

한 걸음 앞선 개발자가 지금 꼭 알아야 할 클로드 코드

한 걸음 앞선 개발자가

지금 꼭 알아야 할

MASTERING
CLAUDE
CODE

조훈, 정찬호 지음

클로드

코드

CLAUDE
CODE

실무에서 검증된 개발 방식 그대로,

매일 1시간 4주

Claude Code 에이전트 실전 훈련!

설치(macOS, Windows)
CLAUDE.md 설정
MCP 연동/다양한 활용 전략
클로드 워크플로 전략
설계 → 부트스트래핑 →
테스트 → 개선 → 명세
클로드 코드의 효율을 극대화
하는 다양한 방법

갈빗

깃허브: https://github.com/sysnet4admin/Book_Claude-Code

에이전트를 기반으로 여러 업무를 처리하다 보면 자연스럽게 지시 후 대기하는 시간을 반복적으로 갖게 됩니다. 이렇게 생긴 공백은 자연스럽게 여러 에이전트를 활용해 병렬로 여러 업무들을 처리하고 싶은 욕구를 만듭니다. 이번 문서에서는 멀티 에이전트를 다루는 방법의 한면으로 **ClaudeSquad**와 **ClaudeSwarm**을 소개하고 비교하겠습니다.

Claude Squad

하나의 **Git** 리포에서 클로드코드로 작업을하다보면 자연스럽게 다른 성격의 작업을 동시에 진행하고 싶을 수 있습니다. 예를들어 동일한 리포에서 **A feature**와 **B feature**를 별도의 브랜치로 작업하고 싶을 수 있습니다. 이런 형태의 작업에서 가장 자연스러운 방법은 **Git worktree**를 사용하는 방법이며 실제 **Anthropic**의 베스트 프랙티스에서도 **Git worktree**를 사용하는걸 추천합니다. 하지만 이 방법엔 다소 불편함이 존재합니다.

- 한 리포에서 파생된 여러 에이전트 태스크를 한눈에 보기가 힘들다.
- 여러 태스크 에이전트가 진행되는 사이를 오갈때 번거로움이 있다.
- **Git worktree**에 대한 최소한의 이해가 필요하다.
- 한눈에 비교를 보기 힘들다.

이럴때 도움을 줄수 있는게 **Claude Squad** 입니다.

<https://github.com/smtg-ai/claude-squad>

이름은 **Claude Squad** 이지만 클로드 외에 **Codex**나 **Gemini**, **Aider**와 같은 에이전트 도구와도 사용할 수 있습니다.

<노트>

안타깝게도 이 포스팅을 작성하는 시점에서 **Claude Squad**는 다소 동작에 많은 버그를 내포하고 있는 상태입니다. 현재 활발하게 개발되고 있는 이런 에이전트 관련 도구들은 아직 성숙도가 충분하지 않은 상태이니 좀 더 인내심을 갖고 상태를 트래킹해가며 써야할것으로 보입니다. 좀 더 신뢰성있게 이를 유사한 워크플로우를 사용하려면 직접 **tmux**와 깃 워크트리를 제어하시는 방향을 추천드립니다.

</노트>

간단한 예제를 통해 **Claude Squad**를 이해 해보겠습니다.

먼저 내부적으로 **Claude Squad**는 **tmux**와 **gh**를 사용합니다. **MacOS**기준으로 모두 아래와 같이 **brew**로 간단하게 설치할 수 있습니다.

<Terminal박스>

```
$ brew install tmux gh claude-squad
```

```
$ ln -s "$(brew --prefix)/bin/claude-squad" "$(brew --prefix)/bin/cs"
```

```
$ cs --help
```

Claude Squad - Manage multiple AI agents like Claude Code, Aider, Codex, and Amp.

Usage:

```
claude-squad [flags]
```

```
claude-squad [command]
```

Available Commands:

completion Generate the autocompletion script for the specified shell

debug Print debug information like config paths

help Help about any command

reset Reset all stored instances

version Print the version number of claude-squad

Flags:

-y, --autoyes [experimental] If enabled, all instances will automatically accept prompts

-h, --help help for claude-squad

-p, --program string Program to run in new instances (e.g. 'aider --model ollama_chat/gemma3:1b')

Use "claude-squad [command] --help" for more information about a command.

</Terminal박스>

Claude Squad는 기본적으로 Git worktree를 Claude Code와 함께 잘 다루기 위한 도구입니다. 간단하게 nodejs기반의 Hello World 프로그램을 담은 깃 리포를 만들어보겠습니다.

<Terminal박스>

```
$ mkdir helloworld
```

```
$ cd helloworld  
  
$ git init  
  
$ claude -p "Lets write a hello world cli in nodejs" --dangerously-skip-permissions  
  
$ git add -A  
  
$ git commit -m "Initial commit"
```

</Terminal박스>

대략 아래와 같은 상태로 시작합니다.

<Terminal박스>

```
$ tree -L 2 .  
  
.  
├── eslint.config.js  
├── index.js  
└── package.json
```

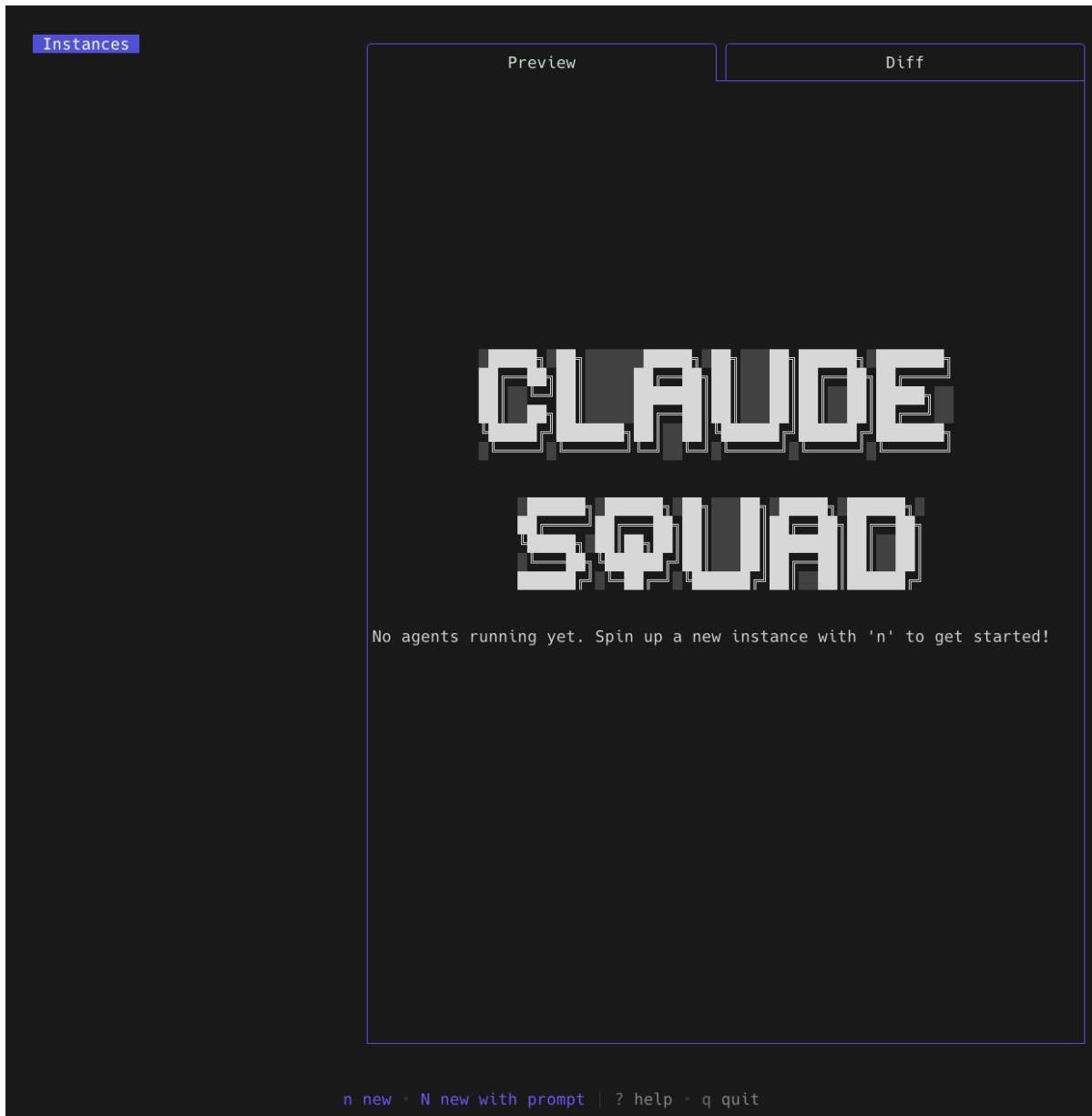
</Terminal박스>

이제 Claude Squad를 써봅니다.

<Terminal박스>

```
$ cs
```

</Terminal박스>



이제 간단한 작업을 두가지 만들어 보도록 합니다.

- hello를 hi로 바꾸기

- hello를 bye로 바꾸기

n 을 누르면 새로운 작업 인스턴스를 만들 수 있습니다. **n**을 누른 후 인스턴스 이름을 입력 해주면 클로드코드가 나타납니다. **Enter**로 인스턴스로 진입 후 다음과 같이 명령해줍니다.

<노트>

기본적으로 **auto-yes** 옵션을 통해 **YOLO**모드로 동작할 수 있도록 배려하고 있으나 현시점에서 제대로 기능하고 있지 않습니다. 승인 프로세스가 반복되는거 피하고 싶은 경우 `~/.claude-squad/config.json` 의 **default_program** 항목을 **wrapping script**로 바꾸고 **claude**

`--dangerously-skip-permissions`가 실행되게끔 하는 방식으로 이를 처리해 볼 수 있습니다.

</노트>

<Claude Code박스>

> Replace hello to hi

</Claude Code박스>

에이전트가 해당 작업을 진행하도록 한 후 **Ctrl-q**를 누르면 다시 인스턴스 리스트 화면이 나옵니다.

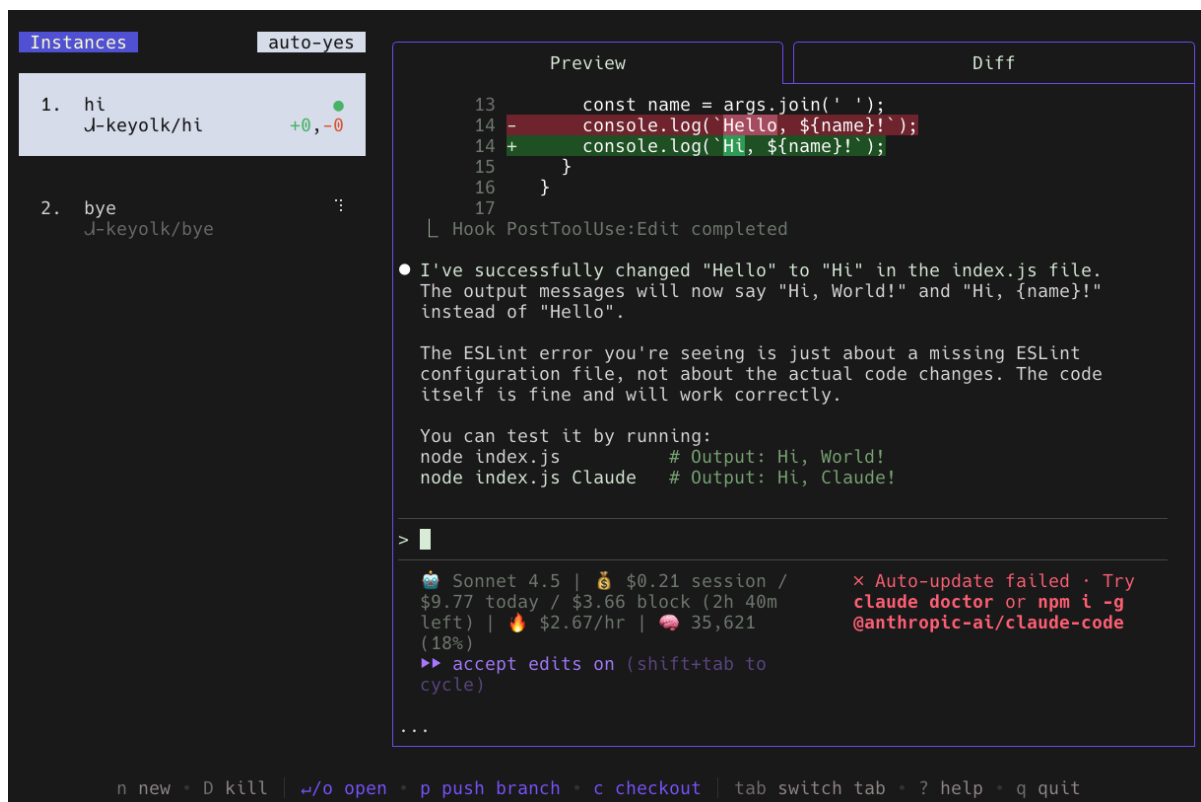
이번에 다시 **n**을 눌러 새로운 인스턴스를 만들고 **Enter**로 진입하여 이번엔 다음과 같은 작업을 진행해봅니다.

<Claude Code박스>

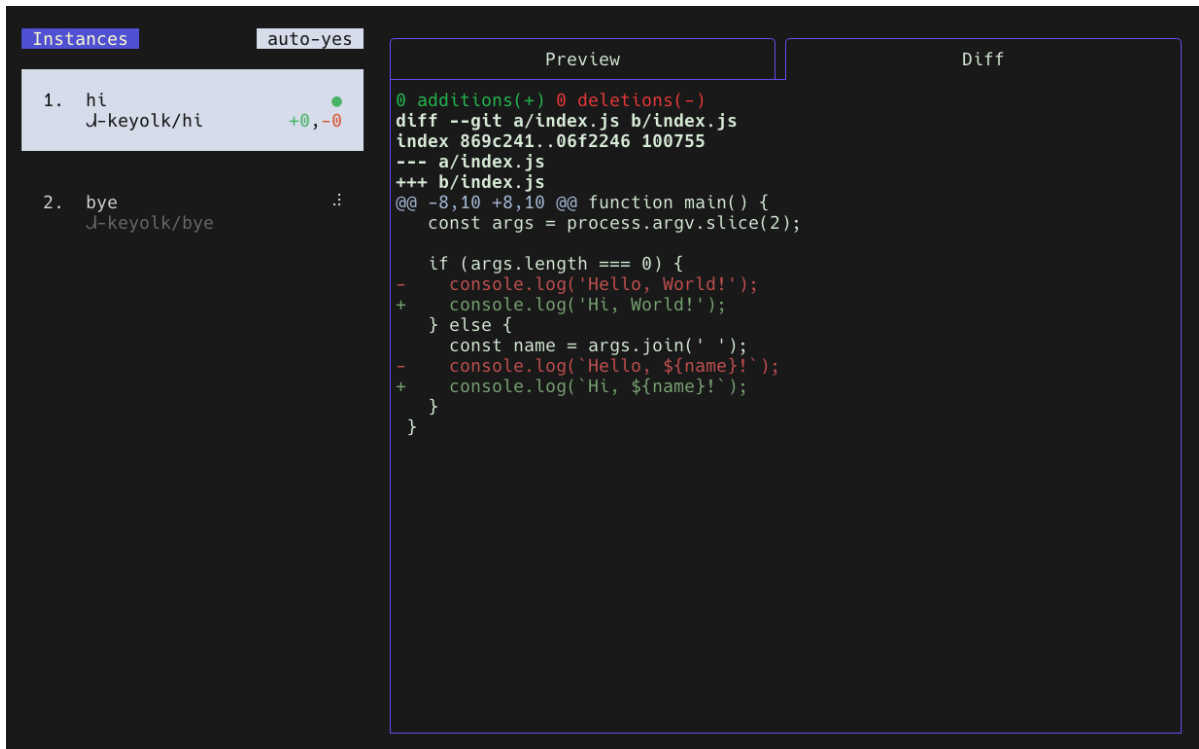
> Replace hello to bye

</Claude Code박스>

이제 화살표 위아래 키로 각 작업의 진행도를 체크할 수 있습니다.



Tab을 누르면 git diff를 확인할 수 있습니다.



각 인스턴스를 선택하고 **c**를 누르면 현재 상태를 커밋 합니다.

각 인스턴스는 별도로 분리된 공간에 해당 리포의 워크트리를 만들어 변경사항을 저장하게됩니다.

<Terminal박스>

```
$ git worktree list
```

```
/Users/keyolk/helloworld
```

```
46c4eaa [master]
```

```
/Users/keyolk/.claude-squad/worktrees/bye_187798c9684a6cf8 80c99bc [keyolk/bye]
```

```
/Users/keyolk/.claude-squad/worktrees/hi_187798b8487ccf30 b5d2241 [keyolk/hi]
```

</Terminal박스>

각 워크트리의 해당하는 브랜치에서 클로드코드 인스턴스가 수행한 작업을 확인할 수 있습니다.

<Terminal박스>

```
$ git log keyolk/hi
```

```
commit b5d2241e440c304226385e1d0b8fb7bbf82a9815 (keyolk/hi)
```

```
Author: Chanhun Jeong <keyolk@gmail.com>
```

```
Date: Fri Nov 14 00:02:10 2025 +0900
```

[claudesquad] update from 'hi' on 14 Nov 25 00:02 KST

commit 46c4eaa0d71bbe5271a529beb0cb0cbeeeafc3202cb (HEAD -> master)

Author: Chanhun Jeong <keyolk@gmail.com>

Date: Thu Nov 13 23:54:48 2025 +0900

Initial commit

</Terminal박스>

비교 후 원하는 체인지를 마스터로 머지 시켜봅니다.

<Terminal박스>

```
$ git merge keyolk/hi
```

```
Updating 46c4eaa..0eac423
```

```
Fast-forward
```

```
index.js | 4 ++--
```

```
1 file changed, 2 insertions(+), 2 deletions(-)
```

</Terminal박스>

필요없어진 워크트리는 다음과 같이 정리할 수 있습니다.

<Terminal박스>

```
$ cs reset
```

```
Storage has been reset successfully
```

```
Tmux sessions have been cleaned up
```

```
Worktrees have been cleaned up
```

```
daemon has been stopped
```

```
wrote logs to /var/folders/75/3vrpqsv57219kg0qncv8xjbh0000gp/T/claudesquad.log
```

```
$ git worktree list
```

```
/Users/keyolk/helloworld 0eac423 [master]
```


</Terminal박스>

여기까지 **Claude Squad**에 대한 대략적인 사용법을 살펴봤습니다. 이런 워크플로우는 다양한 상황에 적용해볼 수 있습니다.

- 독립된 **feature**에 대한 개발
- 하나의 **feature**에 대한 **N**가지 개발 후 최적의 결과물을 가져가기

후자에 대한 표현을 달리 **Bet-of-N**으로 표현하기도 합니다. 이는 하나의 목표에 대해서 에이전트에게 다양한 형태로 지시를 해보고 그 중 가장 좋아보이는걸 고르는 형태의 전략을 뜻합니다. **Claude Squad**를 사용하면 이렇게 하나의 리포를 대상으로 병렬 에이전트 세션을 구동하여 결과물을 만들어가는 워크플로우를 비교적 자연스럽게 진행할 수 있습니다.

Claude Swarm

<노트>

포스팅을 작성하는 시점에서 해당 프로젝트의 이름이 **Swarm**으로 변경되었고, 이전의 **claude-swarm** 보다 많이 확장된 형태로 정리가 되고 있습니다. 기존 설정 역시 호환이되지 않습니다. 이 문서는 책에서 소개되었던 **claude-swarm**을 중심으로 정리되어 있으면 새로 리뉴얼된 **Swarm**에 대해선 해당 리포의 <https://github.com/parruda/swarm/blob/main/docs/v2/README.md> 에서 자세한 내용을 살펴볼 수 있습니다. 안타깝게도 현재 시점에서 **Claude Code**의 플랜 및 **API**로 원활하게 실행되지 않는 부분이 있어 좀 더 성숙도가 올라갈때까지 기다려야하는 상태입니다.

</노트>

앞서 살펴본 **Claude Squad**는 하나의 리포를 다양한 브랜치로 나눠 여러 에이전트로 동시작업을 하기 위한 용도로 사용되었습니다. **Claude Swarm**은 깃 리포와 무관하게 여러 에이전트를 오케스트레이션하여 하나의 목적을 처리하기 위한 루비 기반 프레임워크입니다. **Claude Squad**와 마찬가지로 클로드 코드와 다양한 에이전트를 지원합니다.

설치

설치를 하려면 **Ruby** 환경이 필요합니다. 다음과 같이 **Ruby**를 설치하고 **Ruby**의 패키지 매니저인 **Gem**을 통해 **Swarm**을 설치할 수 있습니다.

<Terminal박스>

```
$ brew install ruby  
  
$ gem install claude_swarm
```

```
$ claude-swarm --help
```

Commands:

claude-swarm clean	...
claude-swarm generate	...
claude-swarm help [COMMAND]	...
claude-swarm init	...
claude-swarm list-sessions	...
claude-swarm mcp-serve --calling-instance=CALLING_INSTANCE -d, --directory=DIRECTORY -m, --model=MODEL -n, --na...	
claude-swarm ps	...
claude-swarm restore SESSION_ID	...
claude-swarm show SESSION_ID	...
claude-swarm start [CONFIG_FILE]	...
claude-swarm version	...
claude-swarm watch SESSION_ID	...

</Terminal박스>

사용 방법

현 시점 리포에서 안내하는 예제를 중심으로 동작을 살펴봅니다.

<Code박스> swarm.yml

```
version: 1
swarm:
  name: "My Dev Team"
  main: lead
  instances:
    lead:
      description: "Team lead coordinating development efforts"
      directory: .
      model: sonnet
      connections: [frontend, backend]
      vibe: true
```

```

frontend:
  description: "Frontend specialist handling UI and user experience"
  directory: ./frontend
  model: sonnet
  allowed_tools:
    - Edit
    - Write
    - Bash
backend:
  description: "Backend developer managing APIs and data layer"
  directory: ./backend
  model: sonnet
  allowed_tools:
    - Edit
    - Write
    - Bash

```

</Code박스>

lead, **frontend**, **backend** 세가지 인스턴스를 구성했습니다. 여기서 인스턴스는 각각 별개의 클로드코드 인스턴스라고 이해하면 됩니다. 이에 각 인스턴스 별로 **description**과 작업공간, 그리고 사용할 모델을 지정해줬습니다. **vibe**는 모든 **Tool**을 제한없이 사용하도록 허용한다는 의미이며 **main** 그리고 **connections**로 인스턴스 사이의 관계와 진입점이 되는 인스턴스를 구성했습니다. 그외 각 인스턴스 별로 사용가능한 **Tool**이나 **MCP**, **Hook**도 지정이 가능합니다.

이제 명시된 작업공간들을 준비해주고 아래와 같이 실행해봅니다.

<Terminal박스>

```

$ mkdir frontend backend

$ claude-swarm start ./swarm.yml

```

</Terminal박스>

이렇게 실행을 하면 우리에게 낮이 익은 클로드코드 프롬프트가 나타납니다. 아래와 같이 간단한 **webapp**을 개발할것을 주문합니다.

<Claude 박스>

```

> Lets build simple example webapp, react/nodejs based






● I'll help you build a simple React/Node.js webapp. Let me first understand the
current directory structure and

then create a plan for this project.

[중략]

🔧 What's Configured

```

-  **CORS enabled for frontend-backend communication**
-  **Vite proxy for API requests**
-  **Hot reload for both frontend and backend**
-  **ESLint configuration for code quality**
-  **Responsive design with modern CSS**

The app is a simple task manager where you can create, complete, and delete tasks. All tasks are stored in

memory (they'll reset when you restart the server), but you can easily add database persistence later!

Would you like me to help you:

1. Install the dependencies and test it?
2. Add database persistence (MongoDB/PostgreSQL)?
3. Add authentication?
4. Deploy it somewhere?

</Claude 박스>

언뜻보면 기존 클로드랑 어떻게 다른건지 구분이 안갈 수 있습니다. 하지만 실제로 우리가 위에서 작성한 인스턴스의 협업을 통해서 결과물이 작성되었습니다.

어떻게 동작하는가?

위에서 우리가 직접 프롬프트를 작성해서 지시한 인스턴스가 설정에서 **main**으로 지정했던 **lead** 인스턴스입니다. 그외 다른 인스턴스는 **connection**을 통해 **lead**와의 관계를 지정했었습니다. 이는 실질적으로는 **frontend**와 **backend** 인스턴스가 **lead**가 사용할 수 있는 **MCP**서버로 동작시킴을 의미합니다. 이 부분을 좀 더 확인해보기 위해 다시 다음과 같이 질의해봅니다.

<Claude 박스>

> Lets explain the outputs

[중략]

🟡 frontend - task (MCP)(prompt: "Explain the React frontend application we just created in the

/Users/gavin.jeong/swarm/frontend directory. Describe:\n1. The main components and their purpose\n2. How the UI works and what features it provides\n3. How it connects to the backend API\n4. The technology stack used (React, Vite, etc.)\n\nKeep the explanation clear and concise for a developer.", description: "Frontend specialist explaining the React app")

└─ Perfect! Now I have a complete picture of the React frontend application. Here's the explanation:

React Frontend Application Overview

... +76 lines (ctrl+o to expand)

● backend - task (MCP)(prompt: "Explain the Node.js Express backend API we just created in the

/Users/gavin.jeong/swarm/backend directory. Describe:\n1. The API endpoints and what they do\n2. The data model and storage approach\n3. How CORS is configured\n4. The overall architecture\n\nKeep the explanation clear and concise for a developer.", description:

"Backend specialist explaining the Express API")

└─ Perfect! Now I can explain the backend API. Here's a comprehensive overview:

Node.js Express Backend API Overview

... +62 lines (ctrl+o to expand)

● Perfect! I've used both the frontend and backend MCP agents from your swarm to explain the application. Here's

the coordinated explanation:

🎯 Complete Application Overview (via Swarm Agents)

Frontend Agent's Explanation

The React frontend is a single-page task manager application:

Key Components:

- App.jsx - Main component managing all state and CRUD operations
- main.jsx - React 18 entry point with StrictMode
- Beautiful gradient UI with purple theme

[중략]

Backend Agent's Explanation

The Node.js backend provides a RESTful API:

5 Endpoints:

- GET /api/health - Health check
- GET /api/tasks - Get all tasks
- GET /api/tasks/:id - Get single task
- POST /api/tasks - Create task (requires title)
- PUT /api/tasks/:id - Update task
- DELETE /api/tasks/:id - Delete task

[중략]

</Claude 박스>

로그를 보면 각 **frontend**, **backend** 인스턴스를 통해서 질의를 수행하는걸 알 수 있습니다. 이 과정은 마치 클로드 코드의 서브 에이전트와 유사한 성격을 갖습니다. 사실 **Claude Swarm**은 클로드 코드의 커스텀 서브 에이전트가 있기 이전부터 존재했었습니다. 실제 동작은 위에서 확인되는것 처럼 내부적으로 각 인스턴스를 **MCP**형태로 띄워서 리드역할의 인스턴스가 이를 활용하도록 하는게 핵심 아이디어 입니다. 이 부분은 또 다음과 같이 확인해볼 수 있습니다.

```
> /mcp

Backend MCP Server

Status: ✓ connected
Command: claude-swarm
Args: mcp-serve --name backend --directory /Users/gavin.jeong/swarm/backend --model sonnet --description Backend
      developer managing APIs and data layer --allowed-tools Edit,Write,Bash --mcp-config-path /Users/gavin.jeong
      /.claude-swarm/sessions/Users+gavin.jeong+swarm/d99a7e79-522b-4631-a1c8-26bdb1e14fa9/backend.mcp.json
      --calling-instance lead --calling-instance-id lead_808e0a66 --instance-id backend_cac736ca
Config location: Dynamically configured
Capabilities: tools
Tools: 3 tools

> 1. View tools
   2. Reconnect
   3. Disable

Esc to go back
```

MCP로 **backend**의 설정을 살펴볼 수 있습니다. 제공하는 툴도 살펴봅니다.

```
Tools for backend (3 tools)

> 1. task          read-only
   2. session_info
   3. reset_session

Esc to go back
```

3개의 툴을 제공하는걸 알 수 있습니다. **task**에 대해 좀 더 상세한 내용을 살펴봅니다.

```
task (backend) [read-only]

Tool name: task
Full name: mcp__backend__task

Description:
Execute a task using Agent backend. Backend developer managing APIs and data layer Thinking budget levels:
"think" < "think hard" < "think harder" < "ultrathink".

Parameters:
• prompt (required): string - The task or question for the agent
• new_session: boolean - Start a new session (default: false)
• system_prompt: string - Override the system prompt for this request
• description: string - A description for the request
• thinking_budget: string - Thinking budget: "think" < "think hard" < "think harder" < "ultrathink". Each
level increases Claude's thinking allocation. Auto-select based on task complexity.
```

우리가 설정으로 넣어준 **description**을 토대로 **prompt**로 주어진 태스크를 처리한다는걸 알 수 있습니다.

커스텀 서브에이전트와의 차이점

앞서 살펴본 동작은 클로드 코드의 서브 에이전트와 매우 유사해보입니다. 그럼 굳이 **Claude Swarm**을 왜 써야할까요? 직관적으로 알 수 있는건 서브 에이전트와 달리 아주 독립된 에이전트로 각 인스턴스가 동작한다는 점입니다. 이를 활용하면 서브 에이전트와 차별되는 지점을 찾을 수 있습니다. 대표적인 예로 **MCP** 설정을 자유롭게 달리 가져갈 수 있는 부분을 생각해 볼 수 있습니다. 커스텀 서브에이전트의 경우 기본적으로 클로드 코드 인스턴스에서 설정된 **MCP**를 사용할 수 있게 되어 있습니다. 이는 즉 메인 세션에서 쓰지 않는 **MCP**는 서브 에이전트로 설정해줄 수는 없다는 것입니다. 결과적으로 서브에이전트에서만 필요한 **MCP**를 전달할 방법이 없습니다. 하지만 클로드 스웜의 모델은 이런 제약이 없어 상위 인스턴스에서 쓰이지 않을 **MCP**는 생략하고 하위 인스턴스에서 쓸 **MCP**만을 따로

독립적으로 설정하여 상위 인스턴스에서 MCP로 인한 토큰 낭비를 줄일 수 있는 장점이 있습니다. 물론 이게 결과적으로 전체 토큰 사용량을 줄일 수 있을지에 대해선 의문이 남을 수 있습니다. 이는 토큰 절약의 관점 보단 각 에이전트가 독립적으로 사용가능한 메모리의 한계를 늘리는데 좀 더 의의가 있다고 볼 수 있겠습니다. 마찬가지로 Hook과 같은 요소도 역시 별도로 구성할 수 있습니다. 구성과 관련된 다양한 예시는 공식 문서를 참조하세요.

<https://github.com/parruda/swarm/blob/main/docs/v1/README.md#mcp-server-types>

마무리

여기까지 다시 아래와 같이 정리해볼 수 있습니다.

구분	사용 맥락	장단점	비고
커스텀 서브 에이전트	클로드 코드 내장	별도의 패키지 설치가 필요없음. 모든 제어가 클로드 코드 내에서 완결. MCP나 hook 설정을 에이전트별로 따로 가져가기 어려움.	
Claude Squad	단일 리포에서 병렬 태스크를 손쉽게 처리하기 위한 도구	Git worktree나 Tmux와 같은 터미널 멀티플렉서에 대한 사용 부담을 줄여줌. 아직까지 자잘한 버그가 많은편.	Gemini, Codex, Aider도 지원
Claude Swarm	완전히 독립적인 Claude Code 인스턴스를 오케스트레이션 하기 위한 도구	완전히 독립적인 멀티 에이전트 오케스트레이션. 빠르게 변경되고 있으며 아직 성숙도가 부족.	v2인 Swarm에선 OpenAI 등 다른 LLM 모델도 인스턴스 별로 구성 가능. 보다 벤더 중립적으로 멀티 에이전트 오케스트레이션이 가능함.

에이전트 기반 SW엔지니어링이 도입되면서 자연스럽게 엔지니어들은 여러 종류의 업무를 병렬로 처리하여 생산성을 높일것을 주문받고 있는 상황입니다. 이에 각 상황에 맞도록 다양한 전략으로 병렬로 에이전트를 다루는 오케이스트레이션은 이제 필연적으로 준비해야할 스킬로 자리잡고 있습니다. 여기서 클로드 코드에 맞게 터미널을 중심으로하는 2가지 오케스트레이션 방법에 대해 기술해봤지만 지금도 끊임없이 새로운 도구와 방법론이 만들어지고 실험되고 있습니다. 어느정도 표준화된 방법론이 나오기전까지 각자의 스타일에 맞는 워크플로우를 잘 정립하고 이를 스케일하는 여러 실험을 꾸준히 해보는걸 추천드립니다.