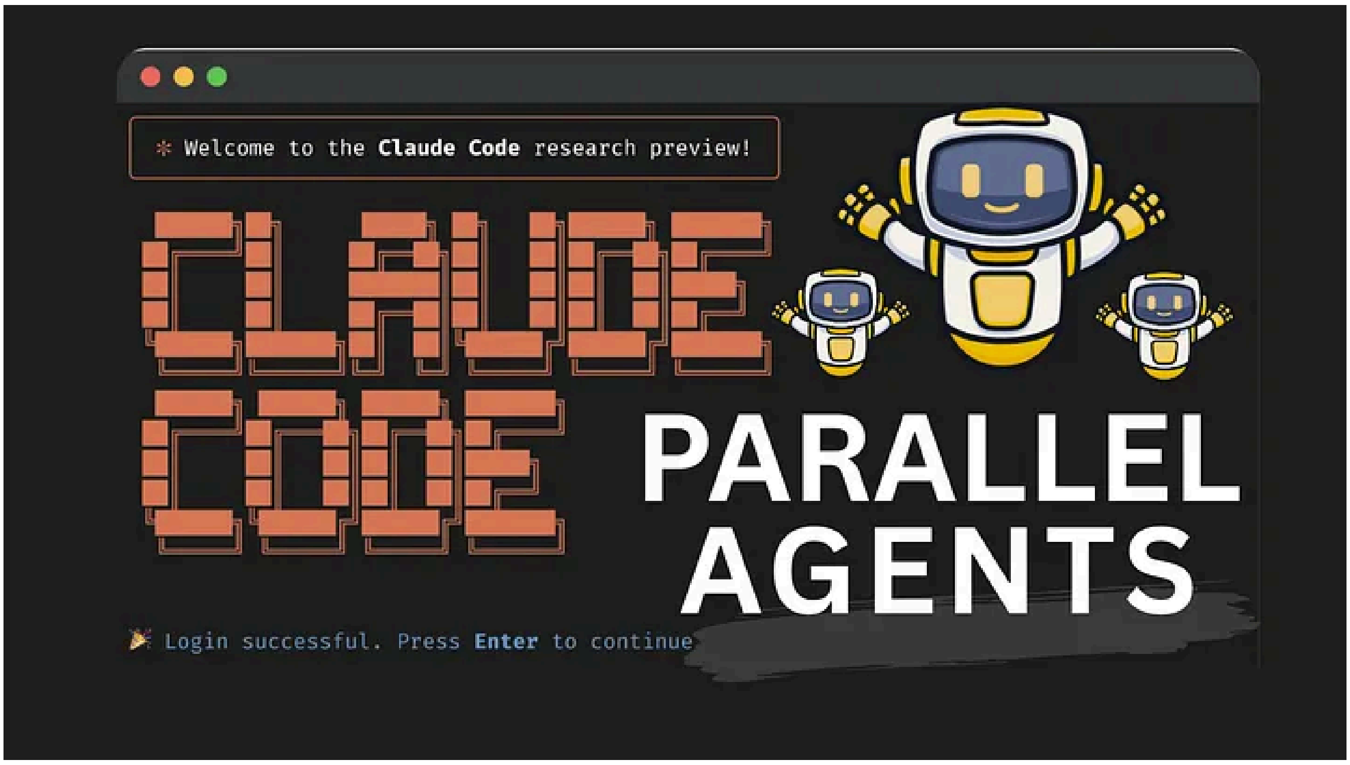
How I’m Using Claude Code Parallel Agents to Blow Up My Workflows


Welcome to the Claude Code research preview!
CLAUDE CODE
PARALLEL AGENTS
Login successful. Press Enter to continue

Claude Code Parallel Agents — Featured Image / By Author

If you are not using parallel agents on Claude Code, you are wasting time, or you don’t know how to use parallel agents to 10x your output.

> ***Luckily, in this post, I will demonstrate how you can use parallel agents in a real-world example and what you can achieve.***

If you are new to agents in Claude Code, ***we covered the subagent tutorial here***, and later shared practical examples that you can use as starter templates here — ***Claude Code subagent examples with templates***.

And,

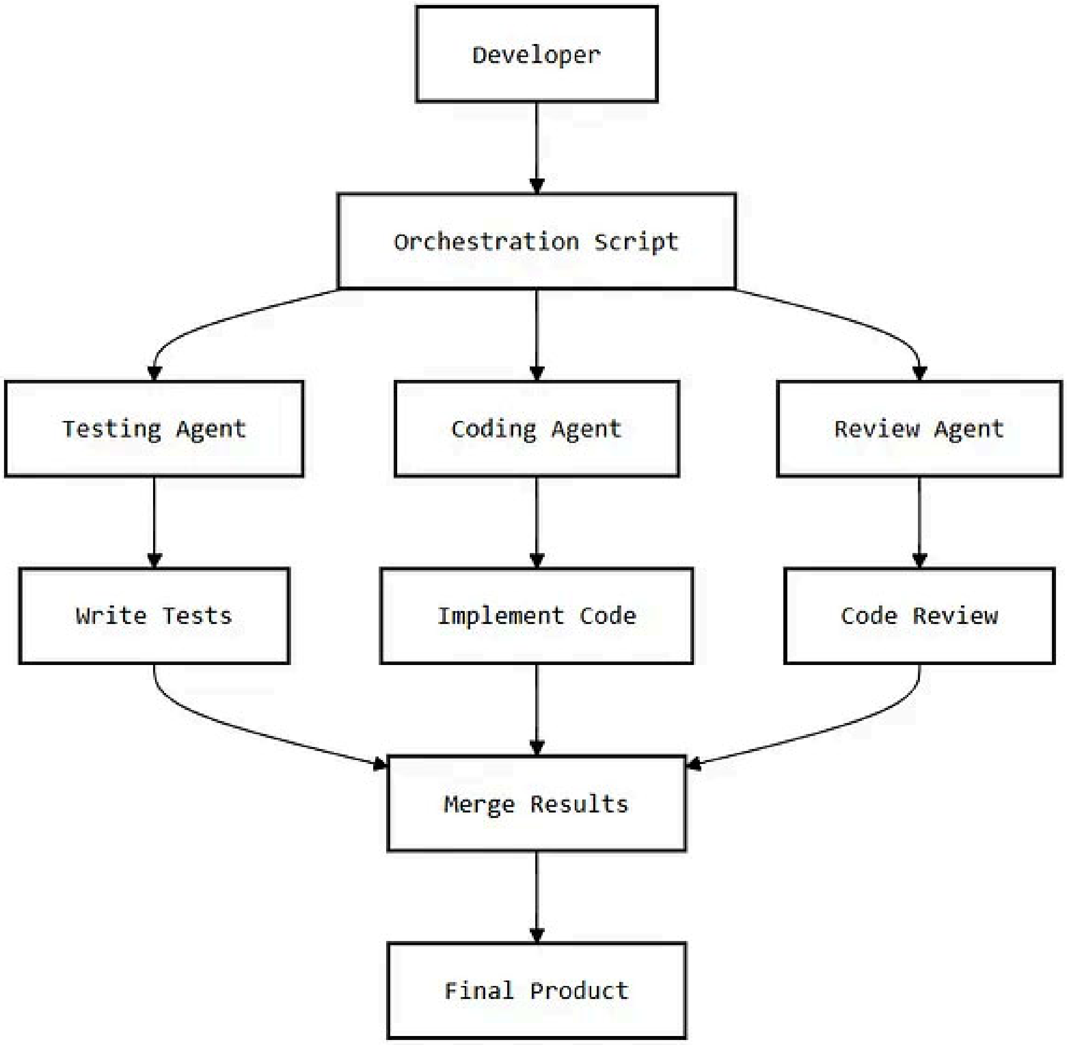
We are still updating the **Claude Code Cheat Sheet** with every example and trick that we come across to give you one repo with all these examples, tips, and use cases.

Quick Note: *If you are not a premium member, you can read the story here.* ***Also, consider supporting me by following me on Medium*** *and* ***my YouTube channel*** *to learn more about Claude Code.*

What are Parallel Agents?

The idea of parallel agents in Claude Code refers to ***running multiple Claude instances*** in parallel.

The easiest way to visualize how parallel agents work is this diagram :



Claude Code Parallel Agents Illustration / By Author

Using parallel agents is a simple but effective approach to have one Claude Code instance write code while another reviews and another tests it.

This is the same traditional way of working with multiple Software Engineers in one team.

- *Backend Engineer / Frontend Engineer*
- *Testing Engineer*
- *Code Review Engineer*

In a real-world scenario, you can imagine how this works. There is code versioning where each person works on different parts — like creating a branch, working on a branch, PRs and code reviews, merging, and such processes.

> *But in this case of parallel agents, how does all this work together ?*

Claude Code Parallel Agent Patterns

The key here is understanding **how multiple Claude instances coordinate and communicate** — which can be summarized in four key concepts:

1. Multi-Agent Code Review:

- One Claude instance writes code
- Another Claude instance reviews or tests the code
- A third instance can integrate feedback from both

2. Multiple Repository Checkouts:

- Create 3–4 git checkouts in separate folders
- Open each folder in separate terminal tabs
- Start Claude in each folder with different tasks
- Cycle through to check progress and approve/deny permission requests

3. Git Worktrees for Isolation:

- Git worktrees allow you to check out multiple branches from the same repository into separate directories.
- Each worktree has its working directory with isolated files, while sharing the same Git history.
- This enables running multiple Claude sessions simultaneously on different parts of your project.

4. Research and Planning Agents:

- Based on Anthropic's multi-agent research system — *How we built our multi-agent research system \ Anthropic*, Claude Code can use tools to create parallel agents that search for information simultaneously.

Another question that comes up — are parallel agents limited to opening multiple tabs only?

> *No,*

Claude Code supports parallel agents through:

- **Multiple terminal sessions** running different Claude instances
- **Shared context through CLAUDE.MD files** that provide consistent instructions
- **MCP (Model Context Protocol)** for tool integration across agents
- **Git worktrees** for isolated workspaces
- **Headless mode** for programmatic control

Up to this point, you fully understand the idea of parallel agents and how they work.

> ***But other than the simple example I shared, what would be examples of advanced workflows using Claude Code parallel agents?***

Advanced Parallel Agents Workflows Examples

Test-Driven Development with Multiple Agents:

- Agent 1: Writes comprehensive test cases
- Agent 2: Implements code to pass the tests
- Agent 3: Reviews both tests and implementation

Parallel Task Execution:

- You can request Claude to "research three separate ideas and do it in parallel. Use three agents to do it."

Headless Mode for Automation:

- Claude Code includes headless mode for non-interactive contexts like CI, pre-commit hooks, build scripts, and automation, enabling programmatic parallel agent orchestration.

Now that's done, let's implement!

Practical Claude Code Parallel Agents Project

At the start, I promised to give you a practical example***. You remember the diagram I shared earlier; we will implement it in this section.***

Our parallel agents will be like an engineering team with multiple specialized team members:

- **Agent 1**: The coder who writes the implementation
- **Agent 2**: The tester who creates comprehensive test suites
- **Agent 3**: The reviewer who catches bugs and suggests improvements
- **Agent 4**: The documentarian who keeps everything up-to-date

> ***The objective is to have each agent maintain its context and focus, leading to better results than trying to do everything with a single***

> ***Claude instance.***

Prerequisites

Before we start, make sure you have:

- Claude Code installed (`npm install -g @anthropic-ai/claude-code`)
- Git is configured on your machine.
- Your favorite terminal and code editor are ready.

Step 1: Project Structure Setup

First, let's create a proper structure for our multi-agent workflow:

```
# Create your main project directory
mkdir my-ai-team-project
cd my-ai-team-project

# Initialize git
git init

# Create the essential CLAUDE.md file
touch CLAUDE.md
```

Step 2: Configure Your Team's Shared Knowledge

Create a `CLAUDE.md` file that all your agents will reference:

```
# Project: AI Team Development

## Common Commands
- `npm test`: Run the test suite
- `npm run build`: Build the project
- `npm run lint`: Check code quality

## Code Style Guidelines
- Use TypeScript for all new files
- Follow ESLint rules strictly
- Write tests for all new functions
- Use descriptive variable names
```

```
## Testing Philosophy
- Write tests before implementation (TDD)
- Aim for 90%+ code coverage
- Include edge cases and error scenarios

## Git Workflow
- Create feature branches for each task
- Use descriptive commit messages
- Squash commits before merging

## Project Structure
- `/src` - Source code
- `/tests` - Test files
- `/docs` - Documentation
```

Step 3: Set Up Git Worktrees for Parallel Work

Git worktrees allow multiple Claude instances to work on different branches simultaneously:

```
# Create separate worktrees for different agents
git worktree add ../project-coding coding-agent
git worktree add ../project-testing testing-agent
git worktree add ../project-review review-agent

# Verify your worktrees
git worktree list
```

Real-World Example: Building a User Authentication System

Let me walk you through a practical example where I used three parallel agents to build a complete user authentication system.

Agent 1: The Test Writer

In terminal tab 1 (`project-testing` directory):

```
cd ../project-testing
claude
```

Prompt for Testing Agent:

```
You are the testing specialist. Your job is to write comprehensive tests for

Create tests for:
1. User registration with email validation
2. Password hashing and verification
3. Login with JWT token generation
4. Password reset functionality
5. Rate limiting for login attempts

Write the tests FIRST - don't implement any functionality yet. Use Jest and f
```

Generated Test Code:

```
// tests/auth.test.js
const { AuthService } = require('../src/auth-service');
const jwt = require('jsonwebtoken');

describe('AuthService', () => {
  let authService;

  beforeEach(() => {
    authService = new AuthService();
  });

  describe('registerUser', () => {
    test('should register user with valid email and password', async () => {
      const user = await authService.registerUser('test@example.com', 'Passwo
      expect(user).toHaveProperty('id');
      expect(user.email).toBe('test@example.com');
      expect(user.password).not.toBe('Password123!'); // Should be hashed
    });

    test('should reject invalid email formats', async () => {
      await expect(authService.registerUser('invalid-email', 'Password123!'))
        .rejects.toThrow('Invalid email format');
    });

    test('should reject weak passwords', async () => {
      await expect(authService.registerUser('test@example.com', '123'))
        .rejects.toThrow('Password must be at least 8 characters');
    });
  });

  describe('loginUser', () => {
    test('should login with correct credentials', async () => {
      // Setup
      await authService.registerUser('test@example.com', 'Password123!');

      // Test
      const result = await authService.loginUser('test@example.com', 'Passwor
      expect(result).toHaveProperty('token');
      expect(jwt.verify(result.token, process.env.JWT_SECRET)).toBeTruthy();
    });

    test('should reject incorrect password', async () => {
      await authService.registerUser('test@example.com', 'Password123!');

      await expect(authService.loginUser('test@example.com', 'WrongPassword')
        .rejects.toThrow('Invalid credentials');
    });
  });
});
```

Agent 2: The Implementation Specialist

In terminal tab 2 (`project-coding` Directory):

```
cd ../project-coding
claude
```

Prompt for Coding Agent:

```
You are the implementation specialist. Look at the tests in ../project-testir

Requirements:
- Use bcrypt for password hashing
- Use jsonwebtoken for JWT tokens
- Implement proper email validation
- Add rate limiting for login attempts
- Follow the existing code style guidelines in CLAUDE.md

Do NOT modify the tests - only implement the functionality.
```

Generated Implementation Code:

```
// src/auth-service.js
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const validator = require('validator');

class AuthService {
  constructor() {
    this.users = new Map(); // In production, use a database
    this.loginAttempts = new Map();
    this.MAX_LOGIN_ATTEMPTS = 5;
    this.LOCKOUT_TIME = 15 * 60 * 1000; // 15 minutes
  }

  async registerUser(email, password) {
    // Validate email
    if (!validator.isEmail(email)) {
      throw new Error('Invalid email format');
    }

    // Validate password strength
    if (password.length < 8) {
      throw new Error('Password must be at least 8 characters');
    }

    // Check if user already exists
    if (this.users.has(email)) {
      throw new Error('User already exists');
    }

    // Hash password
    const hashedPassword = await bcrypt.hash(password, 12);

    // Create user
    const user = {
      id: Date.now().toString(),
      email,
      password: hashedPassword,
      createdAt: new Date()
    };

    this.users.set(email, user);
    // Return user without password
    const { password: _, ...userWithoutPassword } = user;
    return userWithoutPassword;
  }

  async loginUser(email, password) {
    // Check rate limiting
    this._checkRateLimit(email);

    const user = this.users.get(email);
    if (!user) {
      this._recordFailedAttempt(email);
      throw new Error('Invalid credentials');
    }

    const isValidPassword = await bcrypt.compare(password, user.password);
    if (!isValidPassword) {
      this._recordFailedAttempt(email);
      throw new Error('Invalid credentials');
    }

    // Reset login attempts on successful login
    this.loginAttempts.delete(email);

    // Generate JWT token
    const token = jwt.sign(
      { userId: user.id, email: user.email },
      process.env.JWT_SECRET || 'fallback-secret',
      { expiresIn: '24h' }
    );

    return { token, user: { id: user.id, email: user.email } };
  }

  _checkRateLimit(email) {
    const attempts = this.loginAttempts.get(email);
    if (attempts && attempts.count >= this.MAX_LOGIN_ATTEMPTS) {
      const timeSinceLastAttempt = Date.now() - attempts.lastAttempt;
      if (timeSinceLastAttempt < this.LOCKOUT_TIME) {
        throw new Error('Too many login attempts. Try again later.');
      } else {
        // Reset attempts after lockout period
        this.loginAttempts.delete(email);
      }
    }
  }

  _recordFailedAttempt(email) {
    const attempts = this.loginAttempts.get(email) || { count: 0, lastAttempt
    attempts.count++;
    attempts.lastAttempt = Date.now();
    this.loginAttempts.set(email, attempts);
    }
  }

  module.exports = { AuthService };
```

Agent 3: The Code Reviewer in Action

In terminal tab 3 (`project-review` directory):

```
cd ../project-review
claude
```

Prompt for Review Agent:

```
You are the code review specialist. Review the implementation in ../project-c

Focus on:
- Security vulnerabilities
- Code quality and best practices
- Performance issues
- Missing edge cases
- Documentation needs

Provide specific improvement suggestions.
```

Review Agent's Feedback:

```
# Code Review Report - Auth Service

## Critical Issues

1. **Security Risk**: Fallback JWT secret in production
   - Line 63: `process.env.JWT_SECRET || 'fallback-secret'`
   - **Fix**: Throw error if JWT_SECRET not set

2. **Memory Leak**: In-memory storage will grow indefinitely
```

~~~
   - Users Map and loginAttempts Map never cleaned up
   - **Fix**: Implement cleanup mechanism or use database

## Improvements Needed

1. **Input Sanitization**: Email should be normalized
   ```javascript
   email = email.toLowerCase().trim();
~~~

Agent 4: The Documentation Specialist

In terminal tab 4 (`main project` directory):

Prompt for Documentation Agent:

```
You are the documentation specialist. Based on the code in the other worktree

1. README.md with setup instructions
2. API documentation
3. Code comments for complex functions
4. Security guidelines

Make it production-ready documentation.
```

Generated Documentation:

```
# User Authentication System

## Overview
A secure, production-ready authentication system with JWT tokens, password ha

## Features
- ✅ Email validation
- ✅ Secure password hashing (bcrypt)
- ✅ JWT token generation
- ✅ Login rate limiting
- ✅ Comprehensive test coverage

## Quick Start

### Installation
```

~~~
```bash
npm install bcrypt jsonwebtoken validator jest
~~~

This has just been a brief introduction to what is possible with Claude Code working with parallel agents.

> ***We have kept it very basic and have not even explored how to use this with the recently launched subagents.***

This is a good place to start learning how Claude Code works under the hood and getting started with implementing it in your projects.

But this is not enough, to save your money and get it working right, you should consider learning more tips from my upcoming course — ***Claude Code Masterclass.*** (See the details below)

Final Thoughts

Using Claude Code, parallel agents come with these benefits :

1. **Separation of Concerns**: Different agents can focus on specific tasks (coding vs. testing vs. reviewing)
2. **Faster Iteration**: Multiple tasks can proceed simultaneously
3. **Better Context Management**: Each agent maintains a focused context for its role
4. **Improved Quality**: Multiple perspectives on the same codebase

This multi-agent approach transforms Claude Code from a single assistant into a collaborative team of AI agents, each specialized for different aspects of the development workflow.

Claude Course Course

> ***Every day I'm working hard on building the ultimate Claude Code course that demonstrates how to build workflows that coordinate multiple agents for complex development tasks. It's due for release soon.***

It will take what you have learned from this article to the next level of complete automation.

New features are added to Claude Code daily, and keeping up is tough.

The course explores subagents, hooks, advanced workflows, and productivity techniques that many developers may not discover.

Once you join, you'll get all the updates as new features roll out.

This course will cover:

- *Advanced subagent patterns and workflows*
- *Production-ready hook configurations*
- *MCP server integrations for external tools*
- *Team collaboration strategies*
- *Enterprise deployment patterns*
- *Real-world case studies from my consulting work*

If you're interested in getting notified when the Claude Code course launches, **click here to join the early access list →**

(*Currently, I have 2000+ already signed-up developers)*

> ***I'll share exclusive previews, early access pricing, and bonus materials with people on the list.***