# How I Built an AI Development Team on Top of Claude Code

Creating a coordinated team of specialist agents that communicate through markdown files

14 min read · Oct 12, 2025

Mohammed H. Jabreel Follow

Claude Code is Anthropic's AI-powered coding assistant that can write, edit, and understand codebases. But what happens when you need to build something complex that spans multiple domains — backend APIs, frontend UIs, database schemas, tests, and documentation all at once?

The default approach has Claude handle everything sequentially, context-switching between concerns. I wanted something better: **a team of specialized Claude agents, each an expert in their domain, coordinating through structured files.**

This is the story of how I built a development team framework on top of Claude Code — and why I chose markdown files over databases.

. . .

## The Problem: One Agent, Too Many Hats

Even with Claude Code's impressive capabilities, asking a single agent instance to build a full-stack feature creates challenges:

- **Context switching overhead** —jumping between backend models, frontend components, database schemas, API contracts, and test files. For example, implementing "user authentication" requires touching: domain models (User, Session), database migrations, API endpoints, login UI, token validation middleware, unit tests, and integration tests. That's at least 7 different concerns in one conversation.

- **Inconsistent patterns** — backend code might use one architectural style (say, Domain-Driven Design) while frontend uses another (Component-Based Architecture). A single agent juggling both often produces inconsistent implementations.
- **Lost coordination** — no systematic way to track dependencies or blockers across domains. Did the database schema get created before the backend tried to use it? Did the API contract match what the frontend expects? Manual tracking gets messy.
- **Session dependency** — complex projects require multiple conversations, losing context between sessions. You describe requirements in session 1, implement in session 2, but by session 3, Claude has forgotten the original design decisions.

Real software teams don't work this way. Real teams have **specialists** who *coordinate* through *documented communication*.

## The Solution: Leveraging Claude Code's Agent System

Claude Code already supports custom agents through `.claude/agents/` configuration files. Each agent is a specialized instance of Claude with:

- **Specific expertise** defined in its configuration
- **Custom instructions** for its domain
- **Access to the same tools** (Read, Write, Edit, Bash, etc.)
- **Invocable via the Task tool** from other agents

I built on this foundation to create a **coordination framework** where:

- A **Project Manager** agent (replacing the default orchestrator) coordinates the team
- **Specialist agents** (backend, frontend, database, QA, etc.) handle implementation
- A **User Proxy** agent bridges user intent and technical execution

- **Communication happens through markdown files** in a `communication/` directory
- **Zero reliance on memory** — everything persists in version-controlled files
- **Custom slash commands** (like `/init-dev-team`) automate setup and workflows

### Custom Slash Commands in Claude Code

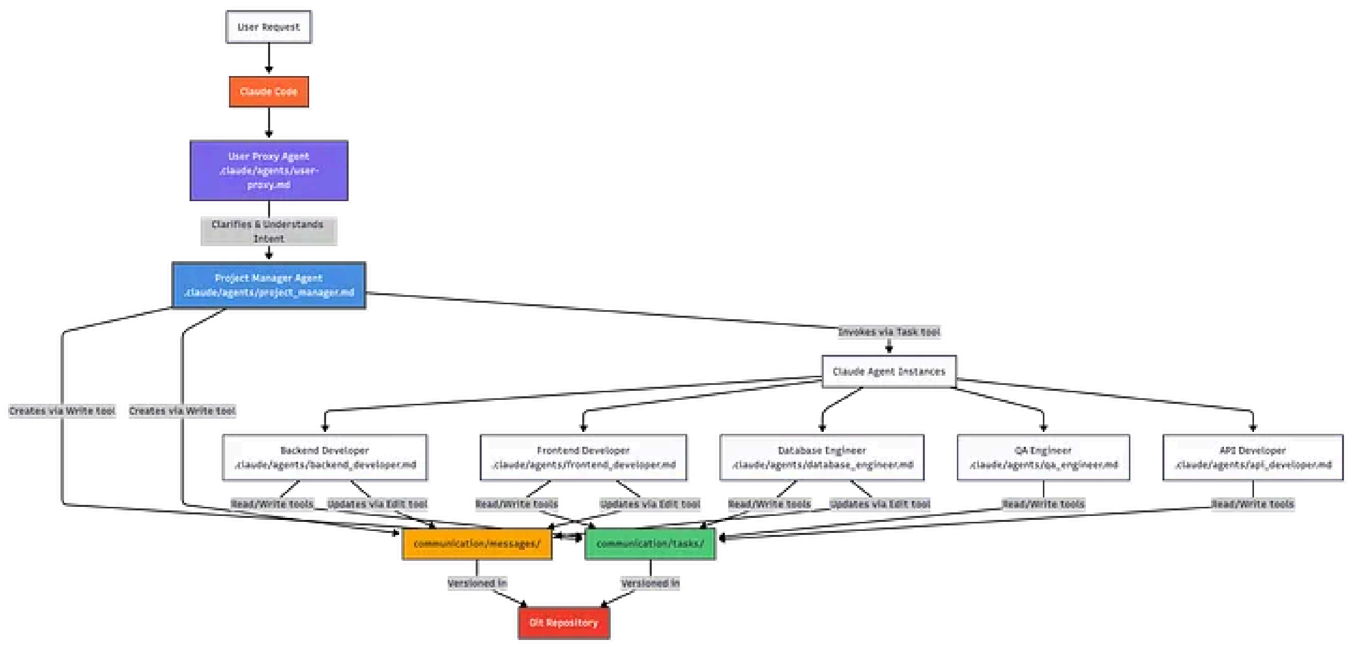
Claude Code supports custom commands via `.claude/commands/` directory. I leveraged this to provide:

**/init-dev-team** — One-command setup that:
- Asks about your project (tech stack, domains, architecture)
- Creates all agent configurations with your specifics
- Sets up the communication directory structure
- Initializes helper scripts
- Customizes everything for your tech stack

**`/delegate`** — Delegate tasks to the team:
- Routes request to user-proxy
- User-proxy understands intent
- Invokes project manager
- Team executes in parallel

This makes the framework feel like an extension of Claude Code, not a separate tool.

### The Architecture: Claude Agents + File Communication


User Request
Claude Code
User Proxy Agent .claude/agents/user-proxy.md
Clarifies & Understands Intent
Project Manager Agent .claude/agents/project_manager.md
Invokes via Task tool
Claude Agent Instances
Creates via Write tool
Creates via Write tool
Backend Developer .claude/agents/backend_developer.md
Frontend Developer .claude/agents/frontend_developer.md
Database Engineer .claude/agents/database_engineer.md
QA Engineer .claude/agents/qa_engineer.md
API Developer .claude/agents/api_developer.md
Read/Write tools
Updates via Edit tool
Read/Write tools
Updates via Edit tool
Read/Write tools
Updates via Edit tool
Read/Write tools
Read/Write tools
communication/messages/
communication/tasks/
Versioned in
Versioned in
Git Repository

**How Claude Code Invokes Agents**

When you configure agents in `.claude/agents/`, Claude Code makes them available via the `Task` tool:

```
Task({
  subagent_type: "backend_developer",  // Matches filename in .claude/agents/
  description: "Implement Article model",
  prompt: "Check communication/tasks/p1-open-backend_developer-T001.md and begi
})
```

Each agent runs as **a separate Claude instance** with:
- Configurable model (Sonnet, Opus, or Haiku — specified in agent config)
- Custom tools (Read, Write, Edit, Bash, Grep, etc.) — optional.
- Custom instructions from its `.md` configuration file
- Access to the shared filesystem (the `communication/` directory)

This is the key insight: **agents coordinate through files, not through Claude's context window.**

## Core Principles

### 1. File-Based Communication (Zero Memory Dependency)

The entire system operates on a simple but powerful principle: **agents never rely on memory**. All coordination happens through files in the `communication/` directory:

```
communication/
├── tasks/          # Task assignments
├── messages/       # Inter-agent messages
├── status/         # Current agent states
├── domains/        # Domain knowledge
├── _templates/     # File templates
└── _index/         # Quick summaries
```

Every interaction is persisted. Every decision is documented. Every status update is recorded. This means:

- **No context lost** — restart anytime without losing state
- **Full auditability** — complete history of who did what and why
- **Version control friendly** — track changes with git
- **Human readable**— just open the markdown files

### 2. Filename-Based Metadata (Smart Naming)

Here's where it gets clever: we encode critical information directly in filenames so agents can find what they need **without reading every file**.

**Task Filename Pattern**

```
p1-open-backend_developer-T001-implement-feature.md
│   │     │                    │    └─ Description
│   │     │                    └────── Task ID
│   │     └─────────────────────────── Agent name
│   └───────────────────────────────── Status
└───────────────────────────────────── Priority
```

**Message Filename Pattern**

```
20251009-1030-orchestrator-to-backend_developer-new-task_assigned.md
│             │            │  │                  │   └─ Type
│             │            │  │                  └───── Status
│             │            │  └──────────────────────── Recipient
│             │            └─────────────────────────── Sender
│             └──────────────────────────────────────── Timestamp
│
```

This enables instant lookups using shell glob patterns:

```
# Find all tasks for backend developer
ls communication/tasks/*-backend_developer-*.md

# Find all blocked tasks
ls communication/tasks/*-blocked-*.md
```

```
# Find unread messages to specific agent
ls communication/messages/*/*-to-backend_developer-new-*.md
```

**No database. No queries. Just filesystem patterns.** Blazingly fast and universally compatible.

### 3. Template-Based Consistency

All files are created from templates. No freestyle markdown. This ensures:

- Consistent structure across all communications
- Required fields are never missing
- Agents always know where to find information
- Easy to parse programmatically

```
# Create a task
./scripts/create-task.sh p1 backend_developer T001 implement-auth

# Send a message
./scripts/create-message.sh orchestrator backend_developer task_assigned

# Update task status
./scripts/update-task-status.sh T001 in_progress
```

### 4. The Coordination Model

. . .

## How It Works in Practice

Let’s walk through a real example: implementing article management for a news platform.

**Step 1: User Request**

> *User: “I need article management with categories and tags”*

**Step 2: User Proxy Clarifies**

The user-proxy agent understands intent and asks clarifying questions if needed:

- What workflows are required? (Create, edit, publish, archive?)
- Who are the users? (Admin panel? Public site?)
- Any specific requirements? (Arabic support? SEO?)

### Step 3: Project Manager Coordinates

The project manager breaks this down:

```
**Agents Involved:**
- Backend Developer: Domain models
- Database Engineer: Schema design
- Frontend Developer: Admin UI
- API Developer: Mobile endpoints
- QA Engineer: Integration tests
**Task Breakdown:**
- T001 (Backend): Article aggregate with DDD patterns
- T002 (Database): Articles table with full-text search
- T003 (Frontend): Article CRUD interface
- T004 (API): RESTful endpoints
- T005 (QA): Test coverage
**Dependencies:**
T003, T004, T005 all depend on T001 completion
```

### Step 4: Task Files Created

The project manager creates structured task files:

**File:** `communication/tasks/p1-open-backend_developer-T001-article-aggregate.md`

```
---
task_id: T001
priority: p1
status: open
assigned_to: backend_developer
created: 2025-10-09T22:35:00Z
domain: Content
type: feature_development
estimated_hours: 8
---

# Task: Implement Article Aggregate Root

## Description
Implement a complete Article aggregate following Domain-Driven Design
principles for the Arabic news platform.

## Requirements
- Create Article aggregate root with proper encapsulation
- Implement value objects: Title, Content, Slug, Excerpt
- Define domain events (ArticleCreated, ArticlePublished, etc.)
- Create repository interface
- Support Arabic content with RTL considerations

## Dependencies
None - foundational task

## Acceptance Criteria
- [ ] Article aggregate root created
- [ ] Value objects implemented
- [ ] Domain events defined
- [ ] Repository interface created
- [ ] Business rules enforced
- [ ] Arabic content supported
```

**Step 5: Agents Invoked via Claude Code's Task Tool**

The project manager uses Claude Code's **Task tool** to spawn new agent instances:

```
Task({
  subagent_type: "backend_developer",  // References .claude/agents/backend_dev
  description: "Implement Article aggregate",
  prompt: "Check communication/tasks/p1-open-backend_developer-T001-article-agg
})
```

Behind the scenes, Claude Code:
1. Reads `.claude/agents/backend_developer.md` for the agent's configuration
2. Spawns a new Claude instance with those instructions
3. Passes the prompt to that instance
4. Returns control once the agent completes its work

The backend developer agent is now running **independently** with full access to the codebase.

**Step 6: Specialists Work (Using Claude Code Tools)**

Each agent instance uses Claude Code's built-in tools:

1. **Check for unread messages:** Uses `Bash` tool to run `./scripts/list-unread-messages.sh backend_developer` (or manually: `ls communication/messages/*/*-to-

backend_developer-new-*.md`)
2. **Read assigned task:** Uses `Read` tool to read `communication/tasks/*-backend_developer-*.md`
3. **Update task status:** Uses `Edit` tool to change `status: open` → `status: in_progress` (or run `./scripts/update-task-status.sh T001 in_progress`)
4. **Implement the feature:** Uses `Write`, `Edit`, `Read` tools to create code files
5. **Send completion message:** Uses `Bash` tool to run `./scripts/create-message.sh backend_developer project_manager task_completed`
6. **Update task status:** Uses `Edit` tool to mark `status: completed` (or run `./scripts/update-task-status.sh T001 completed`)

Every action uses Claude Code's standard tool API — no custom infrastructure needed. The helper scripts are just bash wrappers that make file operations easier.

**Step 7: Results**

*Backend Developer* delivers:

```
├── ValueObjects/
│   ├── Title.php
│   ├── Slug.php
│   ├── Content.php
│   └── Excerpt.php
├── Events/
│   ├── ArticleCreated.php
│   ├── ArticlePublished.php
│   └── ...
├── Repositories/
│   └── ArticleRepositoryInterface.php
└── Enums/ArticleStatus.php
```

With full documentation, type hints, business rules, and Arabic support.

## The Task Lifecycle

# The Message Workflow

## Real-World Benefits

### 1. No Context Loss

Traditional AI assistant:

```
User: “Build article management”
AI: [builds feature]
[Session ends]
[New session]
User: “Add categories”
AI: “What article management?” ❌
```

With this framework:

```
User: “Add categories”
```

Agent: [checks communication/domains/Content/]
Agent: "I see we have Article implemented. I'll extend it." ✅
```

**2. Parallel Work**

The project manager can invoke multiple agents simultaneously:

```
// All invoked in parallel
Task({ subagent_type: "backend_developer", … })
Task({ subagent_type: "database_engineer", … })
Task({ subagent_type: "qa_engineer", … })
```

Agents work concurrently and coordinate through messages when dependencies are met.

**3. Audit Trail**

Every decision, every task, every message is documented:

```
git log communication/
# Complete history of who did what and when
git diff HEAD~1 communication/tasks/
# See exactly what changed in task assignments
```

**4. Human Oversight**

Because everything is markdown files, humans can:

- **Review** task assignments before agents start
- **Modify** requirements mid-execution
- **Override** agent decisions
- **Audit** the entire workflow later

**5. Specialization = Quality**

Instead of one generalist agent, you get:

- **Backend Developer** who deeply understands your architecture patterns (DDD, Clean Architecture, MVC) and backend framework (Django, Laravel, Spring
```

Boot, Express.js)

- **Database Engineer** who optimizes database queries and schema design for your specific database (PostgreSQL, MySQL, MongoDB, etc.)
- **Frontend Developer** who knows your UI framework's patterns and best practices (React, Vue, Angular, Svelte)
- **QA Engineer** who writes tests in your testing framework (Jest, Pytest, PHPUnit, JUnit)

Each agent stays in their zone of expertise. No context switching. Better output quality.

## System Architecture Deep Dive

### The Communication Directory

**Helper Scripts**

The framework includes bash scripts for common operations:

```
# Create a new task
./scripts/create-task.sh <priority> <agent> <task_id> <slug>

# Send a message between agents
./scripts/create-message.sh <from> <to> <type>

# Update task status
./scripts/update-task-status.sh <task_id> <new_status>

# List all tasks (with filtering)
./scripts/list-tasks.sh [status|agent|priority]

# Check agent availability
./scripts/check-agent-status.sh [agent_name]
```

These scripts ensure consistency and make it easy for both agents and humans to interact with the system.

## Workflow Visualization

### From Request to Delivery

## Key Design Decisions

### Why Files Instead of a Database?

- **Simplicity**: No schema, no migrations, no query language. Just text files.
- **Portability**: Works anywhere — no dependencies, no setup.
- **Version Control**: Full git integration for free.
- **Human Readable**: Any developer can inspect the state with `cat` and `ls`.
- **Zero Latency**: Filesystem operations are instant. No network, no connection pooling.

### Why Markdown?

- **Structured Yet Flexible**: YAML frontmatter for metadata, markdown for content.
- **Tooling**: Every editor, IDE, and platform supports it.
- **Readable**: Both machines and humans can parse it easily.
- **Rich Content**: Code blocks, lists, tables — everything you need for technical communication.

### Why Filename-Based Metadata?

- **Fast Lookups**: Find tasks with `ls *-backend_developer-*` instead of reading files.
- **Self-Documenting**: The filename tells you priority, status, agent, and description.
- **Atomic Renames**: Changing task status is just `mv old-name new-name`.
- **Glob Patterns**: Leverage shell wildcards for powerful queries.

## Challenges and Limitations

### 1. Token Costs: The Elephant in the Room

**This is expensive.** Let me be blunt about the biggest limitation:

Each agent invocation spawns a **new sub-agent instance** that:
- Reads its configuration file (`.claude/agents/agent-name.md`)
- Reads task files, message files, status files
- Generates responses and writes files
- All of this consumes tokens

**Why so expensive?**

1. *Context loading*: Each agent reads multiple markdown files to understand the project state
2. *Agent coordination*: Messages between agents = more reads and writes
3. *No shared memory*: Every agent starts from scratch, reading files
4. *Parallel execution*: Multiple agents running simultaneously multiplies costs

**When it's worth it:**
- Large, complex features where coordination saves you days of work
- Projects where mistakes are expensive (the agents catch issues)
- Learning and prototyping new architectures
- When you value your time more than API costs

**When it's NOT worth it:**
- Simple CRUD operations
- Single-file changes
- Quick bug fixes
- Repetitive tasks that don't need coordination

**Not Everything Needs This**

For simple, single-file tasks, invoking multiple agents is overkill. Use the framework when:

- Multiple domains are involved (backend + frontend + database)
- Dependencies between tasks exist (feature depends on schema)
- Long-running work needs tracking (multi-day projects)
- Team coordination is required (5+ specialists needed)

*Rule of thumb:* If a single Claude Code session could handle it in one go, don't use the team framework.

#### 2. Agent Understanding

Agents must be trained to follow the communication protocol. Clear documentation in `.claude/agents/` files is critical. Poorly configured agents will:
- Skip reading messages
- Not update status files
- Create tasks incorrectly
- Waste tokens on unnecessary operations

#### 3. Debugging Multi-Agent Issues

When something goes wrong:
- Which agent caused the issue?
- Was it a communication breakdown?
- Did an agent misunderstand its task?

## Getting Started

#### Prerequisites

You need:
- **Claude Code** (from claude.ai/code)
- A project directory
- Git (for version control)

#### Quick Start: The `/init-dev-team` Command

The framework includes a **custom slash command** that sets up everything automatically:

1. **Clone the framework:**

```
git clone https://github.com/mhjabreel/ai-dev-team-framework.git
cd ai-dev-team-framework
```

2. **Open in Claude Code:**

```
claude
```

3. **Run the initialization command:**

```
/init-dev-team
```

4. **Answer the prompts:**

```
Claude: What's your project name?
 You: TaskFlow - Project Management SaaS
Claude: What tech stack are you using?
 You: Node.js 20, Express, React 18, PostgreSQL
Claude: What are the key domains in your project?
 You: Projects, Tasks, Users, Notifications
Claude: What architecture pattern?
 You: Clean Architecture with Domain-Driven Design
```

**5. Team automatically created:**

✅ AI Development Team initialized for 'TaskFlow'!

Team Created:
🎯 user-proxy — Your advocate, interprets intent
🎛 project_manager — Coordinates team tasks
📋 system-analyst — Defines schemas and contracts
⚙️ backend-developer — Node.js + Express + Clean Architecture specialist
🎨 frontend-developer — React 18 + TypeScript specialist
🗄 database-engineer — PostgreSQL optimization specialist
🧪 qa-engineer — Jest + Supertest testing
📝 technical-writer — Documentation

Domains Created:
— Projects (owned by backend-developer)
— Tasks (owned by backend-developer)
— Users (owned by backend-developer)
— Notifications (owned by backend-developer)

Ready for work! Just tell me what you want to build.

**What `/init-dev-team` Creates**

The command automatically sets up:

**Directory Structure:**

```
.claude/
├── agents/              # Specialized agent configs
│   ├── user-proxy.md
│   ├── project_manager.md
│   ├── backend-developer.md
│   ├── frontend-developer.md
│   └── ...
└── commands/
    └── init-dev-team.md  # The command itself

communication/
├── tasks/               # Task assignments (empty initially)
├── messages/            # Inter-agent messages
├── status/              # Agent availability (initialized)
├── domains/             # Your project domains
│   ├── Content/
│   ├── Users/
│   └── Billing/
├── _templates/          # File templates
└── _index/              # Quick summaries

scripts/                 # Helper bash scripts
├── create-task.sh
├── create-message.sh
├── update-task-status.sh
└── check-agent-status.sh

CLAUDE.md               # Framework instructions
```

*Agent Configurations* customized for your tech stack with all placeholders filled:
- Your frameworks and versions
- Your testing commands
- Your build commands
- Your architecture patterns
- Your domain ownership

*Everything is ready to go* — no manual configuration needed.

**Your First Coordinated Request**

In Claude Code, you can either:

**Option 1: Direct request** (auto-invokes user-proxy):
```
You: “I need to implement user authentication”
```

**Option 2: Use the `/delegate` command:**
```
You: /delegate Add article management with categories and tags
```

Both approaches:
1. Invoke the user-proxy agent
2. User-proxy clarifies requirements if needed
3. User-proxy directs the project manager
4. Project manager creates tasks and invokes specialists
5. Specialists work in parallel (backend, database, frontend, QA)
6. Project manager monitors and reports completion

**Check progress in real-time:**
- `communication/tasks/` — see what each agent is working on
- `communication/status/` — monitor agent availability and current focus
- `communication/messages/` — view inter-agent coordination

All visible in your editor, all version controlled with git.

## Conclusion

By leveraging Claude Code’s agent system (`.claude/agents/`) and Task tool, I built a framework where:

- **Specialization beats generalization** — each agent masters its domain
- **Files replace memory** — no context lost between sessions
- **Git tracks everything** — full audit trail of decisions and implementations
- **Humans maintain control** — readable markdown files you can edit anytime
- **Agents work in parallell**— coordinated through structured communication

Instead of trying to maintain complex state in conversation history, I:
- Persist everything to files
- Use filename patterns for fast lookups
- Coordinate through markdown documents
- Leverage git for versioning

The result? A development workflow that feels like managing a real software team, except:
- The team never forgets context (it's in files)
- The team works 24/7 (agents are always available)
- The team scales instantly (spawn new agents as needed)
- The team documents everything (every decision is recorded)

### What I Learned Building This

1. **Claude Code's agent system is powerful** — the Task tool makes multi-agent coordination natural
2. **Files > Memory** — persisting to filesystem is more reliable than context windows
3. **Markdown is ideal**— human-readable, version-controllable, universally supported
4. **Filename metadata is underrated** — glob patterns are faster than reading every file
5. **Templates prevent chaos** — structured communication is critical for coordination
6. **Agent specialization works**— focused expertise beats jack-of-all-trades

### The Future of Multi-Agent AI Coding

This framework is just the beginning. Imagine where multi-agent AI development could go:

- **Auto-scaling agents** — spawn new specialists based on workload
- **Cross-project learning** — agents reference patterns from past projects
- **Human-in-the-loop workflows** — agents request code review before committing
- **Integration with CI/CD** — agents monitor test results and iterate
- **Metric-driven optimization** — track which agents excel at which tasks

Claude Code's agent platform makes all of this possible.

. . .

## Resources

- **Claude Code:** claude.ai/code

- **Claude Code Docs:** docs.claude.com
- **GitHub Repository:** AI Development Team Framework **(coming soon)**

## Try It Yourself

**Start small:**
1. Create a `.claude/agents/backend_developer.md` file
2. Add a `communication/` directory for coordination
3. Ask Claude Code to invoke your specialist: *"Use the backend_developer agent to implement authentication"*

**Go bigger:**
1. Clone this framework
2. Configure agents for your tech stack
3. Let your AI team build your next feature

**Share your experience:**
- What agents did you create?
- What coordination patterns emerged?
- How did file-based communication work for your project?

*I'm learning in public. Your feedback shapes the future of AI development teams.*

. . .

> *The future of software development isn't human vs. AI. It's human + AI teams, working together through structured coordination.*

AI · Ai Agent · Generative Ai Tools · Claude

Follow

**Written by Mohammed H. Jabreel**

27 followers · 7 following

## Responses (5)

Bgerby

What are your thoughts?

Reza Rezvani
Oct 16

Great reading. Thank you for this great article

1 Reply

Michael Künzler
1 day ago

great article. i would love to test it. are you planning to deploy it soon?

Reply

Jeremy Greven
3 days ago

Great article. Thanks for sharing. I look forward to testing it out when you deploy in git

Reply

See all responses

## Recommended from Medium

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

In AI Software Engineer by Joe Njenga

**Why Claude Weekly Limits Are Making Everyone Angry (And $100/Month Plan Will Not Save You)**

Yesterday, I finally hit my weekly Claude limit, and I wasn't surprised, since I see dozens of other users online going crazy over these...

5d ago 119 20

In nginity by Reza Rezvani

**Master Claude Code “Skills“ Tool to transform repetitive AI prompts into permanent, executable...**

Discover how the Anthropic's new tool for Claude Code called "Skills" transform AI Coding assistant from a generic assistant into your...

Oct 17 118 2

In The Context Layer by Jannis

## How Developers Can Auto-Create Claude Skills from Any Framework's Docs

Turn open-source documentation into Claude-ready Skills in minutes, just your terminal and a GitHub repo.

5d ago 121

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 Coding Nexus by Civil Learning

## MarkItDown: Convert Anything into Markdown — the Smart Way to Feed LLMs

You know that feeling when you're trying to feed a PDF or a Word document into an LLM, and it just doesn't understand what's going on...

Oct 15 209 4

See more recommendations