

Open Code Agents: Complete Guide to AI-Powered Development Workflows

Open Code Agents Project

Contents

1	Chapter 1: Introduction to Open Code Agents	7
1.1	What Are Open Code Agents?	7
1.2	The Problem: Managing Complex Development Workflows	7
1.3	The Solution: Specialized AI Personas	8
1.4	Benefits of Agent-Based Development	8
1.4.1	1. Focused Expertise	8
1.4.2	2. Workflow Orchestration	8
1.4.3	3. Consistency and Quality	8
1.4.4	4. Scalability	9
1.5	Who This Book Is For	9
1.5.1	Developers	9
1.5.2	Team Leads	9
1.5.3	DevOps Engineers	9
1.5.4	Technical Managers	9
1.6	How to Use This Book	9
1.6.1	For Beginners	9
1.6.2	For Practitioners	9
1.6.3	For Advanced Users	9
1.6.4	Reference Material	10
1.7	The Journey Ahead	10
1.8	Getting Started	10
2	Chapter 2: Quick Start Guide	10
2.1	Installation Methods	10
2.1.1	Method 1: Build from Source	10
2.1.2	Method 2: Download Pre-built Binary	11
2.2	Basic Commands and Usage	11
2.2.1	Exploring Available Options	11
2.2.2	Your First Agent Workflow	11
2.3	Understanding the CLI Interface	12
2.3.1	Command Structure	12
2.3.2	Common Options	12

2.4	Setting Up Your Environment	13
2.4.1	Configuration Files	13
2.4.2	Environment Variables	13
2.5	Your First Custom Agent	14
2.5.1	Example: Code Review	14
2.5.2	Example: Bug Investigation	14
2.6	Tips for Success	14
2.6.1	1. Start Small	14
2.6.2	2. Provide Clear Context	14
2.6.3	3. Review Agent Outputs	14
2.6.4	4. Use Version Control	14
2.6.5	5. Learn Patterns	15
2.7	Troubleshooting Common Issues	15
2.7.1	Installation Problems	15
2.7.2	Runtime Issues	15
2.8	Next Steps	15
2.9	Quick Reference	15
3	Chapter 3: System Architecture	16
3.1	Core Components	16
3.1.1	1. CLI Interface (cmd/)	16
3.1.2	2. Agent System (pkg/agent/)	16
3.1.3	3. Configuration Management (pkg/config/)	16
3.1.4	4. CLI Commands (pkg/cli/)	17
3.1.5	5. Orchestrator (pkg/orchestrator/)	17
3.2	Agent Architecture	17
3.2.1	Agent Definition Structure	17
3.2.2	Agent Execution Model	17
3.2.3	Context Management	17
3.3	Workflow Orchestration	18
3.3.1	Workflow Definition	18
3.3.2	Execution Flow	18
3.4	Data Flow Architecture	18
3.4.1	Input Processing	18
3.4.2	Agent Execution Flow	18
3.4.3	Output Generation	19
3.5	Configuration Architecture	19
3.5.1	Configuration Hierarchy	19
3.5.2	Configuration Merging	19
3.6	Integration Points	19
3.6.1	Model Context Protocol (MCP)	19
3.6.2	Embed System	19
3.6.3	Installer System	19
3.7	Error Handling Architecture	19
3.7.1	Error Types	19
3.7.2	Error Recovery Strategies	20

3.8	Performance Considerations	20
3.8.1	Caching Strategy	20
3.8.2	Resource Management	20
3.8.3	Scalability Design	20
3.9	Security Architecture	20
3.9.1	Isolation	20
3.9.2	Data Protection	20
3.10	Testing Architecture	20
3.10.1	Test Organization	20
3.10.2	Test Types	21
3.11	Future Architecture Considerations	21
3.11.1	Distributed Agents	21
3.11.2	Advanced Features	21
3.11.3	Integration Expansion	21
3.12	Summary	21
4	Chapter 4: Installation and Setup	21
4.1	System Requirements	22
4.1.1	Minimum Requirements	22
4.1.2	Recommended Requirements	22
4.1.3	Supported Platforms	22
4.2	Installation Methods	22
4.2.1	Method 1: Build from Source (Recommended for Developers)	22
4.2.2	Method 2: Download Pre-built Binaries	23
4.2.3	Method 3: Package Manager Installation	23
4.3	Configuration Setup	24
4.3.1	Initial Configuration	24
4.3.2	Manual Configuration	24
4.3.3	Environment Variables	25
4.4	Project Setup	25
4.4.1	Initializing a New Project	25
4.4.2	Project Configuration	25
4.5	Integration Setup	26
4.5.1	Git Integration	26
4.5.2	IDE Integration	27
4.5.3	CI/CD Integration	28
4.6	Verification and Testing	29
4.6.1	Basic Verification	29
4.6.2	Integration Testing	30
4.6.3	Performance Testing	30
4.7	Troubleshooting	30
4.7.1	Common Installation Issues	30
4.7.2	Configuration Issues	31
4.7.3	Performance Issues	31
4.8	Maintenance and Updates	32

4.8.1	Updating Open Code Agents	32
4.8.2	Backup Configuration	32
4.8.3	Cleanup	32
4.9	Next Steps	32
5	Chapter 5: The Architect Agent	33
5.1	Role and Responsibilities	33
5.1.1	Primary Focus Areas	33
5.1.2	Core Responsibilities	33
5.2	When to Use the Architect Agent	33
5.2.1	Ideal Scenarios	33
5.2.2	Not Ideal For	34
5.3	Architect Agent Capabilities	34
5.3.1	Design Patterns Expertise	34
5.3.2	Technology Stack Assessment	35
5.4	Using the Architect Agent	35
5.4.1	Basic Usage	35
5.4.2	Advanced Usage	36
5.5	Input and Output Examples	37
5.5.1	Input Examples	37
5.5.2	Output Examples	38
5.6	Best Practices for Architect Agent	40
5.6.1	Providing Good Input	40
5.6.2	Structuring Requirements	40
5.6.3	Reviewing Architect Output	40
5.7	Integration with Other Agents	41
5.7.1	Typical Workflow Sequence	41
5.7.2	Handoff Patterns	41
5.8	Common Use Cases	41
5.8.1	Use Case 1: Startup MVP Architecture	41
5.8.2	Use Case 2: Enterprise System Integration	41
5.8.3	Use Case 3: Performance Optimization	41
5.9	Troubleshooting	42
5.9.1	Common Issues	42
5.9.2	Getting Better Results	42
5.10	Advanced Features	42
5.10.1	Architecture Patterns Library	42
5.10.2	Technology Comparison	42
5.10.3	Cost Estimation	43
5.11	Summary	43
6	Chapter 6: The Implementer Agent	43
6.1	Role and Responsibilities	43
6.1.1	Primary Focus Areas	43
6.1.2	Core Responsibilities	43
6.2	When to Use the Implementer Agent	44

6.2.1	Ideal Scenarios	44
6.2.2	Not Ideal For	44
6.3	Implementer Agent Capabilities	44
6.3.1	Programming Languages Expertise	44
6.3.2	Framework and Library Expertise	45
6.3.3	Development Patterns and Practices	46
6.4	Using the Implementer Agent	46
6.4.1	Basic Usage	46
6.4.2	Advanced Usage	47
6.5	Input and Output Examples	49
6.5.1	Input Examples	49
6.5.2	Output Examples	50
6.6	Best Practices for Implementer Agent	57
6.6.1	Providing Good Input	57
6.6.2	Structuring Requirements	57
6.6.3	Reviewing Implementation Output	57
6.7	Integration with Other Agents	58
6.7.1	Typical Workflow Sequence	58
6.7.2	Handoff Patterns	58
6.8	Common Use Cases	58
6.8.1	Use Case 1: API Development	58
6.8.2	Use Case 2: Component Development	58
6.8.3	Use Case 3: Business Logic Implementation	58
6.9	Troubleshooting	59
6.9.1	Common Issues	59
6.9.2	Getting Better Results	59
6.10	Advanced Features	59
6.10.1	Code Generation from OpenAPI	59
6.10.2	Database Migration Generation	59
6.10.3	Performance Optimization	60
6.11	Summary	60
7	Chapter 7: The Tester Agent	60
7.1	Role and Responsibilities	60
7.1.1	Primary Focus Areas	60
7.1.2	Core Responsibilities	60
7.2	When to Use the Tester Agent	61
7.2.1	Ideal Scenarios	61
7.2.2	Not Ideal For	61
7.3	Tester Agent Capabilities	62
7.3.1	Testing Types Expertise	62
7.3.2	Testing Frameworks and Tools	62
7.3.3	Testing Strategies and Patterns	63
7.4	Using the Tester Agent	64
7.4.1	Basic Usage	64
7.4.2	Advanced Usage	64

7.5	Input and Output Examples	66
7.5.1	Input Examples	66
7.5.2	Output Examples	68
7.6	Best Practices for Tester Agent	79
7.6.1	Providing Good Input	79
7.6.2	Structuring Test Requirements	80
7.6.3	Reviewing Test Output	80
7.7	Integration with Other Agents	80
7.7.1	Typical Workflow Sequence	80
7.7.2	Handoff Patterns	81
7.8	Common Use Cases	81
7.8.1	Use Case 1: API Testing	81
7.8.2	Use Case 2: Component Testing	81
7.8.3	Use Case 3: Performance Testing	81
7.9	Troubleshooting	81
7.9.1	Common Issues	81
7.9.2	Getting Better Results	82
7.10	Advanced Features	82
7.10.1	Test Coverage Analysis	82
7.10.2	Performance Benchmarking	82
7.10.3	Test Data Generation	82
7.11	Summary	82
8	Chapter The Debugger Agent	83
8.1	Coming Soon	83
8.2	Preview	83
9	Chapter The Reviewer Agent	83
9.1	Coming Soon	83
9.2	Preview	84
10	Chapter The Refactorer Agent	84
10.1	Coming Soon	84
10.2	Preview	84
11	Chapter The Documenter Agent	84
11.1	Coming Soon	84
11.2	Preview	85
12	Chapter The Researcher Agent	85
12.1	Coming Soon	85
12.2	Preview	85
13	Chapter Workflow Orchestration	85
13.1	Coming Soon	85
13.2	Preview	86

14 Chapter Command Reference	86
14.1 Coming Soon	86
14.2 Preview	86
15 Chapter Custom Commands	86
15.1 Coming Soon	86
15.2 Preview	87
16 Chapter Integration Strategies	87
16.1 Coming Soon	87
16.2 Preview	87
17 Chapter Performance Optimization	87
17.1 Coming Soon	87
17.2 Preview	88
18 Chapter Security Considerations	88
18.1 Coming Soon	88
18.2 Preview	88
19 Chapter Quick Reference Guide	88
19.1 Coming Soon	88
19.2 Preview	89
20 Chapter Best Practices	89
20.1 Coming Soon	89
20.2 Preview	89
21 Chapter Troubleshooting	89
21.1 Coming Soon	89
21.2 Preview	90

1 Chapter 1: Introduction to Open Code Agents

1.1 What Are Open Code Agents?

Open Code Agents is a revolutionary CLI tool that brings the power of specialized AI personas to software development workflows. Imagine having a team of expert developers—each with their own specialized skills—ready to tackle any development challenge you throw at them. That’s exactly what Open Code Agents provides.

1.2 The Problem: Managing Complex Development Workflows

Modern software development is complex. A single feature might require:

- **Research** into best practices and technologies
- **Architectural design** to ensure scalability
- **Implementation** with clean, maintainable code
- **Testing** to guarantee reliability
- **Review** to maintain quality standards
- **Documentation** for future maintainers

Traditionally, developers must wear all these hats simultaneously, leading to: - Context switching and cognitive overload - Inconsistent quality across different domains - Longer development cycles - Increased risk of mistakes

1.3 The Solution: Specialized AI Personas

Open Code Agents solves this problem by providing **8 specialized AI agents**, each an expert in their domain:

Agent	Specialization	Primary Role
Architect	System Design	High-level architectural planning
Implementer	Code Development	Writing production-quality code
Tester	Quality Assurance	Creating comprehensive test suites
Debugger	Issue Resolution	Analyzing and fixing bugs
Reviewer	Code Review	Evaluating code quality
Refactorer	Code Improvement	Improving code structure
Documenter	Documentation	Creating comprehensive docs
Researcher	Investigation	Researching technologies

Each agent embodies years of expertise in their specific domain, following established best practices and workflows.

1.4 Benefits of Agent-Based Development

1.4.1 1. Focused Expertise

Each agent focuses exclusively on their domain, ensuring: - Deep understanding of best practices - Consistent application of standards - Reduced cognitive load for developers

1.4.2 2. Workflow Orchestration

Agents collaborate through structured handoffs: - Clear communication patterns - Context preservation between stages - Quality gates at each transition

1.4.3 3. Consistency and Quality

Standardized workflows ensure: - Repeatable results - Measurable quality metrics - Reduced human error

1.4.4 4. Scalability

Handle projects of any size: - Small tasks: Single agent execution - Medium features: Multi-agent workflows - Large projects: Complex orchestration patterns

1.5 Who This Book Is For

This book is designed for:

1.5.1 Developers

- Want to improve development efficiency
- Need consistent code quality
- Looking to automate repetitive tasks

1.5.2 Team Leads

- Managing development workflows
- Ensuring code quality standards
- Optimizing team productivity

1.5.3 DevOps Engineers

- Automating development pipelines
- Integrating AI tools into workflows
- Scaling development processes

1.5.4 Technical Managers

- Understanding modern development practices
- Evaluating AI-assisted development
- Planning team capabilities

1.6 How to Use This Book

1.6.1 For Beginners

Start with Chapters 1-3 to understand: - Core concepts and philosophy - Installation and setup - Basic usage patterns

1.6.2 For Practitioners

Dive into Chapters 4-12 for: - Detailed agent capabilities - Workflow patterns - Integration strategies

1.6.3 For Advanced Users

Explore Chapters 13-21 for: - Custom command creation - Advanced configuration - System integration

1.6.4 Reference Material

Use Part 5 (Chapters 19-21) as: - Quick reference for commands - Agent capability matrix - Best practices summary

1.7 The Journey Ahead

This book will take you on a comprehensive journey through the Open Code Agents ecosystem:

1. **Understanding the System:** Learn architecture and philosophy
2. **Mastering the Agents:** Deep dive into each specialized agent
3. **Orchestrating Workflows:** Combine agents for complex tasks
4. **Advanced Integration:** Extend and customize the system
5. **Reference Material:** Quick guides and best practices

By the end of this book, you'll be able to: - Choose the right agent for any task - Design efficient multi-agent workflows - Create custom commands and automation - Integrate Open Code Agents into your existing processes - Scale development workflows for teams and projects

1.8 Getting Started

Ready to begin? The next chapter will guide you through: - Installing Open Code Agents - Setting up your environment - Running your first agent workflow - Understanding the CLI interface

Let's embark on this journey to transform your development workflow with the power of specialized AI agents.

2 Chapter 2: Quick Start Guide

This chapter gets you up and running with Open Code Agents quickly. We'll cover installation, basic usage, and your first agent workflow.

2.1 Installation Methods

2.1.1 Method 1: Build from Source

Open Code Agents is written in Go, making it easy to build from source:

```
# Clone the repository
git clone https://github.com/gsmllg-dev/open-code-agents.git
cd open-code-agents

# Build the CLI tool
go build ./cmd/opencode-setup
```

```
# Verify the installation
```

```
./opencode-setup --help
```

2.1.2 Method 2: Download Pre-built Binary

For convenience, pre-built binaries are available:

```
# Download the latest binary for your platform
```

```
curl -L https://github.com/gsmllg-dev/open-code-agents/releases/download/latest/opencode-setup
```

```
# Make executable
```

```
chmod +x opencode-setup-linux-amd64
```

```
# Move to PATH (optional)
```

```
sudo mv opencode-setup-linux-amd64 /usr/local/bin/opencode-setup
```

Available platforms: - opencode-setup-linux-amd64.tar.gz - Linux

(x86_64) - opencode-setup-linux-arm64.tar.gz - Linux (ARM64) -

opencode-setup-macos-amd64.tar.gz - macOS (Intel) - opencode-setup-macos-arm64.tar.gz

- macOS (Apple Silicon)

2.2 Basic Commands and Usage

2.2.1 Exploring Available Options

```
# Show main help
```

```
./opencode-setup --help
```

```
# List all available agents
```

```
./opencode-setup agents list
```

```
# List all available commands
```

```
./opencode-setup commands list
```

```
# Show available workflows
```

```
./opencode-setup commands workflow
```

2.2.2 Your First Agent Workflow

Let's start with a simple example: implementing a new feature.

2.2.2.1 Step 1: Initialize Workflow

```
./opencode-setup commands workflow
```

You'll see an interactive menu:

Available Workflows:

1. new-feature - Complete feature development

- 2. bug-fix - Bug investigation and resolution
- 3. code-improvement - Code refactoring and optimization

Select workflow: 1

2.2.2.2 Step 2: Provide Context

Enter feature description: Add user authentication with JWT tokens

2.2.2.3 Step 3: Follow Agent Sequence The system will automatically orchestrate:

1. **Researcher:** Investigates JWT authentication best practices
2. **Architect:** Designs the authentication system
3. **Implementer:** Writes the authentication code
4. **Tester:** Creates comprehensive tests
5. **Reviewer:** Ensures code quality
6. **Documenter:** Creates documentation

Each agent will: - Analyze the current state - Perform their specialized task - Provide handoff to the next agent - Document their work

2.3 Understanding the CLI Interface

2.3.1 Command Structure

The CLI follows a consistent structure:

```
./opencode-setup <category> <action> [options]
```

Categories: - **agents** - Manage and execute individual agents - **commands** - Execute predefined workflows - **config** - Configure the system - **help** - Get help and documentation

2.3.2 Common Options

Verbose output

```
./opencode-setup agents execute implementer --verbose
```

Specify configuration file

```
./opencode-setup commands workflow --config ./my-config.json
```

Dry run (show what would happen)

```
./opencode-setup agents execute architect --dry-run
```

Interactive mode (default)

```
./opencode-setup commands workflow --interactive
```

```
# Non-interactive mode
./opencode-setup commands workflow --non-interactive
```

2.4 Setting Up Your Environment

2.4.1 Configuration Files

Open Code Agents uses JSON configuration files:

2.4.1.1 User-wide Configuration Location: ~/.config/opencode/config.json

```
{
  "default_agent": "implementer",
  "workflows": {
    "my-feature": "new-feature"
  },
  "settings": {
    "log_level": "info",
    "auto_save": true,
    "show_hints": true,
    "max_history": 100
  }
}
```

2.4.1.2 Project-specific Configuration Location: .opencode/config.json in project root

```
{
  "project_name": "My Web App",
  "tech_stack": ["node.js", "react", "postgresql"],
  "agents": {
    "implementer": {
      "language": "javascript",
      "framework": "react"
    },
    "tester": {
      "framework": "jest",
      "coverage_threshold": 80
    }
  }
}
```

2.4.2 Environment Variables

```
# Set default log level
export OPENCODE_LOG_LEVEL=debug
```

```
# Set configuration directory
export OPENCODE_CONFIG_DIR=/path/to/config

# Enable verbose mode
export OPENCODE_VERBOSE=true
```

2.5 Your First Custom Agent

Let's try executing a single agent directly:

2.5.1 Example: Code Review

```
# Review the current directory
./opencode-setup agents execute reviewer --scope ./src

# Review specific files
./opencode-setup agents execute reviewer --files ./src/auth.js ./src/user.js

# Review with specific criteria
./opencode-setup agents execute reviewer --criteria security,performance
```

2.5.2 Example: Bug Investigation

```
# Debug a specific issue
./opencode-setup agents execute debugger \
  --issue "Login fails with valid credentials" \
  --context "After recent authentication update"
```

2.6 Tips for Success

2.6.1 1. Start Small

Begin with simple, well-defined tasks to understand how agents work.

2.6.2 2. Provide Clear Context

The more specific your input, the better the results: - Good: “Add JWT authentication with refresh tokens” - Poor: “Add auth”

2.6.3 3. Review Agent Outputs

Always review what each agent produces before proceeding to the next step.

2.6.4 4. Use Version Control

Commit changes after each major agent completion:

```
git add .
git commit -m "feat: implementer - add authentication middleware"
```

2.6.5 5. Learn Patterns

Pay attention to how agents hand off work and what information they need.

2.7 Troubleshooting Common Issues

2.7.1 Installation Problems

Issue: command not found: opencode-setup

```
# Solution: Add to PATH or use full path
export PATH=$PATH:/path/to/opencode-setup
# or
./path/to/opencode-setup commands list
```

Issue: Permission denied

```
# Solution: Make executable
chmod +x opencode-setup
```

2.7.2 Runtime Issues

Issue: Agent asks too many questions

```
# Solution: Provide more context upfront
./opencode-setup agents execute implementer \
  --context "React app with TypeScript, using Material-UI"
```

Issue: Workflow fails midway

```
# Solution: Check logs and resume
./opencode-setup commands workflow --resume --step implementer
```

2.8 Next Steps

Congratulations! You now have: - Installed Open Code Agents - Explored the CLI interface - Run your first agent workflow - Configured your environment

In the next chapter, we'll dive deeper into the system architecture to understand how everything works under the hood.

2.9 Quick Reference

Task	Command
List agents	<code>./opencode-setup agents list</code>
Execute agent	<code>./opencode-setup agents execute <agent></code>
List workflows	<code>./opencode-setup commands workflow</code>
Run workflow	<code>./opencode-setup commands workflow</code>
Show help	<code>./opencode-setup --help</code>

Task	Command
Check version	<code>./opencode-setup --version</code>

Keep this reference handy as you explore the system!

3 Chapter 3: System Architecture

This chapter explores the technical architecture of Open Code Agents, understanding how the system is designed and how the components work together.

3.1 Core Components

3.1.1 1. CLI Interface (cmd/)

The command-line interface serves as the entry point for all interactions:

```
cmd/
  opencode-setup/
    main.go          # Main CLI entry point
```

Key Features: - Command parsing and routing - Interactive menu system - Configuration management - Error handling and logging

3.1.2 2. Agent System (pkg/agent/)

The heart of the system, managing AI agent execution:

```
pkg/agent/
  engine.go          # Agent execution engine
  engine_test.go     # Test suite
```

Core Capabilities: - Agent lifecycle management - Context preservation - Handoff coordination - State management

3.1.3 3. Configuration Management (pkg/config/)

Handles system and project configuration:

```
pkg/config/
  config.go          # Configuration structures
  agents.go          # Agent definitions
```

Configuration Types: - Global user settings - Project-specific settings - Agent configurations - Workflow definitions

3.1.4 4. CLI Commands (pkg/cli/)

Implements all CLI commands and subcommands:

```
pkg/cli/
  root.go          # Root command setup
  agents.go        # Agent management commands
  commands.go      # Workflow commands
  mcp.go           # MCP integration
  skills.go        # Skill management
```

3.1.5 5. Orchestrator (pkg/orchestrator/)

Manages complex multi-agent workflows:

```
pkg/orchestrator/
  workflow.go      # Workflow orchestration
  workflow_test.go # Workflow tests
```

Workflow Features: - Agent sequencing - Context passing - Error recovery - Progress tracking

3.2 Agent Architecture

3.2.1 Agent Definition Structure

Each agent is defined with:

```
type Agent struct {
  Name      string    // Agent identifier
  Role      string    // Professional role
  Description string    // Capabilities description
  Skills    []string  // Available skills
  Config    AgentConfig // Agent-specific settings
}
```

3.2.2 Agent Execution Model

1. **Initialization:** Load agent configuration and context
2. **Analysis:** Understand the task and current state
3. **Execution:** Perform the specialized task
4. **Validation:** Ensure quality and completeness
5. **Handoff:** Prepare context for next agent

3.2.3 Context Management

Agents share context through a structured format:

```
type Context struct {
  Project    ProjectInfo // Project details
```

```

Task          TaskInfo          // Current task
History       []AgentResult      // Previous agent results
State         map[string]interface{} // Current state
Metadata      ContextMetadata    // Additional context
}

```

3.3 Workflow Orchestration

3.3.1 Workflow Definition

Workflows are defined as sequences of agents:

```

workflows:
  new-feature:
    agents:
      - name: researcher
        role: investigate_requirements
      - name: architect
        role: design_solution
      - name: implementer
        role: implement_feature
      - name: tester
        role: create_tests
      - name: reviewer
        role: review_code
      - name: documenter
        role: document_changes

```

3.3.2 Execution Flow

1. **Workflow Selection:** User chooses appropriate workflow
2. **Agent Initialization:** Set up initial context
3. **Sequential Execution:** Run agents in defined order
4. **Context Passing:** Transfer state between agents
5. **Quality Gates:** Validate completion at each step
6. **Completion:** Final validation and cleanup

3.4 Data Flow Architecture

3.4.1 Input Processing

User Input → CLI Parser → Command Router → Agent Engine

3.4.2 Agent Execution Flow

Context → Agent → Analysis → Execution → Result → Updated Context

3.4.3 Output Generation

Agent Result → Formatter → CLI Output → File System/Console

3.5 Configuration Architecture

3.5.1 Configuration Hierarchy

1. **System Defaults:** Built-in default configurations
2. **Global Config:** User-wide settings (`~/.config/opencode/`)
3. **Project Config:** Project-specific settings (`.opencode/`)
4. **Runtime Config:** Command-line overrides

3.5.2 Configuration Merging

Configurations are merged with precedence:

Runtime > Project > Global > System Defaults

3.6 Integration Points

3.6.1 Model Context Protocol (MCP)

Open Code Agents integrates with MCP for: - External tool integration - Extended agent capabilities - Third-party service connections

3.6.2 Embed System

Static resources are embedded in the binary:

```
embed/  
  agents/          # Agent definitions  
  resources/       # Static resources
```

3.6.3 Installer System

Automated installation and setup:

```
pkg/installer/  
  agents.go        # Agent installation logic
```

3.7 Error Handling Architecture

3.7.1 Error Types

1. **Configuration Errors:** Invalid settings or missing files
2. **Agent Errors:** Agent execution failures
3. **Workflow Errors:** Workflow orchestration issues
4. **System Errors:** Infrastructure problems

3.7.2 Error Recovery Strategies

1. **Graceful Degradation:** Continue with reduced functionality
2. **Retry Logic:** Automatic retry for transient failures
3. **User Intervention:** Prompt for manual resolution
4. **Rollback:** Revert partial changes on failure

3.8 Performance Considerations

3.8.1 Caching Strategy

- **Agent Definitions:** Cache loaded agent configurations
- **Configuration:** Cache parsed configuration files
- **Context:** Preserve context between agent executions

3.8.2 Resource Management

- **Memory:** Efficient context storage and sharing
- **CPU:** Optimize agent execution parallelization
- **I/O:** Minimize file system operations

3.8.3 Scalability Design

- **Modular Architecture:** Easy addition of new agents
- **Plugin System:** Extensible capability framework
- **Distributed Execution:** Future support for remote agents

3.9 Security Architecture

3.9.1 Isolation

- **Agent Sandboxing:** Limited access to system resources
- **Context Validation:** Sanitize all inputs and outputs
- **Permission Controls:** Granular access controls

3.9.2 Data Protection

- **Sensitive Data:** Encrypt sensitive configuration values
- **Audit Logging:** Track all agent actions
- **Access Controls:** Role-based access to features

3.10 Testing Architecture

3.10.1 Test Organization

```
pkg/  
  agent/  
    engine_test.go      # Agent engine tests  
  orchestrator/
```

```
workflow_test.go    # Workflow tests
```

3.10.2 Test Types

1. **Unit Tests:** Individual component testing
2. **Integration Tests:** Component interaction testing
3. **End-to-End Tests:** Complete workflow testing
4. **Performance Tests:** Load and stress testing

3.11 Future Architecture Considerations

3.11.1 Distributed Agents

- **Remote Execution:** Support for cloud-based agents
- **Load Balancing:** Distribute work across multiple instances
- **Fault Tolerance:** Handle network failures and timeouts

3.11.2 Advanced Features

- **Learning System:** Agents learn from previous executions
- **Adaptive Workflows:** Dynamic workflow adjustment
- **Collaborative Editing:** Real-time multi-user support

3.11.3 Integration Expansion

- **IDE Plugins:** Direct integration with development environments
- **CI/CD Integration:** Native pipeline integration
- **API Access:** RESTful API for external tool integration

3.12 Summary

The Open Code Agents architecture is designed for:

- **Modularity:** Clear separation of concerns
- **Extensibility:** Easy addition of new capabilities
- **Reliability:** Robust error handling and recovery
- **Performance:** Efficient resource utilization
- **Security:** Comprehensive security measures

This architecture enables the system to scale from simple single-agent tasks to complex multi-agent workflows while maintaining reliability and performance.

4 Chapter 4: Installation and Setup

This chapter provides comprehensive guidance for installing and configuring Open Code Agents across different environments and use cases.

4.1 System Requirements

4.1.1 Minimum Requirements

- **Operating System:** Linux, macOS, or Windows (with WSL2)
- **Go Version:** 1.21 or later (for building from source)
- **Memory:** 512MB RAM minimum, 2GB recommended
- **Storage:** 100MB free disk space
- **Network:** Internet connection for initial setup

4.1.2 Recommended Requirements

- **Operating System:** Linux (Ubuntu 20.04+) or macOS (12.0+)
- **Go Version:** 1.22 or later
- **Memory:** 4GB RAM or more
- **Storage:** 500MB free disk space
- **Network:** Stable internet connection

4.1.3 Supported Platforms

Platform	Architecture	Status	Notes
Linux	x86_64	Fully Supported	Ubuntu, CentOS, Debian
Linux	ARM64	Fully Supported	Raspberry Pi, ARM servers
macOS	x86_64	Fully Supported	Intel Macs
macOS	ARM64	Fully Supported	Apple Silicon
Windows	x86_64	WSL2 Only	Use Windows Subsystem for Linux

4.2 Installation Methods

4.2.1 Method 1: Build from Source (Recommended for Developers)

4.2.1.1 Prerequisites

Install Go 1.21+:

On Ubuntu/Debian

```
sudo apt update
```

```
sudo apt install golang-go
```

On macOS

```
brew install go
```

On other systems

Download from <https://golang.org/dl/>

4.2.1.2 Build Process

Clone the repository

```
git clone https://github.com/gsmllg-dev/open-code-agents.git
```

```
cd open-code-agents
```

```
# Verify Go installation
```

```
go version
```

```
# Build the CLI tool
```

```
go build -o opencode-setup ./cmd/opencode-setup
```

```
# Test the installation
```

```
./opencode-setup --version
```

```
./opencode-setup --help
```

4.2.1.3 Development Build For development with debug information:

```
# Build with debug symbols
```

```
go build -gcflags="all=-N -l" -o opencode-setup-debug ./cmd/opencode-setup
```

```
# Build race detector version
```

```
go build -race -o opencode-setup-race ./cmd/opencode-setup
```

4.2.2 Method 2: Download Pre-built Binaries

4.2.2.1 Automated Installation Script

```
# Download and run installer
```

```
curl -fsSL https://raw.githubusercontent.com/gsmllg-dev/open-code-agents/main/setup-to-user.sh
```

```
# Or download manually
```

```
curl -L https://github.com/gsmllg-dev/open-code-agents/releases/latest/download/opencode-setup-  
chmod +x opencode-setup-$(uname -s)-$(uname -m)
```

4.2.2.2 Manual Download Visit the Releases Page and download the appropriate binary:

```
# Example for Linux x86_64
```

```
wget https://github.com/gsmllg-dev/open-code-agents/releases/download/v1.0.0/opencode-setup-  
tar xzf opencode-setup-linux-amd64.tar.gz
```

```
chmod +x opencode-setup-linux-amd64
```

```
sudo mv opencode-setup-linux-amd64 /usr/local/bin/opencode-setup
```

4.2.3 Method 3: Package Manager Installation

4.2.3.1 Homebrew (macOS)

```
# Add tap (if not already added)
```

```
brew tap gsmllg-dev/open-code-agents
```

```
# Install
```

```
brew install open-code-agents
```

```
# Verify
opencode-setup --version
```

4.2.3.2 APT (Ubuntu/Debian)

```
# Add repository
echo "deb https://packages.gsmlg.dev/ubuntu/ $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/opencode.list
curl -fsSL https://packages.gsmlg.dev/gsmlg-dev.gpg | sudo apt-key add -

# Install
sudo apt update
sudo apt install open-code-agents
```

4.3 Configuration Setup

4.3.1 Initial Configuration

Run the interactive setup:

```
opencode-setup init
```

This will guide you through: - Setting up default preferences - Configuring AI model access - Creating workspace directories - Setting up integrations

4.3.2 Manual Configuration

Create configuration directory:

```
mkdir -p ~/.config/opencode
```

Create basic configuration file:

```
{
  "version": "1.0.0",
  "settings": {
    "log_level": "info",
    "default_agent": "implementer",
    "auto_save": true,
    "show_hints": true
  },
  "agents": {
    "default_config": {
      "model": "gpt-4",
      "temperature": 0.7,
      "max_tokens": 4000
    }
  }
},
```

```

    "workflows": {
      "new_feature": {
        "agents": ["researcher", "architect", "implementer", "tester", "reviewer"],
        "auto_commit": false
      }
    }
  }
}

```

4.3.3 Environment Variables

Set up environment variables in your shell profile (`~/.bashrc`, `~/.zshrc`, etc.):

```

# Open Code Agents Configuration
export OPENCODE_CONFIG_DIR="$HOME/.config/opencode"
export OPENCODE_LOG_LEVEL="info"
export OPENCODE_DEFAULT_AGENT="implementer"

# AI Model Configuration (if using custom models)
export OPENCODE_API_KEY="your-api-key-here"
export OPENCODE_MODEL_ENDPOINT="https://api.example.com/v1"

# Development Configuration (optional)
export OPENCODE_DEV_MODE="true"
export OPENCODE_DEBUG="false"

```

4.4 Project Setup

4.4.1 Initializing a New Project

```

# Create new project directory
mkdir my-project
cd my-project

# Initialize with Open Code Agents
opencode-setup init --project

# This creates:
# .opencode/
#   config.json
#   agents/
#   workflows/

```

4.4.2 Project Configuration

Create `.opencode/config.json`:

```

{
  "project": {

```

```

    "name": "My Project",
    "type": "web_application",
    "tech_stack": ["node.js", "react", "postgresql"],
    "version": "1.0.0"
  },
  "agents": {
    "implementer": {
      "language": "javascript",
      "framework": "react",
      "style_guide": "airbnb"
    },
    "tester": {
      "framework": "jest",
      "coverage_threshold": 80,
      "test_types": ["unit", "integration", "e2e"]
    },
    "reviewer": {
      "focus_areas": ["security", "performance", "maintainability"],
      "strict_mode": true
    }
  },
  "workflows": {
    "feature": {
      "pre_checks": ["lint", "type_check"],
      "post_checks": ["test", "build"],
      "auto_commit": true
    }
  }
}

```

4.5 Integration Setup

4.5.1 Git Integration

Configure Git hooks:

```

# Install pre-commit hook
opencode-setup install git-hook pre-commit

```

```

# Install commit-msg hook
opencode-setup install git-hook commit-msg

```

Example pre-commit hook (~/.git/hooks/pre-commit):

```

#!/bin/bash
# Open Code Agents pre-commit hook

# Run code review

```

```
opencode-setup agents execute reviewer --files "$(git diff --cached --name-only)"
```

```
# Run tests if configured
if [ -f "package.json" ]; then
    npm test
fi

# Check exit code
if [ $? -ne 0 ]; then
    echo " Pre-commit checks failed"
    exit 1
fi

echo " Pre-commit checks passed"
```

4.5.2 IDE Integration

4.5.2.1 VS Code Install VS Code extension:

```
# Install extension
code --install-extension gsmlg.open-code-agents
```

```
# Or install manually from marketplace
```

Configure VS Code settings (.vscode/settings.json):

```
{
  "openCodeAgents.enabled": true,
  "openCodeAgents.autoStart": true,
  "openCodeAgents.defaultAgent": "implementer",
  "openCodeAgents.showOutput": true,
  "openCodeAgents.integratedTerminal": true
}
```

Add tasks to .vscode/tasks.json:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "Open Code: Review Current File",
      "type": "shell",
      "command": "opencode-setup",
      "args": [
        "agents", "execute", "reviewer",
        "--files", "${file}"
      ],
      "group": "build"
    }
  ]
}
```

```

    },
    {
      "label": "Open Code: Quick Fix",
      "type": "shell",
      "command": "opencode-setup",
      "args": [
        "agents", "execute", "debugger",
        "--files", "${file}"
      ],
      "group": "build"
    }
  ]
}

```

4.5.2.2 JetBrains IDEs Install plugin from JetBrains Marketplace: 1. Open File > Settings > Plugins 2. Search for “Open Code Agents” 3. Install and restart IDE

Configure in settings.xml:

```

<application>
  <component name="OpenCodeAgentsSettings">
    <option name="executablePath" value="/usr/local/bin/opencode-setup" />
    <option name="defaultAgent" value="implementer" />
    <option name="autoSync" value="true" />
  </component>
</application>

```

4.5.3 CI/CD Integration

4.5.3.1 GitHub Actions Create .github/workflows/opencode-agents.yml:

```

name: Open Code Agents CI

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  code-review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Setup Open Code Agents

```

```

    run: |
        curl -fsSL https://raw.githubusercontent.com/gsmllg-dev/open-code-agents/main/setup-t

- name: Run Code Review
  run: |
    opencode-setup agents execute reviewer --scope . --output review.md

- name: Upload Review
  uses: actions/upload-artifact@v3
  with:
    name: code-review
    path: review.md

```

4.5.3.2 Jenkins Pipeline

```

pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                checkout scm
            }
        }

        stage('Setup Open Code Agents') {
            steps {
                sh 'curl -fsSL https://raw.githubusercontent.com/gsmllg-dev/open-code-agents/
            }
        }

        stage('Code Analysis') {
            steps {
                sh 'opencode-setup agents execute reviewer --scope . --output analysis.json
                archiveArtifacts artifacts: 'analysis.json'
            }
        }
    }
}

```

4.6 Verification and Testing

4.6.1 Basic Verification

```

# Check installation
opencode-setup --version

```

```

# List available agents
opencode-setup agents list

# List available commands
opencode-setup commands list

# Test basic functionality
opencode-setup agents execute implementer --test-mode

```

4.6.2 Integration Testing

```

# Create test project
mkdir test-project
cd test-project
opencode-setup init --project

# Test workflow
echo "Add a simple hello world function" | opencode-setup commands workflow

# Verify results
ls -la
cat README.md # Should contain documentation

```

4.6.3 Performance Testing

```

# Run performance benchmark
opencode-setup benchmark --duration 60s --concurrent 5

# Test with large codebase
opencode-setup agents execute reviewer --scope /path/to/large/project --timing

```

4.7 Troubleshooting

4.7.1 Common Installation Issues

4.7.1.1 Issue: Permission Denied

```

# Solution: Make executable
chmod +x opencode-setup

# Or install system-wide
sudo mv opencode-setup /usr/local/bin/

```

4.7.1.2 Issue: Command Not Found

```

# Solution: Add to PATH
echo 'export PATH=$PATH:/path/to/opencode-setup' >> ~/.bashrc
source ~/.bashrc

```

4.7.1.3 Issue: Go Version Too Old

```
# Solution: Update Go
# Remove old version
sudo rm -rf /usr/local/go

# Install new version
wget https://golang.org/dl/go1.22.0.linux-amd64.tar.gz
sudo tar -C /usr/local -xzf go1.22.0.linux-amd64.tar.gz
```

4.7.2 Configuration Issues

4.7.2.1 Issue: Invalid Configuration File

```
# Validate configuration
opencode-setup config validate

# Reset to defaults
opencode-setup config reset
```

4.7.2.2 Issue: Agent Not Found

```
# Check available agents
opencode-setup agents list

# Reinstall agents
opencode-setup agents install --all
```

4.7.3 Performance Issues

4.7.3.1 Issue: Slow Execution

```
# Check system resources
opencode-setup system info

# Optimize configuration
opencode-setup config optimize
```

4.7.3.2 Issue: Memory Usage

```
# Monitor memory usage
opencode-setup monitor memory

# Reduce concurrent agents
opencode-setup config set max_concurrent_agents 2
```

4.8 Maintenance and Updates

4.8.1 Updating Open Code Agents

```
# Update to latest version
opencode-setup update

# Check for updates
opencode-setup check-updates

# Update specific components
opencode-setup update agents
opencode-setup update workflows
```

4.8.2 Backup Configuration

```
# Backup configuration
cp -r ~/.config/opencode ~/.config/opencode.backup.$(date +%Y%m%d)

# Export configuration
opencode-setup config export > opencode-config.json
```

4.8.3 Cleanup

```
# Clean cache
opencode-setup cleanup cache

# Remove old versions
opencode-setup cleanup old-versions

# Full reset (careful!)
opencode-setup reset --hard
```

4.9 Next Steps

After completing installation and setup:

1. **Explore the System:** Read Chapter 3 to understand the architecture
2. **Try Basic Workflows:** Start with simple single-agent tasks
3. **Configure Your Environment:** Customize settings for your workflow
4. **Integrate with Tools:** Set up IDE and CI/CD integrations
5. **Join the Community:** Get help and share experiences

You're now ready to dive into the world of AI-powered development workflows!

5 Chapter 5: The Architect Agent

The Architect agent is your strategic system design partner, specializing in creating robust, scalable, and maintainable software architectures. This chapter explores how to leverage the Architect agent for optimal system design.

5.1 Role and Responsibilities

5.1.1 Primary Focus Areas

The Architect agent excels at:

- **System Design:** Creating high-level architectural blueprints
- **Technology Selection:** Choosing appropriate technologies and frameworks
- **Scalability Planning:** Designing systems that can grow with demand
- **Integration Strategy:** Planning how components interact
- **Security Architecture:** Designing secure system foundations
- **Performance Optimization:** Planning for optimal system performance

5.1.2 Core Responsibilities

1. **Requirements Analysis:** Understanding business and technical requirements
2. **Architecture Design:** Creating comprehensive system designs
3. **Technology Evaluation:** Assessing and recommending technologies
4. **Documentation:** Creating architectural documentation
5. **Risk Assessment:** Identifying potential architectural risks
6. **Compliance:** Ensuring architectural standards compliance

5.2 When to Use the Architect Agent

5.2.1 Ideal Scenarios

Use the Architect agent when:

5.2.1.1 New System Design

Scenario: Building a new e-commerce platform

Input: "Design a scalable e-commerce platform for 100k daily users"

Output: Complete system architecture with microservices, database design, and infrastructure

5.2.1.2 System Expansion

Scenario: Expanding existing application

Input: "Our monolithic app needs to handle 10x current load. Plan the migration to microservices"

Output: Migration strategy, service decomposition plan, and timeline

5.2.1.3 Technology Migration

Scenario: Legacy system modernization

Input: "Migrate our PHP monolith to a modern stack. Recommend architecture and approach"

Output: Modernization roadmap, technology stack, and implementation strategy

5.2.1.4 Performance Optimization

Scenario: System performance issues

Input: "Our API response times are increasing. Analyze and redesign for better performance"

Output: Performance analysis and architectural improvements

5.2.2 Not Ideal For

- **Implementation Details:** Use the Implementer agent for coding
- **Bug Fixes:** Use the Debugger agent for troubleshooting
- **Code Reviews:** Use the Reviewer agent for code quality
- **Testing:** Use the Tester agent for test creation

5.3 Architect Agent Capabilities

5.3.1 Design Patterns Expertise

The Architect agent has deep knowledge of:

5.3.1.1 Architectural Patterns

- Microservices Architecture
- Event-Driven Architecture
- Domain-Driven Design (DDD)
- Service-Oriented Architecture (SOA)
- Serverless Architecture
- Hexagonal Architecture

5.3.1.2 Design Patterns

- Repository Pattern
- Factory Pattern
- Observer Pattern
- Strategy Pattern
- Command Pattern
- Decorator Pattern

5.3.1.3 Integration Patterns

- API Gateway
- Message Broker
- Event Sourcing

- CQRS (Command Query Responsibility Segregation)
- Saga Pattern
- Circuit Breaker

5.3.2 Technology Stack Assessment

The Architect agent evaluates technologies across:

5.3.2.1 Frontend Technologies

- React, Vue.js, Angular
- Next.js, Nuxt.js
- State management solutions
- CSS frameworks and methodologies
- Progressive Web App (PWA) technologies

5.3.2.2 Backend Technologies

- Node.js, Python, Go, Java, .NET
- Framework selection (Express, Django, Gin, Spring Boot)
- Database technologies (SQL vs NoSQL)
- Message queue systems
- Container orchestration

5.3.2.3 Infrastructure Technologies

- Cloud providers (AWS, GCP, Azure)
- Container technologies (Docker, Kubernetes)
- Serverless platforms
- CDN and caching solutions
- Monitoring and observability

5.4 Using the Architect Agent

5.4.1 Basic Usage

5.4.1.1 Command Line Interface

```
# Basic architecture design
opencode-setup agents execute architect \
  --task "Design a REST API for user management" \
  --context "Node.js, Express, PostgreSQL"

# Technology recommendation
opencode-setup agents execute architect \
  --task "Recommend tech stack for real-time chat app" \
  --requirements "scalable, low-latency, 10k concurrent users"
```

```
# System redesign
opencode-setup agents execute architect \
  --task "Redesign monolith to microservices" \
  --scope "user authentication, order processing, inventory"
```

5.4.1.2 Interactive Mode

```
# Start interactive architecture session
opencode-setup agents execute architect --interactive
```

```
# Example interaction:
```

```
> What would you like to design?
```

```
I need to design a file sharing service
```

```
> What are the key requirements?
```

- Support for 1M users
- File sizes up to 10GB
- Real-time collaboration
- Mobile and web apps

```
> Any technology preferences?
```

```
We prefer Node.js and cloud-native solutions
```

5.4.2 Advanced Usage

5.4.2.1 Configuration-Driven Architecture

```
# Create architecture configuration
```

```
cat > arch-config.json << EOF
{
  "project": {
    "type": "saas_application",
    "scale": "enterprise",
    "users": "100000+",
    "availability": "99.9%"
  },
  "requirements": {
    "performance": {
      "response_time": "<200ms",
      "throughput": "10000 req/s"
    },
    "security": {
      "authentication": "oauth2",
      "encryption": "aes256"
    },
    "scalability": {
      "horizontal_scaling": true,

```

```

        "auto_scaling": true
    },
    "constraints": {
        "budget": "moderate",
        "timeline": "6 months",
        "team_size": "8 developers"
    }
}
EOF

```

```

# Use configuration for architecture design
opencode-setup agents execute architect \
    --config arch-config.json \
    --output architecture-plan.md

```

5.4.2.2 Multi-Phase Architecture

```

# Phase 1: High-level design
opencode-setup agents execute architect \
    --task "Design system architecture" \
    --phase "high_level" \
    --output phase1-architecture.md

# Phase 2: Detailed design
opencode-setup agents execute architect \
    --task "Detail component design" \
    --phase "detailed" \
    --input phase1-architecture.md \
    --output phase2-architecture.md

# Phase 3: Implementation plan
opencode-setup agents execute architect \
    --task "Create implementation roadmap" \
    --phase "implementation" \
    --input phase2-architecture.md \
    --output implementation-plan.md

```

5.5 Input and Output Examples

5.5.1 Input Examples

5.5.1.1 Simple Request

Design a microservices architecture for an e-commerce platform

5.5.1.2 Detailed Request

Design a scalable e-commerce platform with the following requirements:

- 100k daily active users
- Support for product catalog, shopping cart, payments, user accounts
- Mobile app and web frontend
- Real-time inventory updates
- PCI compliance for payments
- Technology preference: Node.js, React, PostgreSQL, Redis
- Deployment: AWS with auto-scaling

5.5.1.3 Context-Rich Request

We need to redesign our legacy monolithic application. Current issues:

- Single database bottleneck
- Difficult to deploy individual features
- Technology stack is outdated (PHP 5.6, MySQL 5.5)
- No automated testing or deployment

Requirements for new architecture:

- Microservices approach
- CI/CD pipeline
- Modern technology stack
- 99.9% uptime requirement
- Support for 10x growth in users
- Team of 12 developers with varying skill levels

5.5.2 Output Examples

5.5.2.1 Architecture Overview

E-Commerce Platform Architecture

System Overview

- **Architecture Pattern**: Microservices with API Gateway
- **Technology Stack**: Node.js, React, PostgreSQL, Redis, Docker
- **Infrastructure**: AWS EKS, RDS, ElastiCache, CloudFront
- **Scalability**: Horizontal auto-scaling with load balancing

Core Services

1. **User Service**: Authentication, authorization, profile management
2. **Product Service**: Catalog management, search, recommendations
3. **Order Service**: Cart management, order processing, payment integration
4. **Inventory Service**: Stock management, real-time updates
5. **Notification Service**: Email, SMS, push notifications

Data Architecture

- **Primary Database**: PostgreSQL with read replicas
- **Caching Layer**: Redis for session and product data

- ****Search Engine****: Elasticsearch for product search
- ****File Storage****: S3 for images and documents

5.5.2.2 Component Diagram

```
graph TB
    Client[Web/Mobile Client] --> Gateway[API Gateway]
    Gateway --> Auth[User Service]
    Gateway --> Product[Product Service]
    Gateway --> Order[Order Service]
    Gateway --> Inventory[Inventory Service]

    Auth --> UserDB[(User DB)]
    Product --> ProductDB[(Product DB)]
    Order --> OrderDB[(Order DB)]
    Inventory --> InventoryDB[(Inventory DB)]

    Product --> Cache[Redis Cache]
    Product --> Search[Elasticsearch]

    Gateway --> Queue[Message Queue]
    Order --> Queue
    Inventory --> Queue
```

5.5.2.3 Technology Recommendations

Technology Stack Recommendations

Backend Services

- ****Runtime****: Node.js 18+ with TypeScript
- ****Framework****: Express.js with middleware architecture
- ****Database****: PostgreSQL 14+ with connection pooling
- ****ORM****: Prisma for type-safe database access
- ****Authentication****: JWT with refresh tokens

Frontend Applications

- ****Web****: React 18 with Next.js 13
- ****Mobile****: React Native with Expo
- ****State Management****: Redux Toolkit with RTK Query
- ****UI Components****: Material-UI (MUI)

Infrastructure

- ****Containerization****: Docker with multi-stage builds
- ****Orchestration****: Kubernetes with Helm charts
- ****CI/CD****: GitHub Actions with ArgoCD
- ****Monitoring****: Prometheus + Grafana + Jaeger

5.6 Best Practices for Architect Agent

5.6.1 Providing Good Input

5.6.1.1 Good Input Examples

Design a task management system for:

- 50k users
- Real-time collaboration
- Mobile and web apps
- Technology: Python, Django, PostgreSQL
- Deployment: AWS with auto-scaling

5.6.1.2 Poor Input Examples

Design an app

5.6.2 Structuring Requirements

Use the following structure for best results:

```
## Project Context
[Brief description of the project and business goals]

## Functional Requirements
[List of features and capabilities]

## Non-Functional Requirements
[Performance, security, scalability requirements]

## Technical Constraints
[Technology preferences, team skills, budget limitations]

## Business Constraints
[Timeline, team size, regulatory requirements]
```

5.6.3 Reviewing Architect Output

When reviewing the Architect agent's output:

1. **Validate Requirements:** Ensure all requirements are addressed
2. **Check Feasibility:** Verify the proposed solution is achievable
3. **Assess Trade-offs:** Understand the decisions and alternatives
4. **Review Integration:** Ensure components work together properly
5. **Plan Implementation:** Create actionable next steps

5.7 Integration with Other Agents

5.7.1 Typical Workflow Sequence

1. **Researcher** (optional): Research requirements and best practices
2. **Architect**: Design system architecture
3. **Implementer**: Implement based on architectural design
4. **Tester**: Create tests for the implemented system
5. **Reviewer**: Review implementation against architecture
6. **Documenter**: Document the final system

5.7.2 Handoff Patterns

5.7.2.1 Architect → Implementer The Architect provides: - System design documents - Component specifications - API contracts - Database schemas - Technology recommendations

5.7.2.2 Architect → Reviewer The Architect provides: - Architecture principles - Design decisions documentation - Trade-off analysis - Compliance requirements

5.8 Common Use Cases

5.8.1 Use Case 1: Startup MVP Architecture

Input: “Design MVP architecture for social media app”

Output: - Monolithic architecture for rapid development - Simple database design - Basic authentication system - Deployment on single server - Migration path to microservices

5.8.2 Use Case 2: Enterprise System Integration

Input: “Integrate three legacy systems into unified platform”

Output: - API Gateway pattern - Message broker for async communication - Data synchronization strategy - Security integration approach - Phased migration plan

5.8.3 Use Case 3: Performance Optimization

Input: “Optimize architecture for 100x traffic growth”

Output: - Caching strategy - Database optimization - Load balancing configuration - CDN implementation - Auto-scaling policies

5.9 Troubleshooting

5.9.1 Common Issues

5.9.1.1 Issue: Vague Requirements Problem: Insufficient detail leads to generic architecture **Solution:** Provide detailed requirements and constraints

5.9.1.2 Issue: Technology Mismatch Problem: Recommended technologies don't fit team skills **Solution:** Include team capabilities and learning curve considerations

5.9.1.3 Issue: Over-Engineering Problem: Complex architecture for simple requirements **Solution:** Focus on YAGNI (You Aren't Gonna Need It) principle

5.9.1.4 Issue: Missing Constraints Problem: Architecture ignores budget or timeline constraints **Solution:** Clearly specify all business and technical constraints

5.9.2 Getting Better Results

1. **Provide Context:** Include business goals and constraints
2. **Specify Scale:** Define expected user load and growth
3. **List Technologies:** Include current stack and preferences
4. **Define Timeline:** Specify development and deployment timeline
5. **Team Assessment:** Describe team skills and experience level

5.10 Advanced Features

5.10.1 Architecture Patterns Library

```
# Browse available patterns
opencode-setup agents execute architect --list-patterns

# Apply specific pattern
opencode-setup agents execute architect \
  --pattern "microservices" \
  --context "e-commerce platform"
```

5.10.2 Technology Comparison

```
# Compare technologies
opencode-setup agents execute architect \
  --task "Compare databases for high-write workload" \
  --compare "postgresql,mongodb,cassandra"
```

5.10.3 Cost Estimation

```
# Estimate infrastructure costs
opencode-setup agents execute architect \
  --task "Estimate AWS costs for proposed architecture" \
  --duration "monthly" \
  --region "us-east-1"
```

5.11 Summary

The Architect agent is your strategic partner for:

- **System Design:** Creating robust, scalable architectures
- **Technology Selection:** Making informed technology decisions
- **Strategic Planning:** Planning for growth and evolution
- **Risk Mitigation:** Identifying and addressing architectural risks

By providing clear requirements and context, you can leverage the Architect agent to create designs that serve your project's needs both now and in the future.

The key to success with the Architect agent is providing detailed, structured input and carefully reviewing the output to ensure it meets all your requirements and constraints.

6 Chapter 6: The Implementer Agent

The Implementer agent is your expert code craftsman, specializing in writing clean, maintainable, and production-ready code. This chapter explores how to effectively use the Implementer agent for high-quality software development.

6.1 Role and Responsibilities

6.1.1 Primary Focus Areas

The Implementer agent excels at:

- **Code Development:** Writing clean, efficient, and maintainable code
- **Best Practices:** Following industry coding standards and patterns
- **Problem Solving:** Translating requirements into working solutions
- **Code Organization:** Structuring code for readability and maintainability
- **Performance Optimization:** Writing efficient and performant code
- **Testing Integration:** Creating code that's easily testable

6.1.2 Core Responsibilities

1. **Requirement Analysis:** Understanding functional and non-functional requirements

2. **Code Implementation:** Writing production-quality code
3. **Design Patterns:** Applying appropriate design patterns
4. **Error Handling:** Implementing robust error handling
5. **Documentation:** Writing clear code comments and documentation
6. **Integration:** Ensuring code integrates well with existing systems

6.2 When to Use the Implementer Agent

6.2.1 Ideal Scenarios

Use the Implementer agent when:

6.2.1.1 Feature Implementation

Scenario: Adding new functionality to an application

Input: "Implement user authentication with JWT tokens, including login, logout, and token refresh"

Output: Complete authentication module with proper error handling and security measures

6.2.1.2 API Development

Scenario: Creating REST API endpoints

Input: "Create REST API for user management with CRUD operations, validation, and error responses"

Output: Complete API controller with middleware, validation, and proper HTTP status codes

6.2.1.3 Component Development

Scenario: Building reusable UI components

Input: "Create a reusable data table component with sorting, filtering, and pagination for React"

Output: Fully functional React component with TypeScript types and comprehensive props

6.2.1.4 Algorithm Implementation

Scenario: Implementing complex business logic

Input: "Implement a recommendation algorithm based on user behavior and collaborative filtering"

Output: Optimized algorithm with proper data structures and edge case handling

6.2.2 Not Ideal For

- **System Architecture:** Use the Architect agent for high-level design
- **Testing Strategy:** Use the Tester agent for comprehensive test planning
- **Code Review:** Use the Reviewer agent for quality assessment
- **Bug Investigation:** Use the Debugger agent for troubleshooting

6.3 Implementer Agent Capabilities

6.3.1 Programming Languages Expertise

The Implementer agent has deep knowledge of:

6.3.1.1 Web Development

- **JavaScript/TypeScript:** Node.js, Express, React, Vue, Angular
- **Python:** Django, Flask, FastAPI, pandas, NumPy
- **Go:** Gin, Echo, standard library, concurrency patterns
- **Java:** Spring Boot, Jakarta EE, Maven/Gradle
- **PHP:** Laravel, Symfony, Composer

6.3.1.2 Mobile Development

- **React Native:** Cross-platform mobile apps
- **Flutter:** Dart-based mobile development
- **Swift:** iOS native development
- **Kotlin:** Android native development

6.3.1.3 Systems Programming

- **Rust:** Systems programming with memory safety
- **C++:** High-performance systems development
- **C:** Low-level system programming

6.3.1.4 Data Science

- **Python:** pandas, NumPy, scikit-learn, TensorFlow
- **R:** Statistical analysis and visualization
- **SQL:** Database queries and optimization

6.3.2 Framework and Library Expertise

6.3.2.1 Frontend Frameworks

- React, Vue.js, Angular, Svelte
- Next.js, Nuxt.js, Gatsby
- Material-UI, Ant Design, Bootstrap
- Redux, MobX, Zustand

6.3.2.2 Backend Frameworks

- Express.js, FastAPI, Django, Flask
- Spring Boot, ASP.NET Core
- Gin, Echo (Go)
- Laravel, Symfony (PHP)

6.3.2.3 Database Technologies

- PostgreSQL, MySQL, MongoDB
- Redis, Cassandra, DynamoDB
- Prisma, SQLAlchemy, TypeORM
- GraphQL, REST APIs

6.3.3 Development Patterns and Practices

6.3.3.1 Code Organization

- Modular architecture
- Clean code principles
- SOLID principles
- Design patterns implementation
- Separation of concerns

6.3.3.2 Error Handling

- Graceful error handling
- Proper logging
- User-friendly error messages
- Recovery strategies
- Input validation

6.3.3.3 Performance Optimization

- Efficient algorithms
- Memory management
- Database query optimization
- Caching strategies
- Async/await patterns

6.4 Using the Implementer Agent

6.4.1 Basic Usage

6.4.1.1 Command Line Interface

```
# Basic feature implementation
opencode-setup agents execute implementer \
  --task "Implement user registration with email verification" \
  --language "javascript" \
  --framework "express"

# API endpoint creation
opencode-setup agents execute implementer \
  --task "Create REST API endpoints for product management" \
  --context "Node.js, Express, PostgreSQL, Prisma"

# Component development
opencode-setup agents execute implementer \
  --task "Create reusable modal component" \
  --language "typescript" \
```

```
--framework "react" \  
--library "material-ui"
```

6.4.1.2 Interactive Mode

```
# Start interactive implementation session  
opencode-setup agents execute implementer --interactive
```

```
# Example interaction:
```

```
> What would you like to implement?  
I need to implement a file upload service
```

```
> What are the requirements?  
- Support multiple file types  
- Progress tracking  
- File size limits  
- Cloud storage integration
```

```
> Any specific technologies?  
Node.js with Express and AWS S3
```

6.4.2 Advanced Usage

6.4.2.1 Specification-Driven Development

```
# Create implementation specification  
cat > feature-spec.json << EOF  
{  
  "feature": "user_authentication",  
  "requirements": {  
    "login": {  
      "method": "POST",  
      "endpoint": "/api/auth/login",  
      "input": ["email", "password"],  
      "output": ["token", "user", "expires_in"]  
    },  
    "register": {  
      "method": "POST",  
      "endpoint": "/api/auth/register",  
      "input": ["email", "password", "name"],  
      "output": ["user", "message"]  
    },  
    "refresh": {  
      "method": "POST",  
      "endpoint": "/api/auth/refresh",  
      "input": ["refresh_token"],  
      "output": ["token", "expires_in"]  
    }  
  }  
}
```

```

    }
  },
  "security": {
    "password_hashing": "bcrypt",
    "jwt_secret": "env_variable",
    "token_expiry": "24h"
  },
  "validation": {
    "email": "required|email",
    "password": "required|min:8"
  }
}
EOF

# Implement from specification
opencode-setup agents execute implementer \
  --spec feature-spec.json \
  --output auth-module/

```

6.4.2.2 Incremental Implementation

```

# Phase 1: Basic structure
opencode-setup agents execute implementer \
  --task "Create basic user model and database schema" \
  --phase "structure" \
  --output models/user.js

# Phase 2: Core functionality
opencode-setup agents execute implementer \
  --task "Implement user CRUD operations" \
  --phase "core" \
  --input models/user.js \
  --output controllers/userController.js

# Phase 3: API endpoints
opencode-setup agents execute implementer \
  --task "Create API routes for user management" \
  --phase "api" \
  --input controllers/userController.js \
  --output routes/users.js

```

6.4.2.3 Test-Driven Implementation

```

# Generate tests first
opencode-setup agents execute tester \
  --task "Create tests for user authentication" \

```

```

--output tests/auth.test.js

# Implement to pass tests
opencode-setup agents execute implementer \
  --task "Implement authentication to satisfy tests" \
  --test-driven true \
  --tests tests/auth.test.js \
  --output auth/

```

6.5 Input and Output Examples

6.5.1 Input Examples

6.5.1.1 Simple Request

Implement a REST API for managing blog posts with CRUD operations

6.5.1.2 Detailed Request

Implement a user management system with the following features:

- User registration with email verification
- Login with JWT authentication
- Password reset functionality
- Profile management with avatar upload
- Role-based access control (admin, user, moderator)

Technical requirements:

- Node.js with Express framework
- PostgreSQL database with Prisma ORM
- Input validation with Joi
- Error handling middleware
- Logging with Winston
- Unit tests with Jest

Security requirements:

- Password hashing with bcrypt
- Rate limiting on auth endpoints
- CORS configuration
- Input sanitization

6.5.1.3 API Specification

Create REST API endpoints for product management:

```

GET /api/products - List all products (paginated)
GET /api/products/:id - Get product by ID
POST /api/products - Create new product (admin only)

```

PUT /api/products/:id - Update product (admin only)
DELETE /api/products/:id - Delete product (admin only)

Product schema:

- id: UUID (primary key)
- name: string (required, max 255 chars)
- description: text (optional)
- price: decimal (required, min 0)
- category: string (required)
- stock: integer (required, min 0)
- created_at: timestamp
- updated_at: timestamp

Validation:

- Name and price are required
- Price must be positive
- Stock cannot be negative
- Category must be from predefined list

6.5.2 Output Examples

6.5.2.1 Complete API Implementation

```
// controllers/productController.js
const { Product } = require('../models');
const { validationResult } = require('express-validator');
const logger = require('../utils/logger');

class ProductController {
  // GET /api/products
  async getProducts(req, res) {
    try {
      const page = parseInt(req.query.page) || 1;
      const limit = parseInt(req.query.limit) || 10;
      const offset = (page - 1) * limit;

      const { count, rows: products } = await Product.findAndCountAll({
        limit,
        offset,
        order: [['created_at', 'DESC']]
      });

      res.json({
        products,
        pagination: {
          page,
```

```

        limit,
        total: count,
        pages: Math.ceil(count / limit)
    }
});
} catch (error) {
    logger.error('Error fetching products:', error);
    res.status(500).json({ error: 'Internal server error' });
}
}

// GET /api/products/:id
async getProductById(req, res) {
    try {
        const product = await Product.findByPk(req.params.id);

        if (!product) {
            return res.status(404).json({ error: 'Product not found' });
        }

        res.json(product);
    } catch (error) {
        logger.error('Error fetching product:', error);
        res.status(500).json({ error: 'Internal server error' });
    }
}

// POST /api/products
async createProduct(req, res) {
    try {
        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            return res.status(400).json({
                error: 'Validation failed',
                details: errors.array()
            });
        }

        const product = await Product.create(req.body);

        logger.info(`Product created: ${product.id}`);
        res.status(201).json(product);
    } catch (error) {
        logger.error('Error creating product:', error);
        res.status(500).json({ error: 'Internal server error' });
    }
}

```

```

}

// PUT /api/products/:id
async updateProduct(req, res) {
  try {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({
        error: 'Validation failed',
        details: errors.array()
      });
    }

    const product = await Product.findById(req.params.id);

    if (!product) {
      return res.status(404).json({ error: 'Product not found' });
    }

    await product.update(req.body);

    logger.info(`Product updated: ${product.id}`);
    res.json(product);
  } catch (error) {
    logger.error('Error updating product:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
}

// DELETE /api/products/:id
async deleteProduct(req, res) {
  try {
    const product = await Product.findById(req.params.id);

    if (!product) {
      return res.status(404).json({ error: 'Product not found' });
    }

    await product.destroy();

    logger.info(`Product deleted: ${product.id}`);
    res.status(204).send();
  } catch (error) {
    logger.error('Error deleting product:', error);
    res.status(500).json({ error: 'Internal server error' });
  }
}

```

```

    }
  }

  module.exports = new ProductController();

```

6.5.2.2 React Component Implementation

```

// components/DataTable.tsx
import React, { useState, useEffect, useMemo } from 'react';
import {
  Table,
  TableBody,
  TableCell,
  TableContainer,
  TableHead,
  TableRow,
  Paper,
  TablePagination,
  TextField,
  InputAdornment,
  IconButton
} from '@mui/material';
import { Search, ArrowUpward, ArrowDownward } from '@mui/icons-material';

interface Column<T> {
  id: keyof T;
  label: string;
  sortable?: boolean;
  filterable?: boolean;
  render?: (value: any, row: T) => React.ReactNode;
}

interface DataTableProps<T> {
  data: T[];
  columns: Column<T>[];
  rowsPerPage?: number;
  searchable?: boolean;
  className?: string;
}

function DataTable<T extends Record<string, any>>({
  data,
  columns,
  rowsPerPage = 10,
  searchable = true,
  className

```

```

}: DataTableProps<T>) {
  const [page, setPage] = useState(0);
  const [rowsPerPageState, setRowsPerPageState] = useState(rowsPerPage);
  const [searchTerm, setSearchTerm] = useState('');
  const [sortColumn, setSortColumn] = useState<keyof T | null>(null);
  const [sortDirection, setSortDirection] = useState<'asc' | 'desc'>('asc');

  // Filter and sort data
  const filteredAndSortedData = useMemo(() => {
    let filtered = data;

    // Apply search filter
    if (searchTerm && searchable) {
      filtered = data.filter(row => {
        columns.some(column => {
          const value = row[column.id];
          return value &&
            value.toString().toLowerCase().includes(searchTerm.toLowerCase());
        })
      });
    }

    // Apply sorting
    if (sortColumn) {
      filtered = [...filtered].sort((a, b) => {
        const aValue = a[sortColumn];
        const bValue = b[sortColumn];

        if (aValue === null || aValue === undefined) return 1;
        if (bValue === null || bValue === undefined) return -1;

        let comparison = 0;
        if (aValue > bValue) comparison = 1;
        if (aValue < bValue) comparison = -1;

        return sortDirection === 'desc' ? -comparison : comparison;
      });
    }

    return filtered;
  }, [data, searchTerm, sortColumn, sortDirection, columns, searchable]);

  // Paginate data
  const paginatedData = useMemo(() => {
    const startIndex = page * rowsPerPageState;
    return filteredAndSortedData.slice(startIndex, startIndex + rowsPerPageState);
  }, [filteredAndSortedData, page, rowsPerPageState]);

```

```

    }, [filteredAndSortedData, page, rowsPerPageState]);

    const handleSort = (column: keyof T) => {
      if (sortColumn === column) {
        setSortDirection(sortDirection === 'asc' ? 'desc' : 'asc');
      } else {
        setSortColumn(column);
        setSortDirection('asc');
      }
    };

    const handleChangePage = (event: unknown, newPage: number) => {
      setPage(newPage);
    };

    const handleChangeRowsPerPage = (event: React.ChangeEvent<HTMLInputElement>) => {
      setRowsPerPageState(parseInt(event.target.value, 10));
      setPage(0);
    };

    return (
      <Paper className={className}>
        {searchable && (
          <TextField
            fullWidth
            variant="outlined"
            placeholder="Search..."
            value={searchTerm}
            onChange={(e) => setSearchTerm(e.target.value)}
            InputProps={{
              startAdornment: (
                <InputAdornment position="start">
                  <Search />
                </InputAdornment>
              ),
            }}
            sx={{ p: 2, pb: 1 }}
          />
        )}

        <TableContainer>
          <Table>
            <TableHead>
              <TableRow>
                {columns.map((column) => (
                  <TableCell key={column.id.toString()}>

```

```

        {column.sortable ? (
          <IconButton
            size="small"
            onClick={() => handleSort(column.id)}
            sx={{ fontWeight: 'bold' }}
          >
            {column.label}
            {sortColumn === column.id && (
              sortDirection === 'asc' ? <ArrowUpward /> : <ArrowDownward />
            )}
          </IconButton>
        ) : (
          column.label
        )}
      </TableCell>
    )}
  </TableRow>
</TableHead>
<TableBody>
  {paginatedData.map((row, index) => (
    <TableRow key={index}>
      {columns.map((column) => (
        <TableCell key={column.id.toString()}>
          {column.render
            ? column.render(row[column.id], row)
            : row[column.id]
          }
        </TableCell>
      )}
    </TableRow>
  )}
</TableBody>
</Table>
</TableContainer>

<TablePagination
  rowsPerPageOptions={[5, 10, 25, 50]}
  component="div"
  count={filteredAndSortedData.length}
  rowsPerPage={rowsPerPageState}
  page={page}
  onPageChange={handleChangePage}
  onRowsPerPageChange={handleChangeRowsPerPage}
/>
</Paper>
);

```

```
}  
  
export default DataTable;
```

6.6 Best Practices for Implementer Agent

6.6.1 Providing Good Input

6.6.1.1 Good Input Examples

Implement a REST API for user management with:

- CRUD operations for users
- JWT authentication
- Input validation with Joi
- Error handling middleware
- PostgreSQL database with Prisma ORM
- Unit tests with Jest

6.6.1.2 Poor Input Examples

Make a user API

6.6.2 Structuring Requirements

Use the following structure for best results:

```
## Feature Description  
[Clear description of what needs to be implemented]  
  
## Functional Requirements  
[List of specific features and capabilities]  
  
## Technical Requirements  
[Programming language, frameworks, libraries]  
  
## Non-Functional Requirements  
[Performance, security, scalability requirements]  
  
## Integration Requirements  
[How it should integrate with existing code]  
  
## Testing Requirements  
[What tests should be included]
```

6.6.3 Reviewing Implementation Output

When reviewing the Implementer agent's output:

1. **Functionality:** Does it meet all requirements?

2. **Code Quality:** Is the code clean and maintainable?
3. **Error Handling:** Are edge cases properly handled?
4. **Security:** Are security best practices followed?
5. **Performance:** Is the code efficient?
6. **Testing:** Are adequate tests included?

6.7 Integration with Other Agents

6.7.1 Typical Workflow Sequence

1. **Architect:** Design system architecture
2. **Implementer:** Implement based on architectural design
3. **Tester:** Create comprehensive tests
4. **Reviewer:** Review code quality and architecture compliance
5. **Documenter:** Create documentation

6.7.2 Handoff Patterns

6.7.2.1 Implementer → Tester The Implementer provides: - Complete implementation code - Usage examples - Test requirements documentation - Edge case documentation

6.7.2.2 Implementer → Reviewer The Implementer provides: - Source code with comments - Implementation decisions documentation - Trade-off analysis - Performance considerations

6.8 Common Use Cases

6.8.1 Use Case 1: API Development

Input: “Create REST API for blog post management”

Output: - Express.js controller with CRUD operations - Input validation middleware - Error handling - Database integration - API documentation

6.8.2 Use Case 2: Component Development

Input: “Build reusable data table component for React”

Output: - TypeScript React component - Props interface definition - Sorting and filtering functionality - Pagination support - Storybook stories

6.8.3 Use Case 3: Business Logic Implementation

Input: “Implement order processing logic with payment integration”

Output: - Order processing service - Payment gateway integration - Inventory management - Notification system - Error handling and recovery

6.9 Troubleshooting

6.9.1 Common Issues

6.9.1.1 Issue: Incomplete Requirements Problem: Missing requirements lead to incomplete implementation **Solution:** Provide detailed specifications and examples

6.9.1.2 Issue: Technology Mismatch Problem: Implementation uses different technology than expected **Solution:** Clearly specify technology stack and preferences

6.9.1.3 Issue: Missing Error Handling Problem: Code lacks proper error handling **Solution:** Specify error handling requirements and edge cases

6.9.1.4 Issue: No Tests Problem: Implementation lacks test coverage **Solution:** Request specific testing requirements

6.9.2 Getting Better Results

1. **Be Specific:** Provide detailed requirements and constraints
2. **Include Examples:** Show expected input/output formats
3. **Specify Standards:** Define coding standards and patterns to follow
4. **Request Tests:** Always ask for appropriate test coverage
5. **Provide Context:** Include information about existing codebase

6.10 Advanced Features

6.10.1 Code Generation from OpenAPI

```
# Generate API client from OpenAPI spec
opencode-setup agents execute implementer \
  --task "Generate API client from OpenAPI specification" \
  --openapi-spec api.yaml \
  --language "typescript" \
  --output src/api/
```

6.10.2 Database Migration Generation

```
# Generate database migrations
opencode-setup agents execute implementer \
  --task "Create database migrations for user schema changes" \
  --database "postgresql" \
  --orm "prisma" \
  --output prisma/migrations/
```

6.10.3 Performance Optimization

```
# Optimize existing code
opencode-setup agents execute implementer \
  --task "Optimize slow database queries" \
  --input src/models/user.js \
  --profile true \
  --output optimized/
```

6.11 Summary

The Implementer agent is your expert code craftsman for:

- **Feature Development:** Building complete, working features
- **Code Quality:** Writing clean, maintainable code
- **Best Practices:** Following industry standards and patterns
- **Integration:** Ensuring code works well with existing systems

By providing detailed requirements and technical specifications, you can leverage the Implementer agent to create high-quality, production-ready code that meets your exact needs.

The key to success with the Implementer agent is providing clear, comprehensive specifications and reviewing the output to ensure it meets all your requirements and quality standards.

7 Chapter 7: The Tester Agent

The Tester agent is your quality assurance expert, specializing in creating comprehensive test suites that ensure software reliability and maintainability. This chapter explores how to leverage the Tester agent for robust testing strategies.

7.1 Role and Responsibilities

7.1.1 Primary Focus Areas

The Tester agent excels at:

- **Test Strategy:** Designing comprehensive testing approaches
- **Test Implementation:** Writing automated tests across all levels
- **Test Coverage:** Ensuring thorough coverage of functionality
- **Quality Assurance:** Establishing quality gates and standards
- **Performance Testing:** Creating performance and load tests
- **Test Automation:** Setting up automated testing pipelines

7.1.2 Core Responsibilities

1. **Test Planning:** Designing test strategies and test plans

2. **Test Implementation:** Writing unit, integration, and end-to-end tests
3. **Test Data Management:** Creating and managing test data
4. **Test Environment Setup:** Configuring test environments
5. **Regression Testing:** Ensuring new changes don't break existing functionality
6. **Test Reporting:** Creating comprehensive test reports and metrics

7.2 When to Use the Tester Agent

7.2.1 Ideal Scenarios

Use the Tester agent when:

7.2.1.1 New Feature Testing

Scenario: Testing a new user authentication system

Input: "Create comprehensive tests for JWT authentication with login, logout, and token refresh"

Output: Complete test suite covering authentication flows, edge cases, and security scenarios

7.2.1.2 API Testing

Scenario: Testing REST API endpoints

Input: "Create integration tests for product management API with CRUD operations"

Output: API tests covering all endpoints, error cases, and data validation

7.2.1.3 Component Testing

Scenario: Testing React components

Input: "Create unit tests for reusable data table component with sorting and filtering"

Output: Component tests covering user interactions, props validation, and edge cases

7.2.1.4 Performance Testing

Scenario: Load testing critical endpoints

Input: "Create performance tests for user registration endpoint handling 1000 concurrent users"

Output: Load tests with performance benchmarks and bottleneck identification

7.2.2 Not Ideal For

- **Code Implementation:** Use the Implementer agent for writing production code
- **Architecture Design:** Use the Architect agent for system design
- **Bug Fixing:** Use the Debugger agent for troubleshooting
- **Code Review:** Use the Reviewer agent for quality assessment

7.3 Tester Agent Capabilities

7.3.1 Testing Types Expertise

The Tester agent has deep knowledge of:

7.3.1.1 Unit Testing

- Test structure and organization
- Mock and stub creation
- Assertion libraries
- Test doubles and fakes
- Boundary value testing
- Equivalence partitioning

7.3.1.2 Integration Testing

- API testing
- Database integration
- External service integration
- Message queue testing
- Component integration
- Service orchestration

7.3.1.3 End-to-End Testing

- User journey testing
- Cross-browser testing
- Mobile app testing
- Workflow testing
- UI automation
- Visual regression testing

7.3.1.4 Performance Testing

- Load testing
- Stress testing
- Spike testing
- Volume testing
- Scalability testing
- Performance profiling

7.3.2 Testing Frameworks and Tools

7.3.2.1 JavaScript/TypeScript

- **Jest**: Popular testing framework
- **Mocha**: Flexible testing framework
- **Vitest**: Fast unit test framework

- **Cypress:** End-to-end testing
- **Playwright:** Modern E2E testing
- **Testing Library:** Component testing utilities

7.3.2.2 Python

- **Pytest:** Powerful testing framework
- **Unittest:** Built-in testing framework
- **Robot Framework:** Acceptance testing
- **Locust:** Load testing
- **Selenium:** Web automation

7.3.2.3 Java

- **JUnit:** Standard testing framework
- **TestNG:** Advanced testing framework
- **Mockito:** Mocking framework
- **Rest-Assured:** API testing
- **Selenide:** Web automation

7.3.2.4 Go

- **Testing package:** Built-in testing
- **Testify:** Assertion toolkit
- **Ginkgo:** BDD testing framework
- **Gomega:** Matcher library

7.3.3 Testing Strategies and Patterns

7.3.3.1 Test Pyramid

- Unit tests (70%)
- Integration tests (20%)
- End-to-end tests (10%)

7.3.3.2 Testing Patterns

- Arrange-Act-Assert pattern
- Given-When-Then pattern
- Page Object Model
- Data-Driven Testing
- Behavior-Driven Development

7.3.3.3 Quality Gates

- Code coverage thresholds
- Performance benchmarks
- Security scanning

- Accessibility testing
- Usability testing

7.4 Using the Tester Agent

7.4.1 Basic Usage

7.4.1.1 Command Line Interface

```
# Basic test creation
opencode-setup agents execute tester \
  --task "Create tests for user authentication service" \
  --type "unit,integration" \
  --framework "jest"

# API testing
opencode-setup agents execute tester \
  --task "Create integration tests for product API" \
  --type "integration" \
  --framework "supertest"

# Component testing
opencode-setup agents execute tester \
  --task "Create tests for React data table component" \
  --type "unit" \
  --framework "react-testing-library"
```

7.4.1.2 Interactive Mode

```
# Start interactive testing session
opencode-setup agents execute tester --interactive

# Example interaction:
> What would you like to test?
I need to test the user registration API

> What type of tests?
Unit tests for validation logic and integration tests for API endpoints

> Any specific requirements?
Cover success cases, validation errors, and edge cases
```

7.4.2 Advanced Usage

7.4.2.1 Comprehensive Test Suite Generation

```
# Create test specification
cat > test-requirements.json << EOF
```

```

{
  "feature": "user_management",
  "components": [
    {
      "name": "userController",
      "type": "unit",
      "framework": "jest",
      "coverage_target": 90
    },
    {
      "name": "userAPI",
      "type": "integration",
      "framework": "supertest",
      "endpoints": [
        "POST /api/users/register",
        "POST /api/users/login",
        "GET /api/users/profile",
        "PUT /api/users/profile"
      ]
    },
    {
      "name": "userWorkflow",
      "type": "e2e",
      "framework": "cypress",
      "scenarios": [
        "user_registration_flow",
        "login_and_profile_update",
        "password_reset_flow"
      ]
    }
  ],
  "test_data": {
    "valid_users": 10,
    "invalid_emails": 5,
    "edge_cases": ["duplicate_email", "weak_password", "missing_fields"]
  },
  "performance": {
    "load_test": true,
    "concurrent_users": 100,
    "duration": "5m"
  }
}
EOF

```

```

# Generate comprehensive test suite
opencode-setup agents execute tester \

```

```
--spec test-requirements.json \  
--output tests/
```

7.4.2.2 Test-Driven Development Support

```
# Generate failing tests first  
opencode-setup agents execute tester \  
  --task "Create TDD tests for shopping cart functionality" \  
  --tdd true \  
  --output tests/cart.test.js  
  
# Implementation will be done by Implementer agent  
opencode-setup agents execute implementer \  
  --task "Implement shopping cart to satisfy tests" \  
  --test-driven true \  
  --tests tests/cart.test.js
```

7.4.2.3 Regression Test Generation

```
# Analyze code changes and generate regression tests  
opencode-setup agents execute tester \  
  --task "Generate regression tests for recent changes" \  
  --changes "git diff HEAD~1 HEAD" \  
  --focus "critical_paths" \  
  --output tests/regression/
```

7.5 Input and Output Examples

7.5.1 Input Examples

7.5.1.1 Simple Request

Create unit tests for user authentication service

7.5.1.2 Detailed Request

Create comprehensive tests for e-commerce shopping cart system:

Unit Tests:

- Cart item addition/removal
- Price calculation with discounts
- Inventory validation
- Coupon code validation

Integration Tests:

- Cart API endpoints
- Payment integration
- Inventory service integration

- User session management

E2E Tests:

- Complete shopping flow
- Multi-item cart management
- Payment processing
- Order confirmation

Requirements:

- Jest for unit tests
- Supertest for API tests
- Cypress for E2E tests
- 90% code coverage target
- Performance testing for cart operations

7.5.1.3 API Testing Specification

Create integration tests for user management API:

Endpoints to test:

POST /api/users/register

- Valid registration
- Duplicate email
- Invalid email format
- Weak password rejection
- Missing required fields

POST /api/users/login

- Valid credentials
- Invalid email
- Invalid password
- Non-existent user
- Account locked

GET /api/users/profile

- Authenticated user
- Invalid token
- Expired token

PUT /api/users/profile

- Valid update
- Invalid data
- Unauthorized access

Requirements:

- Use Supertest framework

- Test database with fixtures
- Mock external services
- Response time assertions
- Status code validation

7.5.2 Output Examples

7.5.2.1 Complete Unit Test Suite

```
// tests/services/userService.test.js
const userService = require('../../src/services/userService');
const { User } = require('../../src/models');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');

// Mock dependencies
jest.mock('../../src/models');
jest.mock('bcrypt');
jest.mock('jsonwebtoken');

describe('UserService', () => {
  beforeEach(() => {
    jest.clearAllMocks();
  });

  describe('registerUser', () => {
    const validUserData = {
      email: 'test@example.com',
      password: 'SecurePass123!',
      name: 'Test User'
    };

    it('should register a new user successfully', async () => {
      // Arrange
      const hashedPassword = 'hashedPassword';
      const createdUser = { id: 1, ...validUserData, password: hashedPassword };

      User.findOne.mockResolvedValue(null);
      bcrypt.hash.mockResolvedValue(hashedPassword);
      User.create.mockResolvedValue(createdUser);

      // Act
      const result = await userService.registerUser(validUserData);

      // Assert
      expect(User.findOne).toHaveBeenCalledWith({
```

```

        where: { email: validUserData.email }
    });
    expect(bcrypt.hash).toHaveBeenCalledWith(validUserData.password, 10);
    expect(User.create).toHaveBeenCalledWith({
        ...validUserData,
        password: hashedPassword
    });
    expect(result).toEqual(createdUser);
});

it('should throw error if email already exists', async () => {
    // Arrange
    User.findOne.mockResolvedValue({ id: 1, email: validUserData.email });

    // Act & Assert
    await expect(userService.registerUser(validUserData))
        .rejects.toThrow('Email already exists');
});

it('should throw error for invalid email format', async () => {
    // Arrange
    const invalidUserData = { ...validUserData, email: 'invalid-email' };

    // Act & Assert
    await expect(userService.registerUser(invalidUserData))
        .rejects.toThrow('Invalid email format');
});

it('should throw error for weak password', async () => {
    // Arrange
    const weakPasswordData = { ...validUserData, password: '123' };

    // Act & Assert
    await expect(userService.registerUser(weakPasswordData))
        .rejects.toThrow('Password must be at least 8 characters long');
});

describe('authenticateUser', () => {
    const validCredentials = {
        email: 'test@example.com',
        password: 'SecurePass123!'
    };

    it('should authenticate user with valid credentials', async () => {
        // Arrange

```

```

const user = {
  id: 1,
  email: validCredentials.email,
  password: 'hashedPassword',
  name: 'Test User'
};
const token = 'jwt-token';

User.findOne.mockResolvedValue(user);
bcrypt.compare.mockResolvedValue(true);
jwt.sign.mockReturnValue(token);

// Act
const result = await userService.authenticateUser(validCredentials);

// Assert
expect(User.findOne).toHaveBeenCalledWith({
  where: { email: validCredentials.email }
});
expect(bcrypt.compare).toHaveBeenCalledWith(
  validCredentials.password,
  user.password
);
expect(jwt.sign).toHaveBeenCalledWith(
  { userId: user.id, email: user.email },
  process.env.JWT_SECRET,
  { expiresIn: '24h' }
);
expect(result).toEqual({ user, token });
});

it('should throw error for non-existent user', async () => {
  // Arrange
  User.findOne.mockResolvedValue(null);

  // Act & Assert
  await expect(userService.authenticateUser(validCredentials))
    .rejects.toThrow('Invalid credentials');
});

it('should throw error for incorrect password', async () => {
  // Arrange
  const user = {
    id: 1,
    email: validCredentials.email,
    password: 'hashedPassword'
  };

```

```

    };

    User.findOne.mockResolvedValue(user);
    bcrypt.compare.mockResolvedValue(false);

    // Act & Assert
    await expect(userService.authenticateUser(validCredentials))
      .rejects.toThrow('Invalid credentials');
  });
});

describe('getUserProfile', () => {
  const userId = 1;

  it('should return user profile for valid user ID', async () => {
    // Arrange
    const userProfile = {
      id: userId,
      email: 'test@example.com',
      name: 'Test User',
      createdAt: new Date()
    };

    User.findByPk.mockResolvedValue(userProfile);

    // Act
    const result = await userService.getUserProfile(userId);

    // Assert
    expect(User.findByPk).toHaveBeenCalledWith(userId, {
      attributes: { exclude: ['password'] }
    });
    expect(result).toEqual(userProfile);
  });

  it('should throw error for non-existent user', async () => {
    // Arrange
    User.findByPk.mockResolvedValue(null);

    // Act & Assert
    await expect(userService.getUserProfile(userId))
      .rejects.toThrow('User not found');
  });
});
});

```

7.5.2.2 API Integration Tests

```
// tests/integration/users.test.js
const request = require('supertest');
const app = require('../../src/app');
const { User } = require('../../src/models');
const { generateToken } = require('../../src/utils/auth');

describe('User API Integration Tests', () => {
  let testUser;
  let authToken;

  beforeAll(async () => {
    // Setup test database
    await User.sync({ force: true });

    // Create test user
    testUser = await User.create({
      email: 'test@example.com',
      password: 'hashedPassword',
      name: 'Test User'
    });

    authToken = generateToken(testUser.id);
  });

  afterAll(async () => {
    // Cleanup test database
    await User.drop();
  });

  describe('POST /api/users/register', () => {
    it('should register a new user', async () => {
      const newUser = {
        email: 'newuser@example.com',
        password: 'SecurePass123!',
        name: 'New User'
      };

      const response = await request(app)
        .post('/api/users/register')
        .send(newUser)
        .expect(201);

      expect(response.body).toHaveProperty('id');
      expect(response.body.email).toBe(newUser.email);
    });
  });
});
```

```

    expect(response.body.name).toBe(newUser.name);
    expect(response.body).not.toHaveProperty('password');
  });

  it('should return 400 for duplicate email', async () => {
    const duplicateUser = {
      email: testUser.email,
      password: 'SecurePass123!',
      name: 'Duplicate User'
    };

    const response = await request(app)
      .post('/api/users/register')
      .send(duplicateUser)
      .expect(400);

    expect(response.body.error).toBe('Email already exists');
  });

  it('should return 400 for invalid email format', async () => {
    const invalidUser = {
      email: 'invalid-email',
      password: 'SecurePass123!',
      name: 'Invalid User'
    };

    const response = await request(app)
      .post('/api/users/register')
      .send(invalidUser)
      .expect(400);

    expect(response.body.error).toBe('Invalid email format');
  });

  it('should return 400 for weak password', async () => {
    const weakPasswordUser = {
      email: 'weak@example.com',
      password: '123',
      name: 'Weak Password User'
    };

    const response = await request(app)
      .post('/api/users/register')
      .send(weakPasswordUser)
      .expect(400);
  });

```

```

    expect(response.body.error).toBe('Password must be at least 8 characters long');
  });
});

describe('POST /api/users/login', () => {
  it('should login with valid credentials', async () => {
    const credentials = {
      email: testUser.email,
      password: 'SecurePass123!'
    };

    const response = await request(app)
      .post('/api/users/login')
      .send(credentials)
      .expect(200);

    expect(response.body).toHaveProperty('token');
    expect(response.body.user.email).toBe(testUser.email);
    expect(response.body.user).not.toHaveProperty('password');
  });

  it('should return 401 for invalid email', async () => {
    const credentials = {
      email: 'nonexistent@example.com',
      password: 'SecurePass123!'
    };

    const response = await request(app)
      .post('/api/users/login')
      .send(credentials)
      .expect(401);

    expect(response.body.error).toBe('Invalid credentials');
  });

  it('should return 401 for invalid password', async () => {
    const credentials = {
      email: testUser.email,
      password: 'wrongpassword'
    };

    const response = await request(app)
      .post('/api/users/login')
      .send(credentials)
      .expect(401);
  });
});

```

```

    expect(response.body.error).toBe('Invalid credentials');
  });
});

describe('GET /api/users/profile', () => {
  it('should return user profile with valid token', async () => {
    const response = await request(app)
      .get('/api/users/profile')
      .set('Authorization', `Bearer ${authToken}`)
      .expect(200);

    expect(response.body.email).toBe(testUser.email);
    expect(response.body.name).toBe(testUser.name);
    expect(response.body).not.toHaveProperty('password');
  });

  it('should return 401 without token', async () => {
    const response = await request(app)
      .get('/api/users/profile')
      .expect(401);

    expect(response.body.error).toBe('Access token required');
  });

  it('should return 401 with invalid token', async () => {
    const response = await request(app)
      .get('/api/users/profile')
      .set('Authorization', 'Bearer invalid-token')
      .expect(401);

    expect(response.body.error).toBe('Invalid token');
  });
});

describe('PUT /api/users/profile', () => {
  it('should update user profile with valid data', async () => {
    const updateData = {
      name: 'Updated Name',
      email: 'updated@example.com'
    };

    const response = await request(app)
      .put('/api/users/profile')
      .set('Authorization', `Bearer ${authToken}`)
      .send(updateData)
      .expect(200);
  });
});

```

```

        expect(response.body.name).toBe(updateData.name);
        expect(response.body.email).toBe(updateData.email);
    });

    it('should return 400 for invalid email format', async () => {
        const updateData = {
            email: 'invalid-email'
        };

        const response = await request(app)
            .put('/api/users/profile')
            .set('Authorization', `Bearer ${authToken}`)
            .send(updateData)
            .expect(400);

        expect(response.body.error).toBe('Invalid email format');
    });

    it('should return 401 without authentication', async () => {
        const updateData = {
            name: 'Updated Name'
        };

        const response = await request(app)
            .put('/api/users/profile')
            .send(updateData)
            .expect(401);

        expect(response.body.error).toBe('Access token required');
    });
});
});

```

7.5.2.3 E2E Test with Cypress

```

// cypress/e2e/user-journey.cy.js
describe('User Registration and Login Journey', () => {
    beforeEach(() => {
        // Visit the application
        cy.visit('/');
    });

    it('should complete full user registration and login flow', () => {
        const user = {
            name: 'John Doe',

```

```

    email: 'john.doe@example.com',
    password: 'SecurePass123!'
  };

  // Navigate to registration
  cy.get('[data-cy=register-button]').click();
  cy.url().should('include', '/register');

  // Fill registration form
  cy.get('[data-cy=name-input]').type(user.name);
  cy.get('[data-cy=email-input]').type(user.email);
  cy.get('[data-cy=password-input]').type(user.password);
  cy.get('[data-cy=confirm-password-input]').type(user.password);

  // Submit registration
  cy.get('[data-cy=register-submit]').click();

  // Verify successful registration
  cy.url().should('include', '/dashboard');
  cy.get('[data-cy=welcome-message]')
    .should('contain', `Welcome, ${user.name}`);

  // Logout
  cy.get('[data-cy=logout-button]').click();
  cy.url().should('include', '/login');

  // Login with new credentials
  cy.get('[data-cy=email-input]').type(user.email);
  cy.get('[data-cy=password-input]').type(user.password);
  cy.get('[data-cy=login-submit]').click();

  // Verify successful login
  cy.url().should('include', '/dashboard');
  cy.get('[data-cy=user-profile]').should('contain', user.name);
});

it('should handle registration validation errors', () => {
  // Try to submit empty form
  cy.get('[data-cy=register-button]').click();
  cy.get('[data-cy=register-submit]').click();

  // Verify validation errors
  cy.get('[data-cy=name-error]').should('be.visible');
  cy.get('[data-cy=email-error]').should('be.visible');
  cy.get('[data-cy=password-error]').should('be.visible');
});

```

```

    // Try invalid email
    cy.get('[data-cy=name-input]').type('Test User');
    cy.get('[data-cy=email-input]').type('invalid-email');
    cy.get('[data-cy=password-input]').type('password123');
    cy.get('[data-cy=confirm-password-input]').type('password123');

    cy.get('[data-cy=register-submit]').click();

    // Verify email validation error
    cy.get('[data-cy=email-error']')
      .should('contain', 'Invalid email format');
  });

  it('should handle login errors', () => {
    // Navigate to login
    cy.get('[data-cy=login-button]').click();
    cy.url().should('include', '/login');

    // Try login with invalid credentials
    cy.get('[data-cy=email-input]').type('nonexistent@example.com');
    cy.get('[data-cy=password-input]').type('wrongpassword');
    cy.get('[data-cy=login-submit]').click();

    // Verify error message
    cy.get('[data-cy=login-error']')
      .should('contain', 'Invalid credentials');
  });
});

describe('Shopping Cart Functionality', () => {
  beforeEach(() => {
    // Login before each test
    cy.visit('/login');
    cy.get('[data-cy=email-input]').type('test@example.com');
    cy.get('[data-cy=password-input]').type('password123');
    cy.get('[data-cy=login-submit]').click();
    cy.url().should('include', '/dashboard');
  });

  it('should add items to cart and complete checkout', () => {
    const products = [
      { name: 'Product 1', price: 29.99 },
      { name: 'Product 2', price: 49.99 }
    ];

    // Add products to cart

```

```

products.forEach(product => {
  cy.visit('/products');
  cy.contains(product.name).click();
  cy.get('[data-cy=add-to-cart]').click();
  cy.get('[data-cy=cart-count]')
    .should('contain', products.indexOf(product) + 1);
});

// View cart
cy.get('[data-cy=cart-button]').click();
cy.url().should('include', '/cart');

// Verify cart contents
cy.get('[data-cy=cart-item]').should('have.length', 2);
cy.get('[data-cy=cart-total]')
  .should('contain', (29.99 + 49.99).toFixed(2));

// Proceed to checkout
cy.get('[data-cy=checkout-button]').click();
cy.url().should('include', '/checkout');

// Fill checkout form
cy.get('[data-cy=shipping-address]').type('123 Main St');
cy.get('[data-cy=shipping-city]').type('Anytown');
cy.get('[data-cy=shipping-zip]').type('12345');

// Complete checkout
cy.get('[data-cy=place-order]').click();

// Verify order confirmation
cy.url().should('include', '/order-confirmation');
cy.get('[data-cy=order-success]')
  .should('contain', 'Order placed successfully');
});
});

```

7.6 Best Practices for Tester Agent

7.6.1 Providing Good Input

7.6.1.1 Good Input Examples

Create comprehensive tests for user authentication:

- Unit tests for validation logic
- Integration tests for API endpoints
- E2E tests for user flows
- Performance tests for login endpoint

- Target 90% code coverage

7.6.1.2 Poor Input Examples

Test the user service

7.6.2 Structuring Test Requirements

Use the following structure for best results:

Test Scope

[What needs to be tested]

Test Types Required

[Unit, Integration, E2E, Performance, etc.]

Test Requirements

[Specific test cases and scenarios]

Framework Preferences

[Testing frameworks and tools]

Coverage Requirements

[Code coverage targets and quality gates]

Performance Requirements

[Performance benchmarks and load requirements]

7.6.3 Reviewing Test Output

When reviewing the Tester agent's output:

1. **Coverage:** Does it test all important scenarios?
2. **Quality:** Are the tests well-structured and maintainable?
3. **Edge Cases:** Are edge cases properly covered?
4. **Performance:** Are performance requirements tested?
5. **Integration:** Do integration tests work properly?
6. **Data:** Is test data properly managed?

7.7 Integration with Other Agents

7.7.1 Typical Workflow Sequence

1. **Implementer:** Implement the feature/code
2. **Tester:** Create comprehensive tests
3. **Reviewer:** Review both code and tests
4. **Debugger:** Fix any failing tests
5. **Documenter:** Document testing approach

7.7.2 Handoff Patterns

7.7.2.1 Tester → Reviewer The Tester provides: - Complete test suite - Test coverage reports - Performance benchmarks - Quality metrics

7.7.2.2 Tester → Debugger The Tester provides: - Failing test cases - Error logs and diagnostics - Reproduction steps - Expected vs actual behavior

7.8 Common Use Cases

7.8.1 Use Case 1: API Testing

Input: “Create comprehensive tests for REST API”

Output: - Unit tests for business logic - Integration tests for API endpoints - Contract testing for API contracts - Performance tests for critical endpoints

7.8.2 Use Case 2: Component Testing

Input: “Test React component library”

Output: - Component unit tests - Accessibility tests - Visual regression tests - Interaction tests

7.8.3 Use Case 3: Performance Testing

Input: “Load test e-commerce checkout flow”

Output: - Load test scenarios - Performance benchmarks - Bottleneck analysis - Scalability tests

7.9 Troubleshooting

7.9.1 Common Issues

7.9.1.1 Issue: Insufficient Test Coverage Problem: Tests don’t cover important scenarios **Solution:** Specify detailed test cases and edge cases

7.9.1.2 Issue: Flaky Tests Problem: Tests are unreliable and fail intermittently **Solution:** Request proper test isolation and deterministic behavior

7.9.1.3 Issue: Slow Tests Problem: Test suite takes too long to run **Solution:** Request test optimization and parallelization

7.9.1.4 Issue: Complex Test Setup Problem: Tests require complex setup and teardown **Solution:** Request better test fixtures and utilities

7.9.2 Getting Better Results

1. **Specify Test Types:** Clearly define what types of tests are needed
2. **Provide Examples:** Show expected test scenarios and edge cases
3. **Define Coverage:** Specify coverage targets and quality gates
4. **Include Performance:** Define performance requirements and benchmarks
5. **Request Automation:** Ask for CI/CD integration and automation

7.10 Advanced Features

7.10.1 Test Coverage Analysis

```
# Analyze test coverage gaps
opencode-setup agents execute tester \
  --task "Analyze test coverage gaps" \
  --source src/ \
  --tests tests/ \
  --target 90 \
  --output coverage-report.html
```

7.10.2 Performance Benchmarking

```
# Create performance benchmarks
opencode-setup agents execute tester \
  --task "Create performance benchmarks for API endpoints" \
  --baseline "current_performance.json" \
  --threshold "response_time < 200ms" \
  --output benchmarks/
```

7.10.3 Test Data Generation

```
# Generate realistic test data
opencode-setup agents execute tester \
  --task "Generate test data for user management" \
  --count 1000 \
  --realistic true \
  --output fixtures/users.json
```

7.11 Summary

The Tester agent is your quality assurance expert for:

- **Comprehensive Testing:** Creating thorough test suites across all levels
- **Quality Assurance:** Establishing quality standards and gates
- **Test Automation:** Setting up automated testing pipelines
- **Performance Testing:** Ensuring system performance meets requirements

By providing detailed testing requirements and quality criteria, you can leverage the Tester agent to create robust, reliable test suites that ensure your software meets the highest quality standards.

The key to success with the Tester agent is providing clear testing requirements, specifying coverage targets, and defining quality gates that align with your project's quality standards.

8 Chapter The Debugger Agent

This chapter is currently being written and will be available in the next release.

8.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

8.2 Preview

The The Debugger Agent chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

9 Chapter The Reviewer Agent

This chapter is currently being written and will be available in the next release.

9.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

9.2 Preview

The The Reviewer Agent chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

10 Chapter The Refactorer Agent

This chapter is currently being written and will be available in the next release.

10.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

10.2 Preview

The The Refactorer Agent chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

11 Chapter The Documenter Agent

This chapter is currently being written and will be available in the next release.

11.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

11.2 Preview

The The Documenter Agent chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

12 Chapter The Researcher Agent

This chapter is currently being written and will be available in the next release.

12.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

12.2 Preview

The The Researcher Agent chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

13 Chapter Workflow Orchestration

This chapter is currently being written and will be available in the next release.

13.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

13.2 Preview

The Workflow Orchestration chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

14 Chapter Command Reference

This chapter is currently being written and will be available in the next release.

14.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

14.2 Preview

The Command Reference chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

15 Chapter Custom Commands

This chapter is currently being written and will be available in the next release.

15.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

15.2 Preview

The Custom Commands chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

16 Chapter Integration Strategies

This chapter is currently being written and will be available in the next release.

16.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

16.2 Preview

The Integration Strategies chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

17 Chapter Performance Optimization

This chapter is currently being written and will be available in the next release.

17.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

17.2 Preview

The Performance Optimization chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

18 Chapter Security Considerations

This chapter is currently being written and will be available in the next release.

18.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

18.2 Preview

The Security Considerations chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

19 Chapter Quick Reference Guide

This chapter is currently being written and will be available in the next release.

19.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

19.2 Preview

The Quick Reference Guide chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

20 Chapter Best Practices

This chapter is currently being written and will be available in the next release.

20.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

20.2 Preview

The Best Practices chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.

21 Chapter Troubleshooting

This chapter is currently being written and will be available in the next release.

21.1 Coming Soon

This chapter will cover:

- Detailed concepts and examples
- Practical implementation guidance
- Best practices and tips
- Integration with other agents

21.2 Preview

The Troubleshooting chapter is part of our comprehensive guide to Open Code Agents. It will provide in-depth information about this topic to help you master the Open Code Agents ecosystem.

This is a development preview. The complete chapter will be available in the final release.