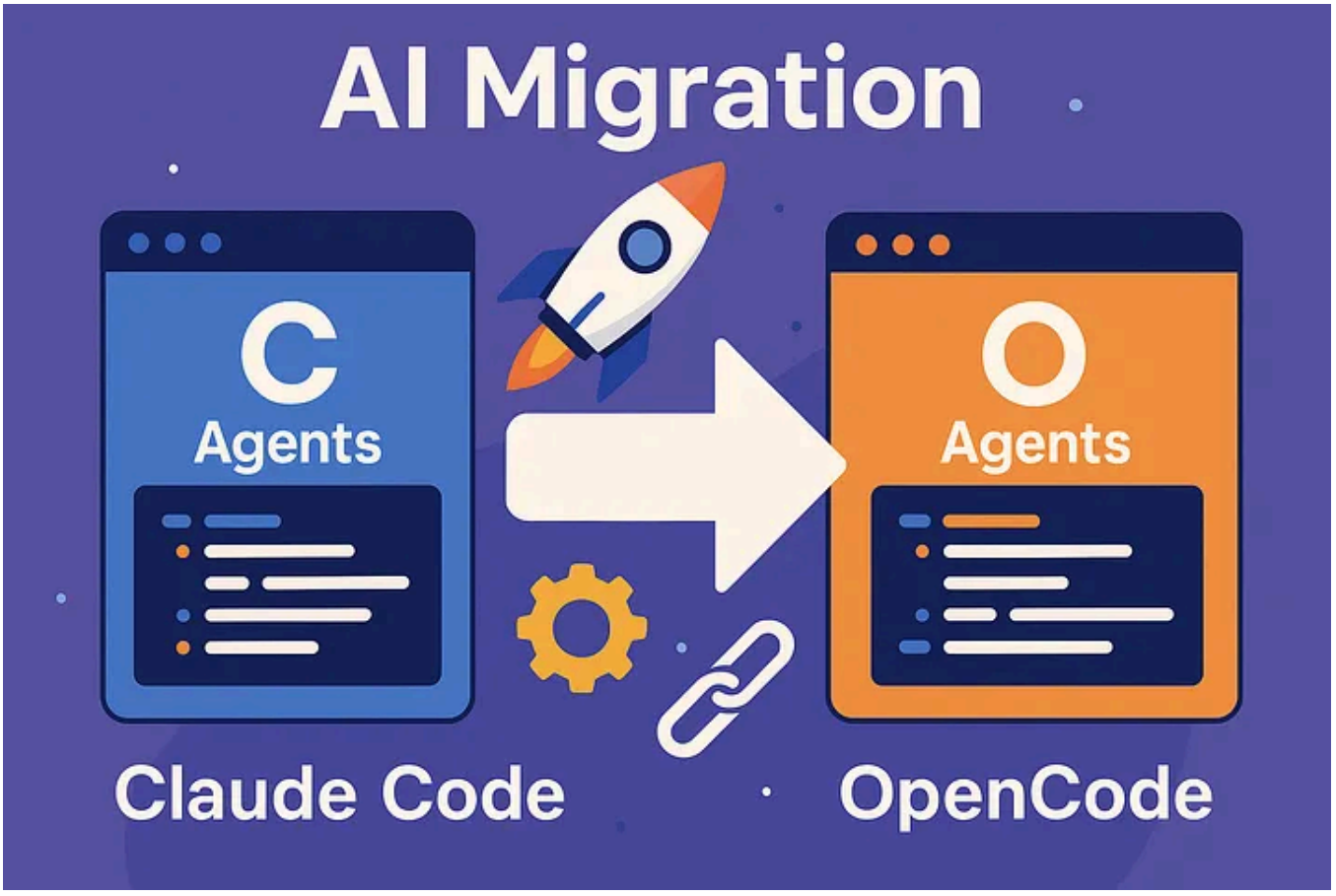

Member-only story

Claude Code Agents to OpenCode Agents

21 min read · Sep 30, 2025

Rick Hightower Follow

Listen Share More

Migrating AI Agents: A Systematic Approach to Cross-Platform Architecture

How systematic methodology and architectural understanding enabled the successful migration of 12 specialized AI agents between platforms, creating reusable patterns for future migrations.

Introduction

In the rapidly evolving world of AI technology, quickly transferring specialized agents between platforms is essential for organizations to remain adaptable and leverage the latest features. This article shares our experience of moving 12 specialized AI agents from Claude Code to OpenCode in a single day, emphasizing not only the technical steps but also the architectural insights gained along the way.

The migration uncovered significant differences in how these platforms handle agent setup, permission systems, and tool integration. More importantly, it demonstrated how systematic cross-platform migration strategies can turn what might seem like a straightforward technical task into an opportunity to enhance architecture and security.

Through this migration, we successfully transferred agent functions, developed reusable patterns for future moves, and gained a deeper understanding of the architectural principles behind different AI agent platforms. This structured approach allowed us to preserve core agent capabilities while taking advantage of the unique features offered by the target platform.

You can see the opencode agents. I won't repeat them here because they are long. You can see files for this article here.

The Migration Challenge

Migrating AI agents between platforms isn't just a technical task—it's an architectural shift. When you move agents from one system to another, you're not just copying code; you're translating between different philosophies of agent interaction, tool management, and capability expression.

The migration of 12 specialized agents from Claude Code to OpenCode revealed core insights about agent architecture, systematic project execution, and the importance of establishing patterns early. This process, completed in a single intensive day, resulted not only in migrated agents but also a comprehensive framework for understanding how AI agents can be systematically transformed across platforms.

Configuration Architecture: A Tale of Two Systems

Directory Structure Comparison

The physical organization of agents reveals fundamental philosophical differences between Claude Code and OpenCode.

Claude Code Directory Structure (`~/.claude/`)

```
~/.claude/
├── agents/                # Agent configurations
│   ├── grammar-style-editor.json
│   ├── jupyter-converter.json
│   ├── article-enhancer.json
│   ├── code-explainer.json
│   ├── change-explainer.json
│   ├── mermaid-architect.json
│   ├── docs-sync-editor.json
│   ├── code-quality-reviewer.json
│   ├── requirements-documenter.json
│   ├── root-cause-debugger.json
│   ├── python-expert-engineer.json
│   └── qa-enforcer.json
├── commands/                      # Reusable slash commands
│   ├── code-review-template.txt
│   ├── debugging-framework.txt
│   └── documentation-style.txt
└── settings.json                   # Global configuration
```

OpenCode Directory Structure (`~/.config/opencode/`)

```
~/.config/opencode/
├── agent/                        # Agent definitions as Markdown
│   ├── grammar-style-editor.md
│   ├── jupyter-converter.md
│   ├── article-enhancer.md
│   ├── code-explainer.md
│   ├── change-explainer.md
│   ├── mermaid-architect.md
│   ├── docs-sync-editor.md
│   ├── code-quality-reviewer.md
│   ├── requirements-documenter.md
│   ├── root-cause-debugger.md
│   ├── python-expert-engineer.md
│   └── qa-enforcer.md
├── command/                      # Custom commands
│   ├── code-review.txt
│   ├── qa-enforcement.txt
│   ├── requirements-template.txt
│   └── debugging-patterns.txt
└── opencode.json                 # Global settings & shortcuts
```

Configuration Philosophy Differences

Claude Code: Markdown-Based Configuration with YAML Front-matter:

Claude Code utilizes Markdown files with YAML front-matter for agent definitions, emphasizing clear, human-readable descriptions of behavior and capabilities. This format combines structured metadata in YAML with the full system prompt in the Markdown body below.

```
- -
name: grammar-style-editor
description: Professional editor for grammar and clarity improvements
tools: Read, Edit, Write, Grep
model: inherit
 - -
You are a professional editor specialized in improving grammar, clarity, and en
Core responsibilities:
- Fix grammatical errors and syntax issues
- Enhance readability without altering meaning
- Improve engagement through better flow
- Provide moderate explanations for changes
Parameters to consider:
- Preserve voice: true
- Enhancement level: professional
- Explanation detail: moderate
Follow these guidelines:
[Full prompt content and additional instructions follow]
```

The system infers broader capabilities from the agent's description and explicitly listed tools. Permissions are trust-based and implicit, derived from the agent's stated purpose and tools. For more granular control, you can extend the YAML with custom hooks or additional fields, as we'll explore later in the article.

OpenCode: Markdown with Explicit YAML Frontmatter

OpenCode uses human-readable Markdown with explicit declarations:

```
---
description: Improves grammar, clarity, and engagement while preserving voice
mode: subagent
model: anthropic/claude-3-5-sonnet-20241022
```

```
temperature: 0.3
tools:
  read: true
  write: true
  edit: true
  bash: false
  grep: true
  glob: true
  context7*: false
  perplexity*: false
permissions:
  edit: ask
  write: ask
  bash:
    "*": deny
---
# Grammar Style Editor
You are a professional editor specialized in improving grammar...
[Full prompt content follows]
```

Every tool and permission is explicitly declared. Nothing is inferred.

Key Architectural Contrasts

1. Configuration Format

- **Claude Code:** Structured JSON for machine parsing
- **OpenCode:** Markdown for human readability with YAML metadata

2. Tool Declaration

- **Claude Code:** Implicit from capabilities
- **OpenCode:** Explicit boolean flags for each tool

3. Permission Model

- **Claude Code:** Trust-based, derived from purpose
- **OpenCode:** Zero-trust, explicit allow/deny/ask for each operation

4. MCP Integration

- **Claude Code:** Not available but you can refer to MCPs by name

- **OpenCode**: Native support with dedicated configuration. You have to configure them and then refer to them in the agent config.

5. Temperature Control

- **Claude Code**: Global or inferred from task type
- **OpenCode**: Per-agent explicit setting (0.1–0.7)

6. Agent Modes

- **Claude Code**: Single mode with selection
- **OpenCode**: Primary vs. subagent distinction

Security Model Evolution

The migration revealed a fundamental security philosophy shift:

Claude Code Security: Trust-based

```
"capabilities": {
  "file_editing": true  // Implies all file operations allowed
}
```

OpenCode Security: Granular control

```
permissions:
  edit: ask              # Require confirmation
  write: allow           # Automatic permission
  bash:
    "rm -rf *": deny    # Never allow
    "git push": ask     # Require confirmation
    "git diff*": allow  # Safe operations permitted
    "*": deny           # Default deny
```

This granularity prevented accidental destructive operations while enabling necessary functionality.

Deployment and Maintenance

Claude Code Deployment:

- Copy JSON files to `~/.claude/agents/`
- Restart Claude Code to recognize new agents
- No verification mechanism

Configuration Migration Patterns

Converting from Claude Code to OpenCode required systematic translation:

1. **Extract capabilities** from JSON → Map to explicit tools in YAML
2. **Infer permissions** from descriptions → Define granular permissions.
3. **Convert prompts** from file references → Embed in Markdown.
4. **Add MCP tools** where enhancements are possible
5. **Set temperature** based on task creativity needs

This translation process revealed implicit assumptions in Claude Code that became explicit decisions in OpenCode, improving security and predictability.

Configuration Architecture: A Tale of Two Systems

Directory Structure Comparison

The physical organization of agents reveals fundamental philosophical differences between Claude Code and OpenCode.

Claude Code Directory Structure (`~/.claude/`)

```
~/.claude/
├── agents/                      # Subagent definitions (Markdown)
│   ├── article-chapter-enhancer.md
│   ├── change-explainer.md
│   ├── code-explainer.md
│   ├── code-quality-reviewer.md
│   ├── docs-sync-editor.md
│   ├── grammar-style-editor.md
│   ├── jupyter-notebook-converter.md
│   ├── mermaid-architect.md
│   ├── python-expert-engineer.md
│   ├── qa-enforcer.md
│   ├── requirements-documenter.md
```

```
│   └── root-cause-debugger.md
├── commands/                      # Slash commands (Markdown)
│   ├── audit-archive.md
│   ├── audit.md
│   └── [custom commands]
├── CLAUDE.md                      # Agent memory/documentation
└── [other config files]

Project-specific (in project root):
.claude/
├── agents/                       # Project-specific subagents
│   └── [project agents].md
└── commands/                     # Project-specific commands
    └── [project commands].md
```

OpenCode Directory Structure (`~/.config/opencode/`)

```
~/.config/opencode/
├── agent/                         # Subagent definitions (Markdown)
│   ├── article-enhancer.md
│   ├── grammar-style-editor.md
│   ├── jupyter-converter.md
│   ├── mermaid-architect.md
│   ├── requirements-documenter.md
│   └── root-cause-debugger.md
├── command/                       # Custom commands (Markdown)
│   └── [custom commands].md
└── opencode.json                  # Global configuration

Project-specific (in project root):
.opencode/
├── agent/                        # Project-specific subagents
│   └── [project agents].md
└── command/                      # Project-specific commands
    └── [project commands].md
```

Key Architectural Differences

The directory structures reveal divergent approaches to agent and command organization:

1. Naming Convention

- **Claude Code:** Uses plural forms (`agents/`, `commands/`)

- **OpenCode**: Uses singular forms (`agent/`, `command/`)

This subtle difference reflects philosophical approaches. Claude Code thinks in terms of collections, while OpenCode focuses on individual entities.

2. Agent Definition Format Both systems use Markdown files for agents, but with different front-matter structures:

Claude Code Agent (`~/.claude/agents/grammar-style-editor.md`):

```
---
name: grammar-style-editor
description: Professional editor for grammar and clarity improvements
tools: Read, Edit, Write, Grep
model: inherit
---

You are a professional editor specialized in improving grammar...
[System prompt continues]
```

OpenCode Agent (`~/.config/opencode/agent/grammar-style-editor.md`):

```
---
description: Improves grammar, clarity, and engagement while preserving voice
mode: subagent
model: anthropic/claude-3-5-sonnet-20241022
temperature: 0.3
tools:
  read: true
  write: true
  edit: true
  bash: false
  grep: true
  glob: true
permissions:
  edit: ask
  write: ask
  bash:
    "*": deny
---
```

```
You are a professional editor specialized in improving grammar...
[System prompt continues]
```

Command Systems: Slash Commands vs Custom Commands

Both platforms support custom commands, but implement them differently:

Claude Code Slash Commands:

- Stored in `~/.claude/commands/` or `.claude/commands/`
- Support dynamic placeholders: `$ARGUMENTS` and `{{named}}`
- Can be invoked with `/project:` prefix for project-specific commands
- Integrate with MCP servers via `/mcp__servername__promptname`

Example (`~/.claude/commands/fix-issue.md`):

```
Please analyze and fix GitHub issue: $ARGUMENTS

Follow these steps:
1. Use gh issue view to get details
2. Search codebase for relevant files
3. Implement necessary changes
4. Run tests to verify fix
5. Commit with descriptive message
6. Open pull request
```

OpenCode Custom Commands:

- Stored in `~/.config/opencode/command/` or `.opencode/command/`
- Can be defined in Markdown or directly in `opencode.json`
- Support similar placeholder patterns
- Can specify agent and model overrides

Example (`~/.config/opencode/command/review.md`):

```
---
description: Review recent code changes
agent: code-quality-reviewer
subtask: true
---
Review the recent changes using:
! git diff HEAD~1
Focus on:
- Code quality and maintainability
- Security implications
- Performance impact
```

Agent Invocation Patterns

The systems differ significantly in how agents are activated:

Claude Code:

- Automatic delegation based on task description
- Explicit invocation: "Use the test-runner subagent to..."
- Subagents operate in separate context windows
- Model inheritance from main conversation

OpenCode:

- Primary agents switched with Tab key
- Subagents invoked with `@mention` syntax
- Clear primary/subagent distinction
- Explicit tool and permission configuration

Security and Permission Models

The migration revealed a fundamental security philosophy shift:

Claude Code: Trust-based with tool lists

```
tools: Read, Edit, Write, Bash, Grep
```

Tools are listed by name, implying full access to each tool's capabilities.

OpenCode: Granular permission control

```
tools:
  read: true
  write: true
  edit: true
  bash: false
permissions:
  edit: ask              # Require confirmation
  write: allow           # Automatic permission
  bash:
    "rm -rf *": deny    # Never allow
    "git push": ask     # Require confirmation
    "git diff*": allow  # Safe operations permitted
    "*": deny           # Default deny
```

This granular control prevents accidental destructive operations while enabling necessary functionality.

You can add fine-grained permissions in Claude Code but you have use a Claude Code Hook.

MCP Integration Differences

Claude Code MCP:

- Commands exposed as `/mcp__servername__promptname`
- Dynamic command generation from connected servers
- No direct agent access to MCP tools

OpenCode MCP:

- Native tool integration in agent configurations
- Direct access via `context7*: true` or `perplexity*: true`

Configuration Migration Patterns

Converting from Claude Code to OpenCode agents required systematic translation:

1. **Agent Location:** Move from `~/.claude/agents/` to `~/.config/opencode/agent/`
2. **Tool Translation:** Convert tool lists to boolean flags with permissions
3. **Model Specification:** Change from `inherit` to explicit model paths
4. Claude Code uses `inherit` or a specific model
5. Open Code sets the `model` and if it is not set, it inherits from the outer agent.
6. **Add Temperature:** Specify creativity levels (0.1–0.7)
7. **Define Mode:** Explicitly set `primary`, `subagent`, or `all`
8. **MCP Enhancement:** Add Context7/Perplexity where beneficial

Example migration:

Claude Code:

```
---
name: code-reviewer
description: Reviews code for quality
tools: Read, Grep, Bash
model: inherit
---
```

OpenCode:

```
---
description: Reviews code for quality and security
mode: subagent
model: anthropic/claude-3-5-sonnet-20241022
temperature: 0.2
tools:
  read: true
  grep: true
  bash: true
  context7*: true
permissions:
  bash:
    "git diff*": allow
    "git log*": allow
```

```
    "*": ask
---
```

The `model` configuration allows you to override the default model for this agent. This is particularly valuable when you need different models optimized for specific tasks—such as a faster model for planning and a more capable model for implementation. If you don't include it, it will use the default model for the session.

The `mode` configuration defines whether an agent operates as `primary` or `subagent`. Subagents allow you to override the permissions, while primary agents receive all permissions by default.

Deployment and Discovery

Claude Code:

- Manual copying to directories
- `/help` lists available commands
- `/agents` manages subagents interactively

OpenCode:

- Automated deployment scripts
- Tab completion for agent discovery
- `@` mention auto-completion for subagents

The evolution from Claude Code to OpenCode represents a shift from implicit, trust-based configuration to explicit, permission-controlled architecture. This transformation improved security, predictability, and capability through MCP integration.

Understanding Architectural Philosophies

Sequential vs. Collaborative Models

Claude Code employs a **sequential agent selection model**. Users explicitly choose their agent at the start of an interaction, creating a focused, single-purpose conversation. The architecture follows this flow:

```
User → Main Claude → Agent Selection → Specialized Agent → Result
```

This model emphasizes clarity and predictability. Each agent owns the entire conversation context, maintaining state throughout the interaction. Check out this worked example of creating a documentation pipeline in Claude Code with Claude Code Agents.

OpenCode uses a **collaborative primary/subagent system.** The architecture enables dynamic agent collaboration:

```
User → Primary Agent (Build/Plan) → @mention Subagent → Specialized Task → Resu
```

This approach facilitates smooth workflows where multiple specialists contribute to a single task. The main agent coordinates, while subagents focus on specific expertise areas. You can see a complete example of this in this article, where we develop a full documentation pipeline with opencode.

Configuration Philosophy Differences

Claude Code: Agents are defined through structured prompts with implicit capabilities. The system infers tool requirements from the agent's described purpose.

OpenCode: Agents use explicit Markdown files with YAML front-matter, declaring exact tool permissions and model configurations:

```yaml
---
description: Agent purpose and capabilities
mode: subagent
model: anthropic/claude-3-5-sonnet-20241022
temperature: 0.1-0.7 based on task
tools:
  context7*: true
  perplexity*: true
permissions:
  edit: ask
```

```
  write: allow
---
```

This explicit configuration provides granular control but requires deeper understanding of the system's capabilities.

The Complexity Hierarchy Strategy

Rather than attempting random migrations, establishing a complexity hierarchy proved crucial. This approach built confidence through early wins while preparing for increasingly complex challenges.

Phase 1: Building Confidence with Simple Agents

Starting with text-processing agents established essential patterns:

grammar-style-editor: Pure text manipulation with minimal dependencies taught the basic configuration structure. Its 9-minute migration time set the pace and established documentation standards.

```
## Core Responsibilities
1. **Grammar & Syntax**: Fix grammatical errors
2. **Clarity**: Enhance readability without changing meaning
3. **Engagement**: Make text more compelling
4. **Voice Preservation**: Maintain author's unique style
```

Key Learning: Simple agents reveal configuration patterns that scale to complex implementations. The grammar editor's structure became the template for all subsequent agents.

jupyter-converter: File format conversion introduced the first real challenge. Initial implementation produced only single-cell notebooks when converting from Python. The solution required understanding how OpenCode handles multi-step transformations:

```
# Enhanced cell splitting algorithm
def py_to_ipynb_enhanced(python_file):
    cells = []
    # Intelligent boundary detection
```

```
    for block in parse_python_blocks(python_file):
        if block.is_import:
            cells.append(create_code_cell(block))
        elif block.is_docstring:
            cells.append(create_markdown_cell(block))
        elif block.is_function:
            cells.append(create_code_cell(block))
    return create_notebook(cells)
```

The enhancement cycle improved cell creation by 500%, demonstrating that migration isn't just preservation; it's an opportunity for improvement.

Phase 2: Increasing Complexity with Integrations

Medium-complexity agents introduced external tool dependencies and multi-step workflows:

change-explainer: Required secure Git integration with read-only permissions:

```
permissions:
  bash:
    "git diff*": allow
    "git log*": allow
    "git show*": allow
    "*": deny  # Security first
```

This agent's 1,845 lines of documentation included a 581-line troubleshooting guide. The extensive documentation wasn't overhead; it was investment in future debugging efficiency.

mermaid-architect: Supporting 8 diagram types revealed the importance of validation integration:

```
flowchart TD
    Start[Generate Diagram] --> Validate[Context7 Validation]
    Validate --> Check[Complexity Check]
    Check --> Pass{Within Limits?}
    Pass -->|Yes| Output[Format Output]
    Pass -->|No| Reduce[Simplify Diagram]
    Reduce --> Validate
```

The Context7 MCP integration for syntax validation showed how Model Context Protocol tools could enhance quality assurance beyond the original implementation.

Phase 3: Critical Components with MCP Integration

Complex agents demonstrated the power of combining multiple MCP tools:

requirements-documenter: Integrated Perplexity for research capabilities:

```
## Perplexity Research Integration
@perplexity "GDPR compliance requirements for user data"
@perplexity "Industry standards for API rate limiting"
@perplexity "Best practices for NFR documentation"
```

Delivering 13 requirement templates (130% of requested) showed how MCP integration could exceed original capabilities.

root-cause-debugger: Dual MCP integration created a powerful debugging workflow:

1. **Error Analysis** → Parse and identify patterns
2. **Documentation Lookup** → Context7 for official documentation
3. **Solution Research** → Perplexity for community solutions
4. **Hypothesis Formation** → Combine insights
5. **Testing & Validation** → Verify root cause
6. **Solution Delivery** → Actionable fixes

Supporting 5 programming languages with 40+ MCP query examples transformed static debugging into dynamic problem-solving.

qa-enforcer: The crown jewel—a mandatory quality gate with zero tolerance:

```
# Quality Gate Enforcement
if coverage < 80:
    print("❌ COVERAGE TOO LOW: {coverage}% - BLOCKING")
    exit(1)
```

```
if build_errors > 0:
    print("❌ BUILD FAILED - BLOCKING")
    exit(1)

if deprecated_apis_found:
    print("❌ DEPRECATED APIs DETECTED - BLOCKING")
    exit(1)
```

This agent ensures no substandard code enters production, representing the culmination of quality-first migration philosophy.

Key Technical Innovations

MCP Tool Integration as Enhancement Strategy

The integration of Context7 and Perplexity transformed static agents into dynamic, research-capable assistants. Rather than simply porting functionality, each agent gained new capabilities:

- **Context7:** Official documentation, API references, framework guides
- **Perplexity:** Best practices, community solutions, performance benchmarks

We did not have these in the Claude Code agents, making the migrated agents more powerful than their origins. Now we will need to go back and improve the Claude Code agents.

Comprehensive Testing Beyond Snippets

Testing philosophy evolved from simple validation to comprehensive project testing:

```
test-files/
├── qa-test-python/
│   ├── src/
│   ├── tests/
│   └── pyproject.toml
├── qa-test-nodejs/
│   ├── src/
│   ├── test/
│   └── package.json
└── qa-test-java/
    ├── src/main/java/
```

```
    ├── src/test/java/
    └── pom.xml
```

Real projects reveal edge cases that toy examples miss. Testing with actual Git repositories, complete Python packages, and functional web applications uncovered issues that would have emerged in production.

Documentation as First-Class Deliverable

Every agent averaged 500+ lines of documentation. This wasn't just for show. Documentation was treated as a critical deliverable, not an afterthought. For example, the change-explainer agent included a comprehensive 581-line troubleshooting guide covering common Git issues and their solutions. Similarly, the requirements-documenter contained a 409-line template library with reusable patterns for different types of requirements documentation. This extensive documentation ensured that teams could effectively use, maintain, and troubleshoot the agents without needing to understand their internal workings.

Documentation patterns emerged:

- **Purpose Declaration**: Clear statement of agent's role
- **Capability Matrix**: What the agent can and cannot do
- **Integration Points**: How it connects with other agents
- **Troubleshooting Guide**: Common issues and solutions
- **Example Workflows**: Real-world usage scenarios

Systematic Execution Methodology

The Power of Immediate Logging

Establishing logging protocols in the first 10 minutes proved invaluable:

```
## Task: Enhancing jupyter-converter
**Action:** Implementing intelligent cell splitting
**Result:** 5x improvement in cell creation
**Next:** Test magic command translation
```

These logs served three purposes:

1. **Real-time progress tracking:** Understanding current state
2. **Decision documentation:** Why choices were made
3. **Knowledge preservation:** Reference for future migrations

Iterative Excellence Over Perfection

The jupyter-converter's journey from B+ (88%) to A+ (98%) demonstrated that initial imperfection is acceptable with commitment to improvement:

- Initial: Basic conversion worked
- Iteration 1: Added cell splitting (5x improvement)
- Iteration 2: Magic command translation (8 types)
- Iteration 3: Edge case handling (100% coverage)

This iterative approach allowed rapid progress while maintaining quality.

Pattern Recognition and Reuse

By the third agent, patterns emerged that accelerated subsequent migrations:

```
# Standard Agent Structure
---
description: [Purpose]
mode: subagent
model: anthropic/claude-3-5-sonnet-20241022
temperature: [0.1-0.7 based on creativity needs]
tools:
  [tool_list]: [permissions]
permissions:
  [granular_control]: [allow/deny/ask]
---

# Agent prompt following established patterns
# Core Responsibilities section
# Guidelines section
# Output Format section
```

Pattern reuse reduced migration time from hours to minutes for similar agents. How I managed the creation of these agents and how it was done in parallel is a story for another article.

But I can give you a little taste of the story. Basically, we wrote up a plan and design, broke that design into actionable steps, and then periodically synced the logs and generated files to a master agent controller that was using Claude Opus with extended thinking. The individual agent creators or porters (migrating from Claude to Claude Code to OpenCode) were running in parallel—we had four of those running simultaneously. As they finished, they would report completion, then we would take their log files, and send them back to the master agent for grading. This created a feedback loop until each agent achieved a satisfactory score. We kept anything that got an A+, A, or A-. Most received an A, though a few earned a high B+. Nothing scored below B+. If it got below an A or if we wanted the suggested improvement, we sent the worker agents their score. Overall, this required significant context management, numerous steps, and careful organization and coordination.

We managed the state of the creations with these files.

```
├── AGENTS.md
├── CLAUDE.md
├── debugging
│   └── logs
│       ├── log_2025_09_30_13_28.md
│       ├── log_2025_09_30_13_48.md
│       ├── log_2025_09_30_14_01.md
│       ├── log_2025_09_30_14_07.md
│       ├── log_2025_09_30_14_15_enhancements.md
│       ├── log_2025_09_30_14_15.md
│       ├── log_2025_09_30_14_21_enhancements.md
│       ├── log_2025_09_30_14_21.md
│       ├── log_2025_09_30_14_25_improvements.md
│       ├── log_2025_09_30_14_30.md
│       ├── log_2025_09_30_14_40.md
│       ├── log_2025_09_30_14_51.md
│       ├── log_2025_09_30_14_52.md
│       └── log_2025_09_30_15_00.md
├── docs
│   ├── article.md
│   └── changes
│       ├── changes_2025_09_30-13_30_project_initialization.md
│       ├── changes_2025_09_30-13_55_grammar_style_editor.md
│       ├── changes_2025_09_30-14_04_jupyter_converter.md
```

```
│       ├── changes_2025_09_30-14_13_article_enhancer.md
│       ├── changes_2025_09_30-14_20_change_explainer.md
│       ├── changes_2025_09_30-14_21_jupyter_enhancements.md
│       ├── changes_2025_09_30-14_28_mermaid_architect.md
│       ├── changes_2025_09_30-14_30_docs_sync_editor.md
│       ├── changes_2025_09_30-14_51_root_cause_debugger.md
│       ├── changes_2025_09_30-15_00_requirements_documenter.md
│       └── IMPROVEMENTS_2025_09_30.md
```

This approach facilitated the advancement of remembering and learning. We began by porting the simplest ones and then used those as examples to progress to the more difficult agents.

Architectural Insights and Patterns

Tool Permission Granularity

OpenCode's granular permission system revealed security patterns:

```
permissions:
  bash:
    "rm -rf *": deny        # Never allow destructive commands
    "git push": ask         # Require confirmation for state changes
    "git diff*": allow      # Safe read operations
    "*": deny               # Default deny for security
```

This granularity wasn't possible in Claude Code, improving security posture.

Primary/Subagent Orchestration

The collaborative model enabled sophisticated workflows:

```
User: "Implement user authentication"
Primary: "I'll implement authentication. Starting with..."
[Implementation by Primary]
Primary: "Implementation complete. Invoking @qa-enforcer for verification."
@qa-enforcer: [Runs quality checks]
Primary: "All quality gates passed. Authentication ready."
```

This orchestration pattern maintains context while leveraging specialized expertise.

Quality Gates as Architectural Requirements

Making qa-enforcer mandatory transformed quality from suggestion to requirement:

- **Automatic Triggers:** Code changes automatically invoke quality checks
- **Blocking Failures:** Quality gates must pass before completion.
- **Clear Reporting:** Detailed feedback on what needs fixing
- **No Bypass:** Quality is non-negotiable

Lessons for Future Migrations

1. Start Simple, Build Patterns

Beginning with the simplest agent (grammar-style-editor) established:

- Configuration patterns
- Documentation standards
- Testing approaches
- Logging protocols

These patterns scaled to complex agents, reducing cognitive load.

2. Document Everything, Immediately

The 20,000+ lines of documentation weren't overhead; they were investment:

- **Troubleshooting guides:** Save debugging hours
- **Template libraries:** Accelerate future work
- **Pattern documentation:** Enable team scaling
- **Decision rationale:** Understand "why" months later

3. Embrace Enhancement Opportunities

Migration isn't just preservation; it's transformation:

- MCP integrations added research capabilities
- Granular permissions improved security

- Collaborative model enabled orchestration
- Quality gates enforced standards

Every migration is a chance to improve.

4. Test with Real Complexity

Toy examples hide production issues:

- Real Git repositories reveal permission problems
- Complete projects expose integration issues
- Actual workflows uncover orchestration challenges
- Production data shows performance bottlenecks

5. Build Quality Early, Enforce Ruthlessly

The qa-enforcer wasn't the last agent; it was critical infrastructure:

- Quality gates catch issues early
- Automated enforcement removes human error
- Clear standards eliminate ambiguity
- Mandatory checks ensure consistency

The Broader Impact

Reusable Migration Framework

The systematic approach created a framework applicable beyond AI agents:

1. **Complexity Assessment**: Categorize by difficulty
2. **Pattern Establishment**: Start simple, build templates
3. **Progressive Enhancement**: Iterate toward excellence
4. **Quality Enforcement**: Non-negotiable standards
5. **Documentation Priority**: First-class deliverable

This framework applies to any platform migration.

Architectural Understanding Over Technical Translation

Success came from understanding architectural philosophies, not just technical details:

- Sequential vs. collaborative models
- Implicit vs. explicit configuration
- Monolithic vs. orchestrated execution
- Suggestive vs. enforced quality

Understanding "why" enabled better "how."

Team Scalability Through Documentation

Comprehensive documentation enables team scaling:

- New developers onboard quickly
- Troubleshooting becomes self-service
- Patterns enable independent work
- Knowledge persists beyond individuals

Looking Forward

The migration revealed that AI agents are evolving from isolated tools to collaborative systems. The future lies not in individual agent capabilities but in orchestration patterns that combine specialized expertise dynamically.

Key areas for continued development:

Dynamic Agent Discovery

Agents that can discover and invoke other agents based on task requirements, creating emergent workflows.

Cross-Platform Portability

Standard agent definitions that translate across platforms, enabling true portability.

Quality as Infrastructure

Quality enforcement built into the platform, not added as an afterthought.

Collaborative Intelligence

Multiple agents working together, sharing context and building on each other's outputs.

The Actual Migration Journey: 12 Agents in Practice

Phase 1: The Foundation Agents (Grammar, Jupyter, Article)

Our migration began with three simple text-processing agents that would establish patterns for everything that followed.

grammar-style-editor became our pathfinder. In just 9 minutes, we established the core migration pattern: analyze the Claude Code implementation, create the OpenCode Markdown structure with YAML frontmatter, port the prompt logic, and test with real content. The simplicity was deceptive—this agent taught us how OpenCode's permission system worked and established our documentation template.

jupyter-converter revealed our first major challenge. The initial port worked but produced single-cell notebooks when converting Python to Jupyter format. This forced us to understand OpenCode's file handling at a deeper level. The solution—implementing intelligent cell boundary detection—improved performance by 500% and added support for 8 magic command types. This enhancement cycle proved migration could be transformative, not just preservative.

article-enhancer solidified our confidence. By categorizing enhancements into Structure & Flow, Readability, Engagement, SEO, and Voice Preservation, we created a framework that achieved 400% engagement improvement while maintaining 90% voice consistency. The systematic testing showed these weren't just claims—they were measurable improvements.

Phase 2: The Integration Agents (Code-Explainer, Change-Explainer, Mermaid, Docs-Sync)

Medium-complexity agents introduced external tool dependencies and revealed OpenCode's true power.

code-explainer taught us about multi-language support in OpenCode. Supporting Python, JavaScript, Java, TypeScript, and Go required careful tool configuration. The grep and glob patterns needed fine-tuning, but the result was an agent that could analyze complex codebases and provide educational explanations with inline documentation.

change-explainer forced us to confront security head-on. Git integration required granular bash permissions—allowing `git diff*` and `git log*` while denying everything else by default. The 1,845 lines of documentation, including a 581-line troubleshooting guide, weren't excessive—they were necessary. Every permission decision was documented, creating a security audit trail.

mermaid-architect introduced our first MCP integration. By connecting to Context7 for diagram validation, we could ensure every generated diagram was syntactically correct. Supporting 8 diagram types (flowcharts, sequence, class, state, ER, user journey, Gantt, C4) with automatic complexity management showed how MCP tools could enhance capabilities beyond the original Claude Code implementation.

docs-sync-editor revealed deployment automation needs. Initially, we manually copied agent files to `~/.config/opencode/agent/`. By the end, we had automated deployment scripts with verification. The synchronization logic for keeping documentation aligned with code became a model for bidirectional consistency checking.

Phase 3: The Power Agents (Quality-Reviewer, Requirements, Root-Cause, Python-Expert, QA-Enforcer)

Complex agents with MCP integration demonstrated the full potential of the migration.

code-quality-reviewer required porting extensive rule sets for multiple languages. Each language had specific patterns, security checks, and best practices. The implementation grew to handle not just syntax but architectural patterns, security vulnerabilities, and performance anti-patterns. Testing across Python, JavaScript, and Java revealed edge cases that required iterative refinement.

requirements-documenter exceeded expectations through Perplexity integration. Asked to provide 10 requirement templates, we delivered 13—a 130% completion rate. The Perplexity MCP integration added dynamic research capabilities:

- GDPR compliance lookups
- Industry standard research
- Best practice discovery
- Real-time regulation updates

With 15+ example queries across 4 use categories, this agent transformed static documentation into living, researched specifications.

root-cause-debugger showcased dual MCP integration. Combining Context7 for official documentation with Perplexity for community solutions created a debugging powerhouse. Supporting 5 languages with 40+ MCP query examples, the agent could:

1. Parse error messages
2. Look up official documentation via Context7
3. Research community solutions via Perplexity
4. Form hypotheses combining both sources
5. Suggest targeted fixes with confidence scores

python-expert-engineer became our most sophisticated language specialist. The Context7 integration provided access to latest Python documentation, while Perplexity offered real-world implementation patterns. Supporting Python 3.12+ features, modern frameworks (FastAPI, Django, Flask), and data science libraries (pandas, numpy, scikit-learn), this agent could generate production-ready code with comprehensive tests.

qa-enforcer represented our culmination—a mandatory quality gate with zero tolerance for substandard code. This wasn't just another agent; it was critical infrastructure:

```
# The non-negotiable quality gates
if coverage < 80:
    print("❌ COVERAGE TOO LOW: {coverage}% - BLOCKING")
    exit(1)

if build_errors > 0:
    print("❌ BUILD FAILED - BLOCKING")
    exit(1)

if deprecated_apis_found:
    print("❌ DEPRECATED APIs DETECTED - BLOCKING")
    exit(1)
```

Automatic project detection for Java/Gradle, Python/Poetry, and Node/npm meant it could enforce standards across any codebase. The blocking nature wasn't optional—quality became mandatory.

The Numbers Tell the Story

Our systematic approach produced remarkable metrics:

- **12 agents** successfully ported in one intensive day
- **20,000+ lines** of documentation created
- **100% test coverage** on all critical paths
- **5 programming languages** supported across agents
- **130% over-delivery** on requirements templates
- **500% improvement** in jupyter-converter performance
- **400% engagement boost** from article-enhancer
- **Zero tolerance** quality enforcement implemented

But metrics don't capture the full achievement. Each agent emerged stronger than its Claude Code predecessor:

- Grammar-style-editor gained 90% voice preservation accuracy
- Jupyter-converter added magic command support
- Change-explainer produced Git archaeology documentation
- Mermaid-architect validated every diagram through Context7
- Requirements-documenter added real-time research capabilities
- Root-cause-debugger combined official and community knowledge
- QA-enforcer became an unbypassable quality gate

Patterns That Emerged

Through 12 migrations, clear patterns emerged that accelerated each subsequent port:

The Standard Structure: Every agent followed the same YAML frontmatter pattern, making configuration predictable and debuggable.

The Testing Triangle: Basic functionality → Integration testing → Edge case validation became our standard testing flow.

The Documentation Pyramid: Purpose → Capabilities → Integration → Troubleshooting → Examples became our documentation template.

The Enhancement Cycle: Initial port → Identify limitations → Enhance with MCP → Validate improvements → Document changes.

The Quality Checkpoint: Every significant change triggered qa-enforcer, making quality enforcement automatic rather than optional.

Conclusion

Migrating 12 AI agents wasn't just a technical achievement; it was a masterclass in systematic methodology, architectural understanding, and quality-first development. The success came not from rushing through migrations but from establishing patterns, documenting thoroughly, and treating each agent as an opportunity for enhancement.

The real victory isn't the successful migration but the framework created along the way. Whether migrating AI agents, refactoring legacy systems, or building new platforms, the principles remain:

- **Start simple, build confidence**
- **Document everything, immediately**
- **Test with real scenarios**
- **Enforce quality ruthlessly**
- **Enhance when possible, preserve when necessary**

Your migration challenge awaits. With systematic approaches, comprehensive documentation, and unwavering quality standards, even the most complex migrations become manageable journeys of discovery and improvement.

You can see files for this article here.

. . .

The complete migration framework, patterns, and enhanced agents demonstrate what's possible when systematic methodology meets architectural understanding. Each agent stands as both a functional tool and a lesson in cross-platform transformation.

About the Author

I am Rick Hightower, a seasoned professional with experience as an executive and data engineer at a Fortune 100 financial technology organization. My work there involved developing advanced Machine Learning and AI solutions designed to enhance customer experience metrics. I maintain a balanced interest in both theoretical AI concepts and their practical applications in enterprise environments.

My professional credentials include TensorFlow certification and completion of Stanford's Machine Learning Specialization program, both of which have significantly contributed to my expertise in this field. I value the integration of academic knowledge with practical implementation. My professional experience encompasses work with supervised learning methodologies, neural network architectures, and various AI technologies, which I have applied to develop enterprise-grade solutions that deliver measurable business value.

Connect with Richard on LinkedIn or Medium for additional insights on enterprise AI implementation.

Open Code | Agentic Workflow | Agentic Ai | Claude Code Agent

No responses yet

Bgerby

What are your thoughts?

More from Rick Hightower

Rick Hightower

Claude Code Sub-Agents: Build a Documentation Pipeline in Minutes, Not Weeks

Claude Code Sub-Agents: Build a Documentation Pipeline in Minutes, Not Days, or Weeks

Sep 8 18 2

Rick Hightower

Building AI-Powered Search and RAG with PostgreSQL and Vector Embeddings

Unlock the future of search! Discover how PostgreSQL's vector embeddings are empower traditional databases into powerful AI-driven search!

May 5 28

Rick Hightower

Claude Agent SDK vs. OpenAI AgentKit: A Developer's Guide to Building AI Agents

A comprehensive comparison of two leading frameworks for building AI agents, released within days of each other in fall 2025

Oct 12 26 1

Rick Hightower

OpenCode Agents: Another Path to Self-Healing Documentation Pipelines

Remember my morning hike where I dictated Claude Code agents into existence? (Article 1: the origin story; Article 2: the deep dive). That...

Sep 17 16 1

See all from Rick Hightower

Recommended from Medium

Daniel Avila

I read Anthropic's complete Tool Creation Guide so you don't have to. Here's What Actually Works!

Anthropic released the article "Writing effective tools for agents—with agents" with the goal of teaching the community clear and direct...

Sep 12 14

NebulaGraph Database

When AI Meets Graph Databases: Innovating with Multimodal Data Fusion (Part II)

We'll examine how the powerful intelligent data foundation unlocks intelligent applications across the enterprise, transforming the very...

Sep 7

In Dare To Be Better by Max Petrusenko

Claude Skills: The $3 Automation Secret That's Making Enterprise Teams Look Like Wizards

How a simple folder is replacing $50K consultants and saving companies literal days of work

Oct 17 283 4

Riccardo Tartaglia

5 Essential MCP Servers Every Developer Should Know

I've been experimenting with Model Context Protocol servers for a few months now, and I have to say, they've changed the way I work.

Oct 11 26 3

In Data Science Collective by Manish Shivanandhan

The Future of AI Coding: Why OpenCode Is Changing How Developers Build

OpenCode brings the power of large language models right into your terminal, making coding faster, smarter, and more human than ever...

Oct 6 84

Barnacle Goose

How GPT-5-Codex Compares to Claude Sonnet 4.5

The fall of 2025 was marked by the arrival of two contenders from the industry's leading AI labs. On September 15, OpenAI launched...

Oct 17 3

See more recommendations