. **Frontmatter**

    - `on`: GitHub Actions triggers (issues, schedules, dispatch, etc.)
    - `permissions`: least-privilege access to GitHub APIs
    - `tools`: the capabilities your agent can invoke (edit, bash, web, github)
    - `engine`: AI model/provider (Copilot, Claude Code, Codex)
2. **Markdown instructions**
    - Natural language steps for the agent
    - Context variables from the event payload (issue number, PR, repo)

6.3 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.

6.3.1 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` 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.

6.3.2 Key Behaviors

- **Frontmatter edits require recompile**.
- **Markdown instruction updates can often be edited directly** (the runtime loads the markdown body).
- **Shared components** can be stored as markdown files without `on:`; they are imported, not compiled.

6.4 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.

6.4.1 Example 1: Issue Triage Workflow

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

```
---
on:
  issues:
    types: [opened]
permissions:
  contents: read
  issues: write
tools:
  github:
    toolsets: [issues]
```

```
engine: copilot
---

# Triage this issue
Read issue #${{ github.event.issue.number }} and summarize it.
Then add 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: write
jobs:
  agent:
    runs-on: ubuntu-latest
    steps:
      - name: Run GH-AW agent
        uses: github/agentic-workflows@v1
        with:
          engine: copilot
          tools: github:issues
          prompt: |
            # Triage this issue
            Read issue #${{ github.event.issue.number }} and summarize it.
            Then add labels: needs-triage and needs-owner.
```

What changed during compilation

- Frontmatter was converted into workflow metadata (`on`, `permissions`, `jobs`).
- `engine` and `tools` moved into the `with:` block for the GH-AW runner.
- The markdown body became the `prompt` payload executed by the agent.

6.4.2 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: write
imports:
  - shared/common-tools.md
---
```

```
# 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: write
jobs:
  agent:
    runs-on: ubuntu-latest
    steps:
      - name: Run GH-AW agent
        uses: github/agentic-workflows@v1
        with:
          engine: copilot
          tools: bash,edit
          prompt: |
            # Refresh docs
            Update the changelog with the latest release notes.
```

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.

6.4.3 Example 3: Safe Outputs in the Compiled Job

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

```
---
on:
  workflow_dispatch:
permissions:
  contents: read
  pull-requests: write
tools:
  edit:
safe_outputs:
  pull_request_body:
    format: markdown
engine: copilot
---

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

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

```
name: GH-AW release notes
on:
  workflow_dispatch:
permissions:
  contents: read
  pull-requests: write
```

```
jobs:
  agent:
    runs-on: ubuntu-latest
    steps:
      - name: Run GH-AW agent
        uses: github/agentic-workflows@v1
        with:
          engine: copilot
          tools: edit
          safe-outputs:
            pull_request_body:
              format: markdown
          prompt: |
            # Draft release notes
            Summarize commits since the last tag and open a PR with the notes.
```

What changed during compilation

- `safe_outputs` became `safe-outputs` input for the GH-AW runner.
- The prompt stayed identical; guardrails are enforced by the compiled job.

6.5 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.

6.5.1 Tools

Tools are capabilities the agent can use:

- **edit**: modify files in the workspace
- **bash**: run shell commands (by default only safe commands)
- **web-fetch / web-search**: fetch or search web content
- **github**: operate on issues, PRs, discussions, projects
- **playwright**: browser automation for UI checks

6.5.2 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.

6.5.3 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.

6.6 Imports and Reusable Components

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 %}`

Imports let you create a library of reusable workflow fragments (shared tools, standard prompts, or organization-wide policies). Files without `on:` become *components* that can be imported without being compiled into Actions.

6.7 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 repo or ecosystem (new libraries, frameworks, scaffolds). - It produces a report in an issue or discussion.

Phase 2: Plan - Maintainers review the report and decide whether to proceed. - A planning agent drafts the implementation steps if approved.

Phase 3: Assign & Implement - Agents are assigned to implement the approved changes. - Updates are validated, committed, and published.

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.

6.8 Applying GH-AW to This Repository

Here's how GH-AW can drive the book's maintenance:

1. **Research (scheduled)**
   - Use web-search tooling to scan for new agentic workflow libraries.
   - Produce a structured report in a GitHub issue.
2. **Consensus (issues/discussions)**
   - Collect votes or comments to accept/reject the proposal.
   - Label outcomes (`accepted`, `needs-revision`, `rejected`).
3. **Implementation (assigned agent)**
   - Update or add chapters.
   - Refresh the table of contents and homepage.
   - Add a blog post summarizing the update.
4. **Publish (automation)**
   - Pages and PDF rebuild automatically after merge.

This approach keeps the book aligned with the latest GH-AW practices while maintaining a transparent, auditable workflow.

6.9 Key Takeaways

- GH-AW turns markdown instructions into reproducible GitHub Actions workflows.
- Frontmatter defines triggers, permissions, tools, and models.
- Imports enable composable, reusable workflow building blocks.
- Safe inputs/outputs and least-privilege permissions reduce risk.
- ResearchPlanAssign provides a practical loop for continuous, agent-powered improvement.

Chapter 7

Chapter 6: GitHub Agents

7.1 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.

7.2 The GitHub Agent Ecosystem

7.2.1 GitHub Copilot

GitHub Copilot is the foundation of GitHub's AI-powered development tools. It provides:

- **Code Completion**: Real-time suggestions as you type
- **Chat Interface**: Natural language conversations about code
- **Context Awareness**: Understanding of your codebase and intent

```python
# 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))
```

7.2.2 GitHub Copilot Coding Agent

The Coding Agent extends Copilot's capabilities to autonomous task completion:

- **Assigned Tasks**: Receives issues or requests and works independently
- **Multi-File Changes**: Can modify multiple files across a codebase
- **Pull Request Creation**: Generates complete PRs with descriptions
- **Iterative Development**: Responds to review feedback

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

7.2.3 GitHub Actions Agents

Agents can be orchestrated through GitHub Actions workflows:

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

jobs:
  agent-task:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Process with Agent
        uses: actions/github-script@v7
        with:
          script: |
            // Agent logic to analyze and respond
            const issue = context.payload.issue;
            // ... agent processing
```

7.3 Agent Capabilities

7.3.1 Reading and Understanding

Agents can read and understand:

- **Code**: Source files, configurations, dependencies
- **Documentation**: READMEs, wikis, comments
- **Issues and PRs**: Descriptions, comments, reviews
- **Repository Structure**: File organization, patterns

7.3.2 Writing and Creating

Agents can produce:

- **Code Changes**: New files, modifications, refactoring
- **Documentation**: READMEs, comments, API docs
- **Issues and Comments**: Status updates, analysis reports
- **Pull Requests**: Complete PRs with proper descriptions

7.3.3 Reasoning and Deciding

Agents can:

- **Analyze Problems**: Understand issue context and requirements
- **Plan Solutions**: Break down tasks into steps
- **Make Decisions**: Choose between approaches

- **Adapt**: Respond to feedback and changing requirements

7.4 Multi-Agent Orchestration

7.4.1 Why Multiple Agents?

Single agents have limitations. Multi-agent systems provide:

1. **Specialization**: Each agent excels at specific tasks
2. **Perspective Diversity**: Different models bring different strengths
3. **Scalability**: Parallel processing of independent tasks
4. **Resilience**: Failure of one agent doesn't stop the workflow

7.4.2 Orchestration Patterns

7.4.2.1 Sequential Pipeline

Agents work in sequence, each building on the previous:

```
Issue → ACK Agent → Research Agent → Writer Agent → Review Agent → Complete
```

Example workflow stages:

```yaml
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
```

7.4.2.2 Parallel Discussion

Multiple agents contribute perspectives simultaneously:

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

7.4.2.3 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 - Resolving ambiguous decisions - Quality assurance

7.4.3 Agent Handoff Protocol

When agents need to pass context to each other:

1. **State in Labels**: Use GitHub labels to track workflow stage
2. **Context in Comments**: Agents document their findings in issue comments
3. **Structured Output**: Use consistent formats for machine readability

```yaml
# Example: Structured agent output
- name: Agent Report
  uses: actions/github-script@v7
  with:
    script: |
      const report = {
        stage: 'research',
        findings: [...],
        recommendation: 'proceed',
        nextAgent: 'writer'
      };
      // Store in comment or labels
```

7.5 Implementing GitHub Agents

7.5.1 Agent Definition Files

Define agents in markdown files with frontmatter:

```markdown
---
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
```

7.5.2 Agent Configuration

Control agent behavior through configuration:

```yaml
# .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
```

7.5.3 Error Handling

Agents should handle failures gracefully:

```yaml
- name: Agent Task with Error Handling
  id: agent_task
  continue-on-error: true
  uses: actions/github-script@v7
  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: `  Agent encountered an error: ${error.message}`
        });
      }
```

7.6 Best Practices

7.6.1 1. Clear Agent Personas

Give each agent a clear identity and responsibility:

```markdown
## 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
```

7.6.2 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]
```

7.6.3 3. Human Checkpoints

Always include human review points:

- Before significant changes
- After agent recommendations
- Before closing issues

7.6.4 4. Audit Trail

Maintain visibility into agent actions:

- All agent actions should be visible in comments
- Use labels to track workflow state
- Log important decisions and reasoning

7.6.5 5. Graceful Degradation

Design for agent failures:

- Use `continue-on-error` for non-critical steps
- Provide manual fallback options
- Alert maintainers when intervention is needed

7.7 Security Considerations

7.7.1 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
```

7.7.2 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');
}
```

7.7.3 Output Sanitization

Sanitize agent outputs:

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

7.7.4 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
```

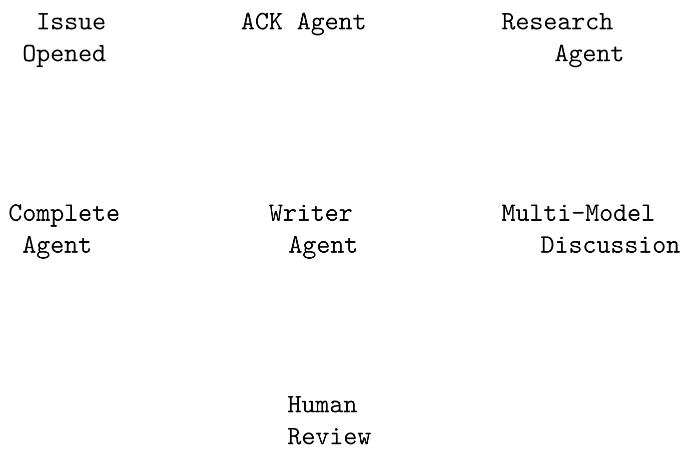
7.8 Real-World Example: This Book

This very book uses GitHub Agents for self-maintenance:

7.8.1 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

7.8.2 How It Works



7.8.3 Configuration

The workflow is defined using GitHub Agentic Workflows (GH-AW). The repository includes: - GH-AW workflows: `.github/workflows/issue-*.lock.yml` - Agent definitions: `.github/agents/*.md` - Legacy fallback: `.github/workflows/process-suggestions.yml`

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

7.9 Multi-Agent Platform Compatibility

Modern repositories need to support multiple AI agent platforms. Different coding assistants—GitHub Copilot, Claude, OpenAI 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.

7.9.1 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** uses `CLAUDE.md` for dedicated configuration, or can read `AGENTS.md` files for project context
- **Codex/ChatGPT** can be configured with custom instructions or system prompts
- **Generic agents** look for `AGENTS.md` as the emerging standard

Each platform has slightly different expectations, but the core information they need is similar.

7.9.2 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
`CONTRIBUTING.md`	Contribution guidelines	Workflow rules and constraints
`.github/copilot-instructions.md`	N/A	Copilot-specific configuration
`AGENTS.md`	N/A	Generic agent instructions
`CLAUDE.md`	N/A	Claude-specific configuration

7.9.3 The copilot-instructions.md File

GitHub Copilot reads `.github/copilot-instructions.md` to understand how to work with your repository. This file should include:

```
# Copilot Instructions for [Project Name]

## Project Overview
Brief description of what this project does.

## Tech Stack
- **Language**: Python 3.11
- **Framework**: FastAPI
- **Database**: PostgreSQL
- **Testing**: pytest

## Coding Guidelines
```

```
- Follow PEP 8 style guide
- All functions require type hints
- Tests are required for new features

## File Structure
Describe important directories and their purposes.

## Build and Test Commands
- `make test` - Run all tests
- `make lint` - Run linters
- `make build` - Build the project

## Important Constraints
- Never commit secrets or credentials
- Protected paths: `.github/workflows/`, `secrets/`
- All PRs require review before merge
```

7.9.4 Cross-Platform Strategy

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

1. **Use AGENTS.md as the canonical source** for project instructions
2. **Create platform-specific files** that import or reference AGENTS.md content
3. **Keep instructions DRY** by avoiding duplication across files
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
```

7.9.5 This Repository's Approach

This book repository demonstrates multi-platform compatibility:

- **`.github/copilot-instructions.md`** - Detailed Copilot configuration with project structure, coding guidelines, and constraints
- **Chapters 4 and 7** discuss AGENTS.md as the emerging standard
- **Documentation files** (README, CONTRIBUTING, etc.) provide context any agent can use
- **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're simultaneously creating better agent configuration.

7.9.6 Best Practices for Agent-Friendly Repositories

1. **Be explicit about constraints**: Clearly state what agents should NOT do
2. **Document your tech stack**: Agents perform better when they understand the tools in use
3. **Describe the project structure**: Help agents navigate your codebase efficiently
4. **Provide examples**: Show preferred patterns through code examples
5. **List protected paths**: Specify files agents should not modify

6. **Include build/test commands**: Enable agents to verify their changes
7. **State coding conventions**: Help agents write consistent code

7.10 Future of GitHub Agents

7.10.1 Emerging Capabilities

- **Code Generation**: Agents writing production-quality code
- **Test Authoring**: Automatic test creation and maintenance
- **Documentation Sync**: Keeping docs in sync with code
- **Security Analysis**: Proactive vulnerability detection

7.10.2 Integration Trends

- **IDE Integration**: Deeper VS Code and editor integration
- **CI/CD Native**: Agents as first-class CI/CD citizens
- **Cross-Repo**: Agents working across multiple repositories
- **Multi-Cloud**: Agents coordinating across platforms

7.11 Key Takeaways

1. **GitHub Agents** are AI-powered assistants that can reason, decide, and act within repositories.
2. **Copilot Coding Agent** can autonomously complete tasks and create pull requests.
3. **Multi-agent orchestration** enables specialized, resilient, and scalable automation.
4. **Human checkpoints** remain essential for quality and safety.
5. **Clear protocols** for agent communication ensure smooth handoffs.
6. **Security** must be designed into agent workflows from the start.
7. **Multi-platform compatibility** is achieved through well-structured documentation (copilot-instructions.md, AGENTS.md, etc.).
8. **This book** demonstrates these concepts through its own multi-agent maintenance workflow.

7.12 Learn More

7.12.1 Repository Documentation

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

- **README** - Overview and quick start guide
- **CONTRIBUTING** - How to contribute using the multi-agent workflow
- **WORKFLOWS** - Detailed workflow guide and GH-AW explanation
- **SETUP** - Installation and configuration instructions
- **WORKFLOW_PLAYBOOK** - Agentic workflow maintenance patterns
- **PROJECT_SUMMARY** - Complete project overview
- **SECURITY_SUMMARY** - Security practices and scan results
- **CHANGELOG** - Version history and changes
- **CODE_OF_CONDUCT** - Community guidelines
- **LICENSE** - MIT License

7.12.2 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.

7.12.3 Related Chapters

- **Chapter 4: Skills and Tools** - Covers AGENTS.md standard and MCP protocol for tool management
- **Chapter 5: GitHub Agentic Workflows** - GH-AW specification and engine configuration
- **Chapter 7: Agents for Coding** - Detailed coverage of coding agent platforms

---

Chapter 8

Chapter 7: Agents for Coding

8.1 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.

8.2 The Evolution of Coding Agents

8.2.1 From Autocomplete to Autonomy

The progression of coding agents follows a clear trajectory:

1. **Code Completion (2020-2022)**: Basic pattern matching and next-token prediction
2. **Context-Aware Assistance (2022-2024)**: Understanding project structure and intent
3. **Task-Oriented Agents (2024-present)**: Completing multi-step tasks independently
4. **Autonomous Development (emerging)**: Full feature implementation, testing, and deployment

8.2.2 Current Capabilities

Modern coding agents can:

- **Understand Requirements**: Parse natural language specifications and translate them to code
- **Plan Solutions**: Break down complex features into implementable steps
- **Generate Code**: Write production-quality code across multiple files
- **Test and Debug**: Create tests, identify bugs, and fix issues
- **Scaffold Projects**: Initialize projects with appropriate structure and configuration
- **Review and Refactor**: Analyze code quality and suggest improvements

8.3 Specialized Architectures

8.3.1 Single-Agent Architectures

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

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

```python
    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.

8.3.2 Multi-Agent Architectures

Complex projects benefit from specialized agents working together.

```python
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.

8.3.3 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)
```

8.4 Scaffolding for Coding Agents

8.4.1 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
```

8.4.2 The AGENTS.md Standard

The **AGENTS.md** file has emerged as the de facto standard for providing AI coding agents with project-specific instructions. It serves as a "README for agents," offering structured, machine-readable guidance.

8.4.2.1 Purpose and Placement

```
project/
   AGENTS.md           # Root-level agent instructions
   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.

8.4.2.2 Standard Structure

```markdown
# AGENTS.md

## Project Overview
Brief description of the project and its purpose.

## Setup Instructions
How to install dependencies and configure the environment.

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

## Testing Requirements
- Framework: Jest
- Coverage threshold: 80%
- Required test types: unit, integration

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

## Agent-Specific Notes
- Always run linting before committing
- Never modify files in `vendor/`
- Secrets are in environment variables, never hardcoded
```

8.4.2.3 Benefits

1. **Consistency**: All agents receive the same instructions
2. **Onboarding**: New agents (or new agent sessions) understand the project immediately
3. **Safety**: Clear boundaries prevent accidental damage
4. **Maintainability**: Single source of truth for agent behavior

8.4.3 Context Management

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

```python
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)
```

8.4.4 Tool Registries

Coding agents need well-organized tool access:

```python
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]
```

8.5 Leading Coding Agent Platforms

8.5.1 GitHub Copilot and Coding Agent

GitHub Copilot has evolved from an IDE autocomplete tool to a full coding agent:

- **Copilot Chat**: Natural language interaction about code
- **Copilot Coding Agent**: Autonomous task completion and PR creation
- **Copilot Workspace**: Full development environment with agent integration

```yaml
# Example: Using Copilot Coding Agent via GitHub Actions
name: Copilot Task
on:
  issues:
    types: [labeled]

jobs:
  copilot-task:
    if: contains(github.event.issue.labels.*.name, 'copilot')
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Assign to Copilot
        uses: github/copilot-agent@v1
        with:
          issue: ${{ github.event.issue.number }}
          permissions: read-write
```

8.5.2 Claude Code

Claude Code provides multi-agent orchestration for complex development tasks:

- **Subagent Architecture**: Spawn specialized agents for different concerns
- **Swarms Mode**: Parallel execution of independent tasks
- **Extended Context**: Handle large codebases through intelligent context management

8.5.3 Cursor AI

Cursor is an AI-first code editor designed around agent workflows:

- **Project-Wide Understanding**: Indexes entire codebase for context
- **Multi-File Generation**: Creates and modifies multiple files in one operation
- **Framework Integration**: Deep understanding of popular frameworks

8.5.4 CodeGPT and Agent Marketplaces

Marketplace-based approaches offer specialized agents:

- **Specialized Agents**: Over 200 pre-built agents for specific tasks
- **Custom Agent Creation**: Build and share domain-specific agents
- **Multi-Model Support**: Combine different LLMs for different tasks

8.6 Best Practices

8.6.1 1. Clear Task Boundaries

Define clear boundaries for what agents can and cannot do:

```python
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
```

8.6.2 2. Incremental Changes

Prefer small, focused changes over large rewrites:

```python
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
```

8.6.3 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
```

8.6.4 4. Human Review Integration

Always include human checkpoints for significant changes:

```yaml
# 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@v4

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

      - name: Create PR for Review
        uses: peter-evans/create-pull-request@v5
        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
```

8.7 Common Challenges

8.7.1 Context Window Limitations

Large codebases exceed agent context windows:

Solution: Implement intelligent context retrieval and summarization.

```python
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)
```

8.7.2 Hallucination and Accuracy

Agents may generate plausible but incorrect code:

Solution: Implement validation and testing at every step.

8.7.3 Security Concerns

Agents with code access pose security risks:

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

```python
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
```

8.8 Key Takeaways

1. **Coding agents** have evolved from autocomplete to autonomous development assistants.

2. **Multi-agent architectures** mirror development teams, with specialized agents for architecture, implementation, testing, and review.
3. **AGENTS.md** is the emerging standard for providing agents with project-specific instructions.
4. **Scaffolding** for coding agents includes context management, tool registries, and security boundaries.
5. **Human review** remains essential—agents create PRs for review, not direct commits.
6. **Incremental changes** with continuous validation are safer than large rewrites.
7. **Security** must be designed in from the start: sandboxing, permissions, and audit logging.

Chapter 9

Chapter 8: Agents for Mathematics and Physics

9.1 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.

9.2 The Landscape of Scientific Agents

9.2.1 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

9.2.2 Categories of Scientific Agents

1. **Theorem Proving Agents**: Construct formal proofs in systems like Lean, Coq, or Isabelle
2. **Symbolic Computation Agents**: Work with computer algebra systems (CAS)
3. **Numerical Simulation Agents**: Set up and run physics simulations
4. **Research Assistants**: Search literature, summarize findings, identify gaps
5. **Educational Scaffolding Agents**: Help students learn mathematical and physical concepts

9.3 Theorem Proving Agents

9.3.1 Formal Verification Background

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

9.3.2 Ax-Prover Architecture

Ax-Prover is a notable deep reasoning framework for theorem proving in mathematics and quantum physics. It demonstrates how multi-agent orchestration can tackle formal proofs:

```python
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)
```

9.3.3 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)
```

9.3.4 Challenges in Theorem Proving

1. **Search Space Explosion**: Proofs can have many possible paths
2. **Creativity Required**: Non-obvious proof strategies
3. **Formalization Gap**: Translating informal to formal
4. **Domain Knowledge**: Deep mathematical understanding needed

9.4 Symbolic Computation Agents

9.4.1 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
```

```python
        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)
```

9.4.2 Combining Symbolic and Neural Approaches

Modern agents combine symbolic precision with neural flexibility:

```python
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)
        }
```

9.5 Physics Simulation Agents

9.5.1 Computational Physics Workflows

Physics agents orchestrate simulation workflows:

```python
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")
```

9.5.2 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
```

```python
        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)
```

9.6 Scaffolding for Scientific Agents

9.6.1 Tool Integration Layer

Scientific agents need access to specialized tools:

```yaml
# 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
```

9.6.2 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
```

9.6.3 Verification Pipeline

All scientific agent outputs should be verified:

```python
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
        )
```

9.7 Educational Scaffolding Agents

9.7.1 Mathematics Education

AI agents are transforming mathematics education:

```python
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)
```

9.7.2 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)
```

9.8 Research Agent Workflows

9.8.1 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
```

9.9 Best Practices

9.9.1 1. Rigorous Verification

Always verify scientific outputs:

```python
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
```

9.9.2 2. Uncertainty Quantification

Scientific agents should express uncertainty:

```python
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)
        }
```

9.9.3 3. Reproducibility

Ensure all computations are reproducible:

```python
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()
        }
```

9.9.4 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
```

9.10 Key Takeaways

1. **Scientific agents** require formal verification and rigorous validation beyond what coding agents need.
2. **Theorem proving agents** combine LLM creativity with proof assistant verification for mathematical rigor.
3. **Symbolic computation** and neural approaches are complementary—use both for best results.
4. **Physics agents** must respect conservation laws, dimensional consistency, and physical constraints.
5. **Educational scaffolding** agents adapt explanations to student level and address misconceptions.

6. **Verification pipelines** should check proofs, dimensions, and compare with known results.
7. **Reproducibility** is essential—record seeds, versions, and all parameters.
8. **Collaboration** with domain experts improves agent reliability and trustworthiness.