# Multi-Agent AI Demonstrations: Hands-On Workshop Plan

This document provides three progressive demonstrations of multi-agent AI systems, designed to accompany the "Multi-Agent AI: Architectures for Collaborative Intelligence" presentation. Each demonstration increases in complexity while remaining achievable within a workshop setting.

## Overview

| Demo | Architecture | Tools | Time Estimate | Difficulty |
|---|---|---|---|---|
| 1: Research & Write | Sequential Pipeline | Claude Code + Subagents | 15-20 min | Beginner |
| 2: Code Review Pipeline | Hierarchical + Debate | Claude Agent SDK | 25-35 min | Intermediate |
| 3: Self-Organizing | Collaborative Network | Claude Flow | 20-30 min | Intermediate |

| Demo | Architecture | Tools | Time Estimate | Difficulty |
|---|---|---|---|---|
| Swarm | | | | |

### Prerequisites (All Demos)

Before starting any demonstration, ensure you have:

- **Node.js 18+** installed ( `node --version` )
- **Claude Code** installed globally ( `npm install -g @anthropic-ai/claude-code` )
- **Anthropic API key** set in your environment ( `export ANTHROPIC_API_KEY=sk-ant-...` )
- A **Claude Pro, Max, or API subscription** for optimal performance

## Demo 1: Research & Write Pipeline

**Architecture:** Sequential Pipeline (Relay)
**Time:** 15-20 minutes
**What You'll Build:** A two-agent system where a Researcher agent gathers information and a Writer agent synthesizes it into a document.

### Learning Objectives

- Understand how subagents operate with isolated context windows
- See how agents hand off work in a sequential pipeline
- Observe the efficiency gains from task specialization

### Setup

1. Create a new project directory:

```
mkdir demo-research-write
cd demo-research-write
```

2. Initialize Claude Code and create the subagent definitions:

```
mkdir -p .claude/agents
```

3. Create the Researcher subagent ( `.claude/agents/researcher.md` ):

```
---
name: researcher
description: Use this agent to research topics, gather information, and
tools: Read, Grep, Glob, WebFetch, WebSearch
---

You are a Research Specialist. Your role is to:

1. Search for relevant information on the given topic
2. Identify key facts, statistics, and expert opinions
3. Organize findings into a structured research brief
4. Note sources and their credibility

## Output Format
Always structure your findings as:
- **Key Facts**: Bullet points of essential information
- **Context**: Background needed to understand the topic
- **Sources**: Where information was found
- **Gaps**: What information is still missing

Be thorough but focused. Quality over quantity.
```

4. Create the Writer subagent ( `.claude/agents/writer.md` ):

```
---
name: writer
description: Use this agent to transform research and notes into polishe
tools: Read, Write, Edit
---

You are a Writing Specialist. Your role is to:

1. Review provided research and notes
2. Structure content for the target audience
3. Write clear, engaging prose
4. Ensure logical flow and coherence

## Writing Principles
- Lead with the most important information
- Use concrete examples to illustrate abstract concepts
- Vary sentence length for rhythm
- End sections with transitions or key takeaways
```

```
## Output
Produce publication-ready content. Include a brief note about any areas
```

## Running the Demo

Launch Claude Code and use this prompt to trigger the pipeline:

```
I need a 500-word briefing on "the current state of quantum computing
for business applications."

Use the researcher subagent to gather current information about
quantum computing business use cases, then use the writer subagent to
produce a polished executive briefing.

Work in sequence: complete research first, save findings to research-
notes.md, then have the writer transform those notes into the final
briefing.md document.
```

## What to Watch For

- **Context isolation**: Notice how each subagent works in its own context window
- **Handoff mechanism**: The research notes file acts as the interface between agents
- **Specialization**: Each agent focuses on its specific task without distraction
- **Output in Task()**: Look for `Task(Performing research...)` in the terminal output

## Variation (3-Agent Pipeline)

Add a third agent for editing/fact-checking:

Create `.claude/agents/editor.md`:

```
---
name: editor
description: Use this agent for final review, fact-checking, and polish
tools: Read, Write, Edit, WebSearch
---

You are an Editorial Specialist. Review content for:
- Factual accuracy (verify key claims)
- Clarity and readability
- Grammar and style consistency
- Logical flow and structure
```

```
Provide specific edits, not general suggestions.
```

Then modify your prompt to include the editing pass.

## Demo 2: Code Review Pipeline with Anthropic Tools

**Architecture:** Hierarchical (Manager-Worker) + Debate elements
**Time:** 25-35 minutes
**What You'll Build:** A code review system with a coordinator managing specialized reviewers who can flag disagreements.

### Learning Objectives

- Build a multi-agent system using the Claude Agent SDK
- Implement a hierarchical pattern with a coordinator agent
- Add debate elements through parallel review with disagreement surfacing
- See how agents can operate autonomously with tool access

### Setup

1. Create project and initialize:

```
mkdir demo-code-review
cd demo-code-review
npm init -y
npm install @anthropic-ai/claude-agent-sdk
```

2. Create the main orchestrator ( `review-system.js` ):

```
import { query, ClaudeAgentOptions } from '@anthropic-ai/claude-agent-s

// Configuration for different reviewer personas
const REVIEWERS = {
  security: {
    name: "Security Reviewer",
    prompt: `You are a security-focused code reviewer. Analyze code for
- Input validation vulnerabilities
- Authentication/authorization issues
- Data exposure risks
```

```
- Injection vulnerabilities
- Secrets in code

Rate severity: CRITICAL, HIGH, MEDIUM, LOW
Be specific about line numbers and remediation.`
  },

  performance: {
    name: "Performance Reviewer",
    prompt: `You are a performance-focused code reviewer. Analyze code
- Algorithmic complexity issues
- Memory leaks or excessive allocation
- Unnecessary database queries
- Missing caching opportunities
- Blocking operations that should be async

Estimate impact: HIGH, MEDIUM, LOW
Suggest specific optimizations.`
  },

  maintainability: {
    name: "Maintainability Reviewer",
    prompt: `You are a maintainability-focused code reviewer. Analyze c
- Code clarity and readability
- Appropriate naming conventions
- Function/class size and complexity
- Test coverage gaps
- Documentation needs

Focus on long-term code health.`
  }
};

async function runReview(codeFile) {
  console.log(`\n🔍 Starting multi-agent code review of: ${codeFile}\n`

  const reviews = {};

  // Run all reviewers in parallel
  const reviewPromises = Object.entries(REVIEWERS).map(async ([type, co
    console.log(`  ▶ ${config.name} starting...`);

    const result = await query({
      prompt: `Review the following code file: ${codeFile}

Read the file and provide your specialized review.`,
      systemPrompt: config.prompt,
      options: {
        maxTurns: 5,
```

```
        tools: ['Read', 'Grep', 'Glob']
      }
    });

    console.log(`  ✓ ${config.name} complete`);
    return { type, review: result };
  });

  const results = await Promise.all(reviewPromises);
  results.forEach(r => reviews[r.type] = r.review);

  // Coordinator synthesizes reviews
  console.log(`\n📋 Coordinator synthesizing reviews...\n`);

  const synthesis = await query({
    prompt: `You are a Code Review Coordinator. Synthesize these three s

SECURITY REVIEW:
${reviews.security}

PERFORMANCE REVIEW:
${reviews.performance}

MAINTAINABILITY REVIEW:
${reviews.maintainability}

Create a unified report that:
1. Highlights consensus issues (mentioned by multiple reviewers)
2. Flags any contradictions between reviewers
3. Prioritizes the top 5 issues to address
4. Provides an overall code health score (A-F)

Be concise but actionable.`,
    options: { maxTurns: 1 }
  });

  console.log("═══════════════════════════════════════════");
  console.log("        FINAL CODE REVIEW REPORT           ");
  console.log("═══════════════════════════════════════════\n");
  console.log(synthesis);

  return synthesis;
}

// Run on a file passed as argument
const targetFile = process.argv[2] || 'sample.js';
runReview(targetFile).catch(console.error);
```

3. Create a sample file to review ( `sample.js` ):

```
// User authentication handler
const mysql = require('mysql');

function authenticateUser(username, password) {
  const query = "SELECT * FROM users WHERE username='" + username + "' A

  const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: 'admin123',
    database: 'myapp'
  });

  let result = null;
  connection.query(query, function(err, rows) {
    if (rows.length > 0) {
      result = rows[0];
    }
  });

  return result;
}

function getAllUsers() {
  const users = [];
  for (let i = 0; i < 10000; i++) {
    const user = database.getUserById(i);
    if (user) users.push(user);
  }
  return users;
}

module.exports = { authenticateUser, getAllUsers };
```

## Running the Demo

```
node review-system.js sample.js
```

## What to Watch For

- **Parallel execution**: All three reviewers run simultaneously
- **Specialization**: Each finds different issues (SQL injection vs N+1 query vs naming)

- **Synthesis**: The coordinator identifies consensus and contradictions
- **Debate element**: When reviewers disagree, the coordinator must resolve

### Interactive Variation

Add this to `review-system.js` to create an interactive debate when reviewers disagree:

```
async function resolveDisagreement(issue, reviews) {
  // Have reviewers defend their positions
  const debate = await query({
    prompt: `Two reviewers disagree about: ${issue}

Reviewer A says: ${reviews.a}
Reviewer B says: ${reviews.b}

Act as a neutral arbiter. Ask each reviewer to defend their position wi
    options: { maxTurns: 3 }
  });

  return debate;
}
```

## Demo 3: Self-Organizing Swarm with Claude Flow

**Architecture:** Collaborative Network (Society) **Time:** 10 minutes (pre-recorded walkthrough) **What You'll Show:** A pre-recorded swarm building a todo app, with walkthrough of the output and memory trail.

**Important:** This demo is pre-recorded due to alpha software unpredictability and student attention in hour 3. A live demo option is available if time permits.

### Learning Objectives

- Understand how emergent coordination differs from structured orchestration
- See how agents discover work through shared memory
- Observe the output of a self-organizing swarm
- Understand when swarm patterns are appropriate

# Pre-Recording Instructions (Before the Live Session)

## Recording Setup

### Environment:

```
# Install claude-flow
npm install -g claude-flow@alpha

# Create fresh project directory
mkdir demo-swarm-recording
cd demo-swarm-recording

# Initialize
npx claude-flow@alpha init --force
```

### Recording Tools:

- Terminal recording: Use `asciinema` or screen recording software
- Capture both terminal output AND resulting files
- Record multiple takes—pick the cleanest run

## Recording the Demo

### Step 1: Initialize and spawn agents

```
# Start fresh
npx claude-flow@alpha swarm init --topology mesh --max-agents 6

# Spawn specialized agents (record each spawn)
npx claude-flow@alpha swarm spawn architect "System Designer"
npx claude-flow@alpha swarm spawn coder "Frontend Developer"
npx claude-flow@alpha swarm spawn coder "Backend Developer"
npx claude-flow@alpha swarm spawn tester "QA Engineer"
npx claude-flow@alpha swarm spawn documenter "Documentation Writer"
```

### Step 2: Monitor in second terminal

```
# In separate terminal, capture status output
npx claude-flow@alpha swarm status --watch
```

### Step 3: Assign the task

```
npx claude-flow@alpha swarm task "Build a todo list web application witl
```

### Step 4: Capture memory trail

```
# After completion, capture the memory log
npx claude-flow@alpha memory list > memory-trail.log

# Get stats
npx claude-flow@alpha memory stats > memory-stats.log
```

### Step 5: Save all output files

Copy the generated files to a backup directory:

- `index.html`
- `styles.css`
- `app.js`
- `tests/` directory
- `README.md`
- `memory-trail.log`
- `memory-stats.log`

## What to Capture

### Required Artifacts:

| Artifact | Purpose |
|---|---|
| Terminal recording or screenshots | Show agent spawning and coordination |
| `index.html` , `styles.css` , `app.js` | The actual output to show students |
| `memory-trail.log` | Evidence of coordination via memory |
| `memory-stats.log` | Summary statistics |

### Ideal Memory Trail Example:

```
[12:01:15] architect → STORED: "architecture-decision"
           Value: "Single-page app, vanilla JS, localStorage for
persistence"
```

```
[12:01:47] frontend → QUERIED: "architecture"
           Found: architect's decision

[12:02:03] frontend → STORED: "component-started"
           Value: "Building header and input form"

[12:02:15] backend → QUERIED: "architecture"
           Found: architect's decision

[12:02:31] backend → STORED: "component-started"
           Value: "Building todo CRUD operations"

[12:03:45] qa → QUERIED: "components"
           Found: frontend and backend work

[12:04:02] qa → STORED: "tests-written"
           Value: "Unit tests for add/delete/toggle"
```

### Recording Tips

1. **Run multiple times** — Pick the cleanest, most illustrative run
2. **Watch for good examples** — Look for clear memory coordination patterns
3. **Note timestamps** — Helps explain the sequence during walkthrough
4. **Test the output** — Make sure the generated todo app actually works
5. **Have backup** — If no clean run, create synthetic memory trail that illustrates concepts

## Live Session: Demo Walkthrough

### Presenting the Pre-Recorded Demo (10 min)

**Introduction (1 min)** "Unlike our previous demos, this one was pre-recorded. Swarm behavior is inherently unpredictable—that's its nature. Rather than wait for live execution, let's walk through what a successful run produced and understand how it happened."

**Show the Task (1 min)** "We asked a swarm to build a todo list web application. No predefined workflow. Just spawn agents and let them figure it out."

Show the agent configuration:

```
Agents Spawned:
- Architect (System Designer)
- Frontend Developer
- Backend Developer
- QA Engineer
- Documentation Writer
```

**Show the Output (3 min)** Walk through each file:

- `index.html` — Point out structure decisions
- `styles.css` — Note styling choices
- `app.js` — Highlight functionality
- `README.md` — Show how documenter synthesized everything

"Open the app in a browser if possible to show it works."

**Show the Memory Trail (4 min)** This is the key teaching moment. Walk through the memory log:

1. "Architect wrote the architecture decision first"
2. "Frontend queried memory, found the architecture, decided to build UI"
3. "Backend did the same independently—parallel work"
4. "QA discovered new code by querying, wrote tests"
5. "Nobody assigned these tasks. Each agent discovered useful work."

**The Key Insight (1 min)** "In Section 2, our coordinator explicitly assigned Security, Performance, and Maintainability reviewers. Here, nobody told Frontend to build the header. It queried memory, saw what was needed, and acted. That's the fundamental difference—discovery versus assignment."

## Contrast Table (Show on Screen)

| Section 2 (Code Review) | Section 3 (Swarm) |
|---|---|
| Coordinator assigned reviewers | No assignment—agents claimed work |
| Predefined roles | Roles discovered organically |
| Coordinator synthesized | Synthesis emerged |
| Predictable, auditable | Adaptive, exploratory |

## Optional: Live Demo (If Time Permits)

If there's extra time and student interest, offer a brief live demonstration:

```
# Quick swarm demo (simpler task for reliability)
npx claude-flow@alpha init --force
npx claude-flow@alpha swarm "Summarize the key files in this directory"
```

**Set Expectations:** "This is live, so results may vary. That's the nature of emergent systems. If it works, great. If not, that's also a lesson about unpredictability."

## Troubleshooting (For Recording)

### Common Issues During Recording

**claude-flow MCP connection fails:**

```
claude mcp remove claude-flow
claude mcp add claude-flow npx claude-flow@alpha mcp start
```

**Swarm won't start:**

```
npx claude-flow@alpha memory init
npx claude-flow@alpha swarm init --force
```

**Agents stuck:**

```
npx claude-flow@alpha swarm stop
npx claude-flow@alpha swarm init --force
```

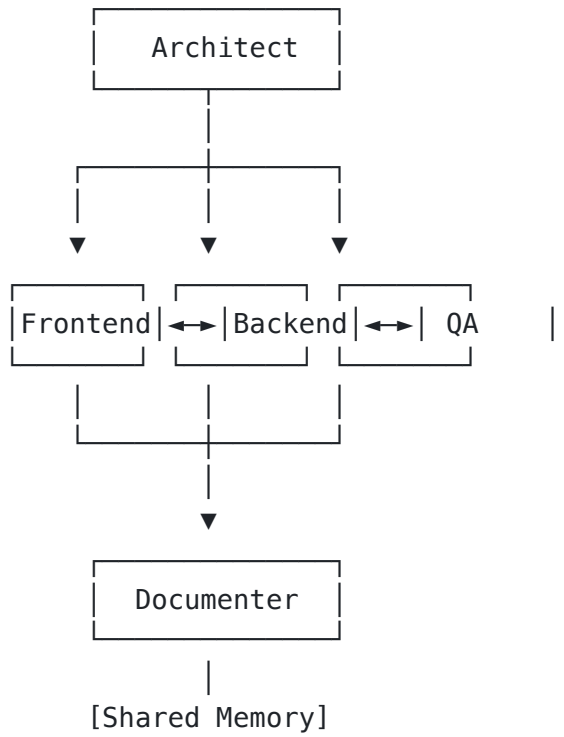
**Memory not persisting:**

```
npx claude-flow@alpha memory init
```

### Fallback: Create Illustrative Output

If you cannot get a clean recording, create representative files manually that illustrate what a swarm would produce. The teaching value is in the memory trail pattern, not the specific code generated.

## Visualization: Swarm Topology




All agents share the same memory store, enabling organic coordination without explicit message passing.

## Troubleshooting

### Common Issues

**Claude Code not finding subagents:**

- Ensure `.claude/agents/` directory exists
- File extension must be `.md`
- Restart Claude Code after adding agents

**Claude Flow MCP connection fails:**

```
# Remove and re-add the MCP server
claude mcp remove claude-flow
claude mcp add claude-flow npx claude-flow@alpha mcp start
```

**API rate limits:**

- Reduce parallel agent count
- Add delays between agent spawns
- Use `--max-agents 3` for smaller swarms

**Memory not persisting:**

```
# Initialize memory system
npx claude-flow@alpha memory init
```

## Discussion Points for Each Demo

### After Demo 1 (Pipeline)

- How does context isolation help and hurt?
- What happens if the researcher misses key information?
- How would you add error correction to the pipeline?

### After Demo 2 (Code Review)

- Why run reviewers in parallel instead of sequence?
- How does the debate pattern improve output quality?
- What happens when reviewers fundamentally disagree?

### After Demo 3 (Swarm)

- How did agents coordinate without explicit instructions?
- What role does shared memory play?
- What are the risks of emergent behavior?

## Next Steps

After completing these demos, explore:

1. **Hybrid architectures**: Combine patterns (hierarchical swarm with debate)
2. **Custom agents**: Design agents for your specific domain
3. **Production deployment**: Add monitoring, cost controls, and safety rails
4. **MCP integrations**: Connect agents to your data sources (GitHub, Slack, databases)

### Resources

- Claude Code Subagents Documentation
- Claude Agent SDK GitHub
- Claude Flow Wiki
- Agentics Foundation
- Model Context Protocol