The Hitchhiker's Guide to AI Coding Agents

(Don't Panic!)

Four Ways to Make AI Agents Collaborate on Your Codebase

Ken Kousen • AI Codecon • September 2025

The Problem

Each AI agent has unique strengths:

- **Gemini:** 2M+ token context window
- **Claude:** Superior reasoning & code generation
- **GPT-5:** Speed modes & broad capabilities

The Challenge: How do we combine their strengths?

Four Collaboration Patterns

1. **Programmatic Orchestration**
   LangChain4j passing chat memory between agents

2. **Manual Multi-Terminal**
   Three console windows, copy-paste coordination

3. **MCP Server Wrapping**
   Headless agents exposed as MCP servers

4. **Claude Code Agent Orchestration**
   Sub-agents managed by Claude Code

Method 1: Programmatic (LangChain4j)

Real Example: OperaGenerator (github.com/kousen/OperaGenerator)

```java
@Service
public class OperaGenerator {
    @Autowired ChatLanguageModel claude;
    @Autowired ChatLanguageModel chatGPT;
    @Autowired ChatLanguageModel gemini;

    public Opera createOpera(String theme) {
        ChatMemory sharedMemory = new ChatMemory();

        // ChatGPT & Claude trade writing scenes
        String scene1 = chatGPT.chat(sharedMemory,
            "Write opening scene about " + theme);
        String scene2 = claude.chat(sharedMemory,
            "Continue the opera with next scene...");

        // Gemini acts as music critic
        String critique = gemini.chat(sharedMemory,
            "As a music critic, review this opera...");

        return new Opera(scenes, critique);
    }
}
```

Method 2: Manual Multi-Terminal

Real Example: Cosmic Catalog (github.com/kousen/cosmic-catalog)

Terminal 1: Gemini

```
$ gemini "review the code"
> Created tags v6–v7
> Added integration tests
> Version conflict tests
```

Terminal 2: Claude

```
$ claude code "refactor"
> Created tags v8, v11–v12
> Service extraction
> Caching, exceptions
```

Terminal 3: Codex

```
$ codex "add docs"
> Created tags v9–v10
> API documentation
> Docker, CI pipeline
```

Result: v5 → v15 with full production features

But: Manual copy-paste between terminals

Method 3: MCP Server Wrapping

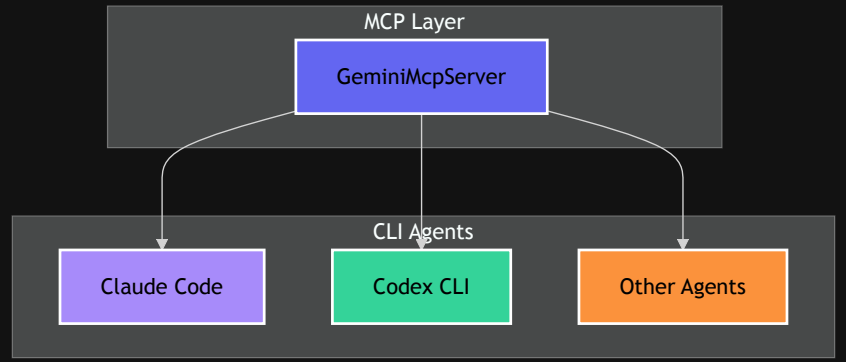
Real Example: GeminiMcpServer (github.com/kousen/GeminiMcpServer)

Setup & Usage

```
# Build and connect
$ git clone .../GeminiMcpServer
$ ./gradlew build
$ claude mcp add gemini-analyzer \
  -- java -jar build/libs/*.jar
```

Benefits: ✅ 2M+ tokens ✅ Reusable ✅ Token efficient ✅ MCP standard

Architecture



One wrapper, infinite reuse across all your AI agents

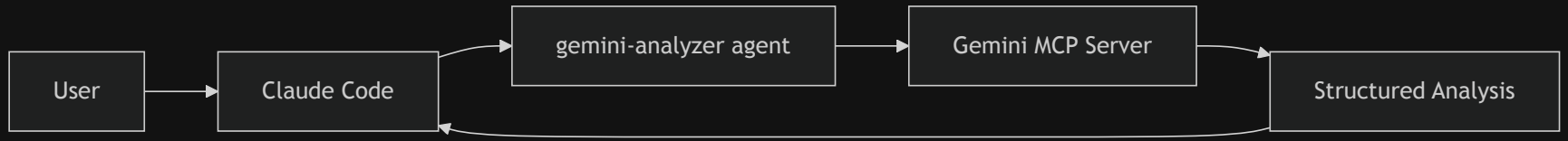
Method 4: Claude Code Agent Orchestration

Real Example from Today:

```
User: "Ask gemini to analyze the codebase"
```

What Happened:

1. Claude Code invoked `gemini-analyzer` agent
2. Gemini analyzed 68 files (~78k tokens)
3. Returned comprehensive architecture analysis
4. Saved ~75k tokens in Claude's context

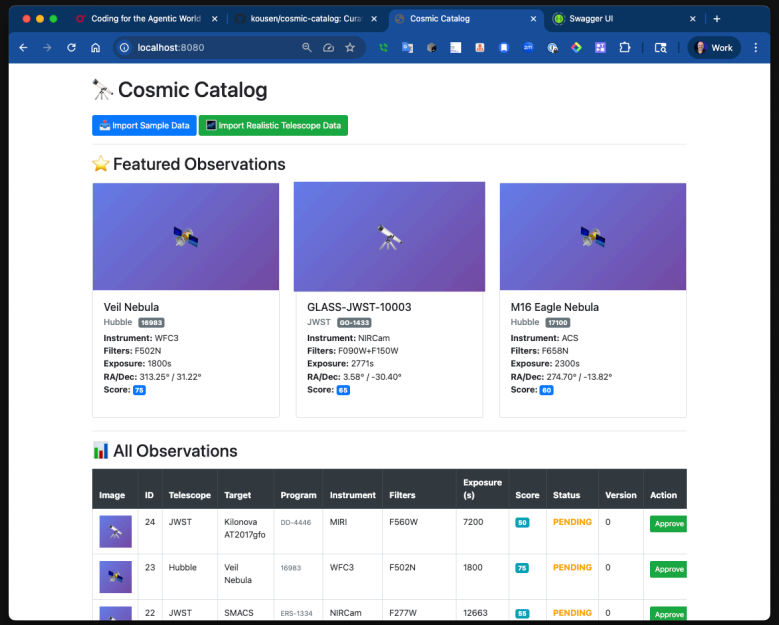


✅ Token efficient | ✅ Intelligent routing | ✅ No context switching

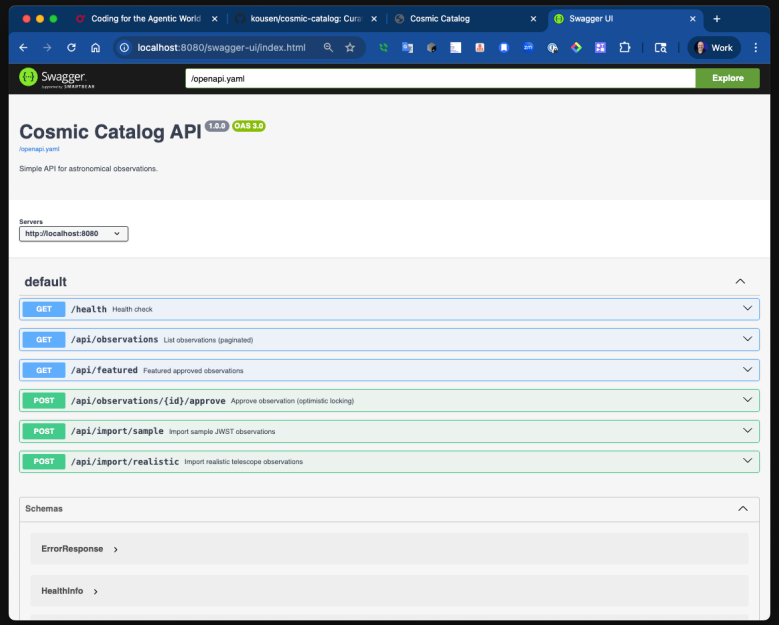
Comparison Matrix

Method	Setup Complexity	Token Efficiency	Automation	Best For
Programmatic	High	Excellent	Full	Production systems
Multi-Terminal	None	Poor	Manual	Quick experiments
MCP Wrapping	Medium	Good	Semi	Tool sharing
Agent Orchestration	Low	Excellent	Full	Complex workflows

The Cosmic Catalog Demo



Featured observations with scoring system



Professional API documentation (OAS 3.0)

The Cosmic Catalog Demo (Details)

What We Built

Spring Boot app managing telescope observations:

- 🔭 **Featured Observations**: Veil Nebula, M16 Eagle, GLASS-JWST
- 📊 **Smart Scoring**: Exposure time, filters, instruments
- 🚀 **Real Data**: Hubble (16983, 17100) & JWST (DD-4446, GO-1433)
- ✅ **Approval Workflow**: Version control, optimistic locking

How Agents Contributed

Gemini (v6-7):
• Integration tests
• Version conflicts

Claude (v8,v11-12):
• Service extraction
• Caching, exceptions

Codex (v9-10):
• Swagger API
• Docker, CI/CD

All (v13-15):
• E2E tests
• Production ready

API Documentation

Swagger UI (OAS 3.0):

```
GET  /health
GET  /api/observations
GET  /api/featured
POST /api/observations/{id}/approve
POST /api/import/sample
POST /api/import/realistic
```

Production Features:

- Optimistic locking
- Error responses
- Pagination
- Caching

Time: Hours not weeks!

Key Insights

What We Learned:

1. **Token efficiency matters**
   MCP servers and agents save 90%+ context

2. **Each method has its place**
   Quick tasks → Multi-terminal | Production → Programmatic

3. **Orchestration is the future**
   Intelligent routing beats manual coordination

4. **Standards enable innovation**
   MCP protocol allows tool interoperability

Practical Recommendations

Start Simple, Scale Smart:

For Exploration:

1. Try multi-terminal first
2. Identify repetitive patterns
3. Wrap frequent tasks in MCP

For Production:

1. Start with agent orchestration
2. Add programmatic for CI/CD
3. Use MCP for tool sharing

Remember: The best method is the one that ships code

🚀 Get Everything

Three Repositories, Four Methods

Cosmic Catalog

github.com/kousen/cosmic-catalog

- 📦 Spring Boot app
- 🏷️ Git tags v5-v15
- 📄 Multi-terminal method

OperaGenerator

github.com/kousen/OperaGenerator

- 🎭 LangChain4j demo
- 🤖 Programmatic method
- 📝 AI collaboration

GeminiMcpServer

github.com/kousen/GeminiMcpServer

- 🔌 MCP wrapper
- 💎 2M+ token context
- 🚀 Agent orchestration

```
git clone https://github.com/kousen/[cosmic-catalog|OperaGenerator|GeminiMcpServer]
```

Thank You!

Ken Kousen

President, Kousen IT, Inc.

ken.kousen@kousenit.com
github.com/kousen
@talesfromthejarside
kousenit.substack.com
linkedin.com/in/kenkousen
bsky.app/profile/kousenit.com

AI agents aren't here to replace developers—
they're here to make us **better** developers.