BUILDING A DOCUMENTATION GENERATOR — Your Agent's first useful job

10 min read · Oct 3, 2025

Reza Rezvani Following

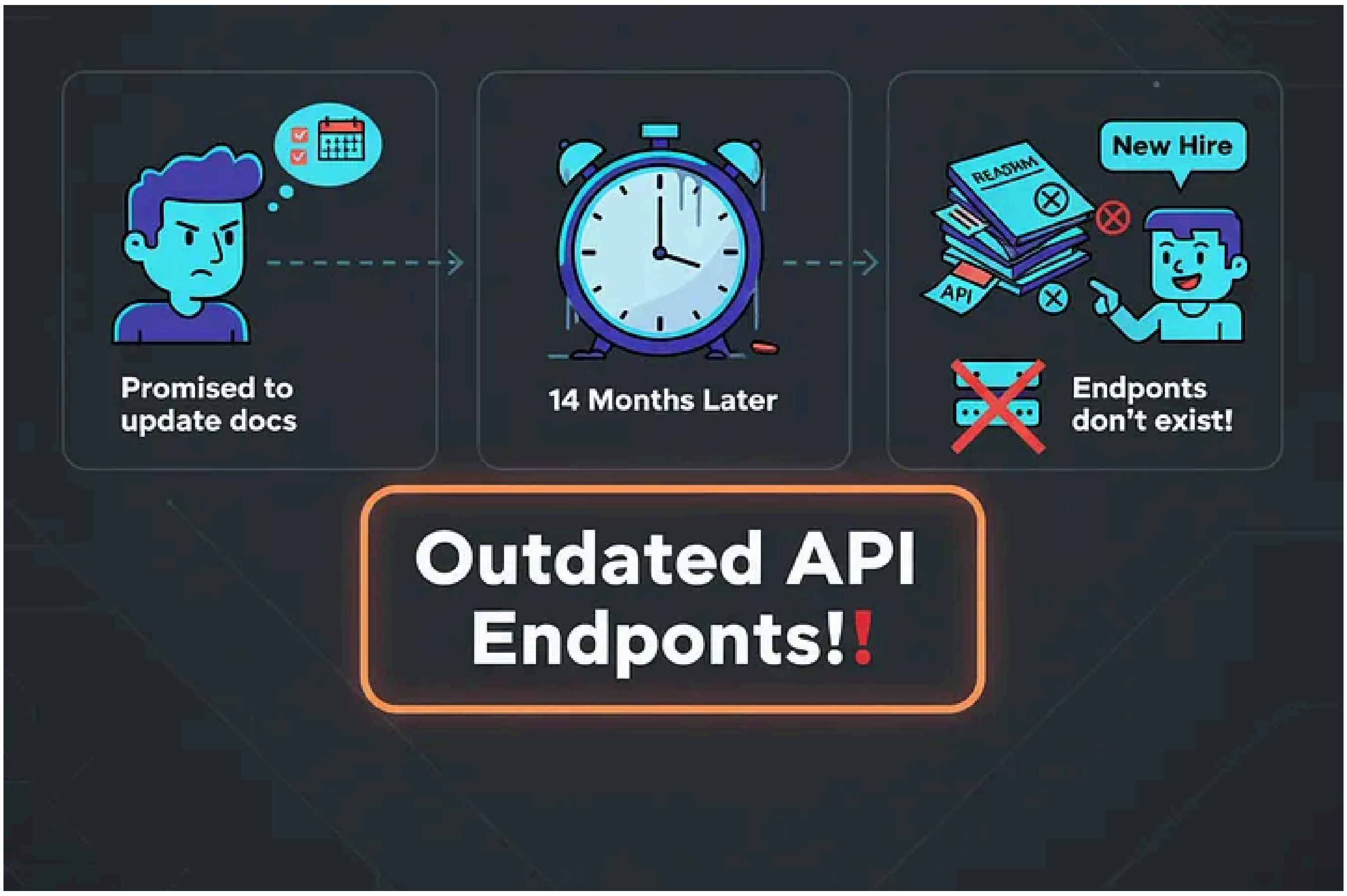
Article 2 of 7 — Foundation Series: I've promised myself I'd keep our docs updated at least a dozen times, and fourteen months later, a new hire pointed out that half our API endpoints in the README don't even exist anymore.

How to start with the Claude Agent SDK? *Read Article 1 of this series!*

. . .

The Documentation Problem

Six months ago, I onboarded a new engineer to our team. On day one, they asked for the API documentation. I pointed them to our README. They opened it, scrolled through outdated endpoints, deprecated functions, and examples that didn't match our current codebase.



Build an Documentation Agent with Claude Agent SDK

"When was this last updated?" they asked.

> *I checked the git history. Eleven months ago.*

Documentation is every developer's least favorite task. We know it's important. We promise ourselves we'll keep it updated. But when you're shipping features under a deadline, documentation always loses.

What if your agent could read your code and write the docs for you?

That's what we're building today: a documentation generator that traverses your codebase, analyzes the structure, and creates a comprehensive README automatically. It's the first agent you'll actually use.

. . .

What We're Building

By the end of this article, you'll have an agent that:

- Scans a project directory for source files

- Reads and analyzes code structure
- Identifies key components (functions, exports, modules)
- Generates formatted markdown documentation
- Writes the README to disk

Unlike *Article 1's simple file reader*, this agent orchestrates multiple tools to complete a complex task. You'll learn how to design tool workflows and guide the agent through multi-step processes.

. . .

Agent Architecture: Planning Your Tools

Before writing code, let's design the system. What tools does the agent need?

Tool 1: list_files
Traverse directories and return file paths. The agent needs to know what files exist before reading them.

Tool 2: read_file
Read file contents. We built this in Article 1; now we'll use it as part of a larger workflow.

Tool 3: write_file
Write generated documentation to disk. The agent needs to save its output.

Here's the workflow:

```
User: "Generate documentation for the src directory"
     ↓
Agent: Calls list_files("src") → Gets file list
     ↓
Agent: Calls read_file() for each relevant file
     ↓
Agent: Analyzes code structure
     ↓
Agent: Generates markdown content
     ↓
Agent: Calls write_file("README.md", content)
```

```
     ↓
Done: README.md created
```

The agent orchestrates this entire flow. You just ask for documentation.

> **Understanding Tool Design**
> Tools are just functions with descriptions. The better you describe what a tool does and when to use it, the better the agent performs. Think of tool descriptions as instructions for a colleague — be specific and clear.

. . .

Building the Tools

Let's implement each tool. Open `src/index.ts`:

Tool 1: List Files

```
import Anthropic from '@anthropic-ai/sdk';
import * as fs from 'fs';
import * as path from 'path';
import * as dotenv from 'dotenv';

dotenv.config();
const client = new Anthropic({
  apiKey: process.env.ANTHROPIC_API_KEY,
});
const tools: Anthropic.Tool[] = [
  {
    name: 'list_files',
    description: 'List all files in a directory. Returns file paths relative to
    input_schema: {
      type: 'object',
      properties: {
        directory: {
          type: 'string',
          description: 'The directory path to list files from',
        },
      },
      required: ['directory'],
    },
  },
  {
    name: 'read_file',
    description: 'Read the contents of a file from the filesystem. Use this aft
    input_schema: {
      type: 'object',
      properties: {
        file_path: {
          type: 'string',
          description: 'The path to the file to read',
        },
      },
      required: ['file_path'],
    },
  },
  {
    name: 'write_file',
    description: 'Write content to a file. Use this to save generated documenta
    input_schema: {
      type: 'object',
      properties: {
        file_path: {
          type: 'string',
          description: 'The path where the file should be written',
        },
        content: {
          type: 'string',
          description: 'The content to write to the file',
        },
      },
      required: ['file_path', 'content'],
    },
  },
];
```

Tool Implementation

```
function executeTool(toolName: string, toolInput: any): string {
  if (toolName === 'list_files') {
    try {
      const dirPath = path.resolve(toolInput.directory);
      const files = fs.readdirSync(dirPath, { recursive: true });

      // Filter for relevant files (TypeScript, JavaScript)
      const relevantFiles = files
        .filter((file: any) => {
          const ext = path.extname(file.toString());
          return ['.ts', '.js', '.tsx', '.jsx'].includes(ext);
        })
        .map((file: any) => path.join(toolInput.directory, file.toString()));

      return JSON.stringify(relevantFiles, null, 2);
    } catch (error: any) {
      return `Error listing files: ${error.message}`;
    }
  }

if (toolName === 'read_file') {
    try {
      const filePath = path.resolve(toolInput.file_path);
      const content = fs.readFileSync(filePath, 'utf-8');
      return content;
    } catch (error: any) {
      return `Error reading file: ${error.message}`;
    }
  }
  if (toolName === 'write_file') {
    try {
      const filePath = path.resolve(toolInput.file_path);
      fs.writeFileSync(filePath, toolInput.content, 'utf-8');
      return `Successfully wrote to ${toolInput.file_path}`;
    } catch (error: any) {
      return `Error writing file: ${error.message}`;
    }
  }
  return 'Unknown tool';
}
```

The Agent Loop

```
  async function runAgent(userMessage: string) {
    console.log(`\n${'='.repeat(60)}`);
    console.log(`User: ${userMessage}`);
    console.log('='.repeat(60));

  const messages: Anthropic.MessageParam[] = [
      { role: 'user', content: userMessage },
    ];
    let response = await client.messages.create({
      model: 'claude-sonnet-4-20250514',
      max_tokens: 8192,
      tools: tools,
      messages: messages,
    });
    let iterations = 0;
    const maxIterations = 20;
    while (response.stop_reason === 'tool_use' && iterations < maxIterations) {
      iterations++;

      const toolUseBlock = response.content.find(
        (block): block is Anthropic.ToolUseBlock => block.type === 'tool_use'
      );
      if (!toolUseBlock) break;
      console.log(`\n🔧 Agent using tool: ${toolUseBlock.name}`);
      console.log(`   Input: ${JSON.stringify(toolUseBlock.input, null, 2)}`);
      const toolResult = executeTool(toolUseBlock.name, toolUseBlock.input);
      messages.push({ role: 'assistant', content: response.content });
      messages.push({
        role: 'user',
        content: [
          {
            type: 'tool_result',
            tool_use_id: toolUseBlock.id,
            content: toolResult,
          },
        ],
      });
      response = await client.messages.create({
        model: 'claude-sonnet-4-20250514',
        max_tokens: 8192,
        tools: tools,
        messages: messages,
      });
    }
    const textBlock = response.content.find(
      (block): block is Anthropic.TextBlock => block.type === 'text'
    );
    console.log(`\n✅ Agent: ${textBlock?.text}\n`);
    console.log(`Total tool calls: ${iterations}`);
  }
  // Test it
  runAgent('Generate documentation for the src directory. Read the TypeScript fil
```

Run Your Documentation Generator

```
npm start
```

You'll see the agent:

1. List files in `src/`
2. Read each TypeScript file
3. Analyze the code structure

4. Generate markdown documentation
5. Write `README.md`

Sample output:

```
============================================================
User: Generate documentation for the src directory...
============================================================
```

```
🔧 Agent using tool: list_files
   Input: {
     "directory": "src"
   }

🔧 Agent using tool: read_file
   Input: {
     "file_path": "src/index.ts"
   }

🔧 Agent using tool: write_file
   Input: {
     "file_path": "README.md",
     "content": "# Documentation Generator Agent..."
   }

✅ Agent: I've created comprehensive documentation in README.md. The
document includes an overview of the project, descriptions of the
three main tools (list_files, read_file, write_file), and usage
examples showing how to run the agent.

Total tool calls: 3
```

Open `README.md` and you'll see the generated documentation.

. . .

Understanding the Agent's Decisions

Let's break down what happened:

The agent didn't just execute tools sequentially. It made decisions:

- Which files to read (only TypeScript files, not configs)

- What information to extract (functions, exports, patterns)
- How to structure the documentation (sections, examples)
- When to stop (after writing the file)

You gave it a goal. It figured out the steps.

The iteration limit is important. We set `maxIterations = 20` to prevent infinite loops. In production, you'd want more sophisticated controls, but for now, this keeps the agent focused.

Tool descriptions matter. Notice how we tell the agent *when* to use each tool: *"Use this after listing files"* or "Use this to save generated documentation." These hints guide the workflow.

> **Common Pattern: Tool Chaining**
> This agent demonstrates tool chaining — using the output of one tool as input to another. The agent lists files, then reads each one. This pattern appears in almost every complex agent.

. . .

Full Python Implementation

Here's the complete Python version:

```python
import anthropic
import os
import json
from pathlib import Path
from dotenv import load_dotenv

load_dotenv()
client = anthropic.Anthropic(
    api_key=os.environ.get("ANTHROPIC_API_KEY")
)
tools = [
    {
        "name": "list_files",
        "description": "List all files in a directory. Returns file paths relat
        "input_schema": {
            "type": "object",
            "properties": {
                "directory": {
                    "type": "string",
                    "description": "The directory path to list files from",
                }
            },
            "required": ["directory"],
        },
    },
    {
        "name": "read_file",
        "description": "Read the contents of a file from the filesystem. Use th
        "input_schema": {
            "type": "object",
            "properties": {
                "file_path": {
                    "type": "string",
                    "description": "The path to the file to read",
                }
            },
            "required": ["file_path"],
        },
    },
    {
        "name": "write_file",
        "description": "Write content to a file. Use this to save generated doc
        "input_schema": {
            "type": "object",
            "properties": {
                "file_path": {
                    "type": "string",
                    "description": "The path where the file should be written",
                },
                "content": {
                    "type": "string",
                    "description": "The content to write to the file",
                }
            },
            "required": ["file_path", "content"],
        },
    },
]
def execute_tool(tool_name: str, tool_input: dict) -> str:
    if tool_name == "list_files":
        try:
            dir_path = Path(tool_input["directory"]).resolve()
            files = []

            for file_path in dir_path.rglob("*"):
                if file_path.is_file() and file_path.suffix in ['.py', '.js', '
                    relative_path = file_path.relative_to(Path.cwd())
                    files.append(str(relative_path))
            return json.dumps(files, indent=2)
        except Exception as e:
            return f"Error listing files: {str(e)}"

    if tool_name == "read_file":
        try:
            file_path = Path(tool_input["file_path"]).resolve()
            with open(file_path, 'r', encoding='utf-8') as f:
                return f.read()
        except Exception as e:
            return f"Error reading file: {str(e)}"

    if tool_name == "write_file":
        try:
            file_path = Path(tool_input["file_path"]).resolve()
            with open(file_path, 'w', encoding='utf-8') as f:
                f.write(tool_input["content"])
            return f"Successfully wrote to {tool_input['file_path']}"
        except Exception as e:
            return f"Error writing file: {str(e)}"

    return "Unknown tool"
def run_agent(user_message: str):
    print(f"\n{'=' * 60}")
    print(f"User: {user_message}")
    print('=' * 60)
    messages = [{"role": "user", "content": user_message}]
    response = client.messages.create(
        model="claude-sonnet-4-20250514",
        max_tokens=8192,
        tools=tools,
        messages=messages,
    )
    iterations = 0
    max_iterations = 20
    while response.stop_reason == "tool_use" and iterations < max_iterations:
        iterations += 1

        tool_use = next((block for block in response.content if block.type == "

        if not tool_use:
            break
        print(f"\n🔧 Agent using tool: {tool_use.name}")
        print(f"   Input: {tool_use.input}")
        tool_result = execute_tool(tool_use.name, tool_use.input)
        messages.append({"role": "assistant", "content": response.content})
        messages.append({
            "role": "user",
            "content": [
                {
                    "type": "tool_result",
                    "tool_use_id": tool_use.id,
                    "content": tool_result,
                }
            ],
        })
        response = client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=8192,
            tools=tools,
            messages=messages,
        )
    final_text = next((block.text for block in response.content if hasattr(bloc
    print(f"\n✅ Agent: {final_text}\n")
    print(f"Total tool calls: {iterations}")
if __name__ == "__main__":
    run_agent("Generate documentation for the src directory. Read the Python fi
```

. . .

Improving the Documentation

The basic generator works, but you can make it smarter. Here are enhancements to try:

1. Better Prompting

Guide the agent with more specific instructions:

```
const prompt = `Generate documentation for the src directory following this str
```

```
# Project Name
Brief overview

## Installation
Setup instructions

## Architecture
Main components and their roles

## API Reference
Key functions and their signatures

## Usage Examples
Code examples showing common patterns

Read all TypeScript files, analyze them, and create README.md with
these sections.`;
```

```
runAgent(prompt);
```

2. Filter Non-Code Files

Improve `list_files` to skip tests and config:

```
const relevantFiles = files
  .filter((file: any) => {
    const fileName = file.toString();
    const ext = path.extname(fileName);

    // Include only source files
    if (!['.ts', '.js', '.tsx', '.jsx'].includes(ext)) return false;

    // Skip test files
    if (fileName.includes('.test.') || fileName.includes('.spec.')) return fals

    // Skip config files
    if (fileName.includes('config')) return false;

    return true;
  });
```

3. Add Error Handling

Handle edge cases in tool execution:

```
if (toolName === 'write_file') {
  try {
    const filePath = path.resolve(toolInput.file_path);
    const dir = path.dirname(filePath);

    // Create directory if it doesn't exist
    if (!fs.existsSync(dir)) {
      fs.mkdirSync(dir, { recursive: true });
    }

    fs.writeFileSync(filePath, toolInput.content, 'utf-8');
    return `Successfully wrote ${toolInput.content.length} characters to ${tool
  } catch (error: any) {
    return `Error writing file: ${error.message}`;
  }
}
```

. . .

What We Learned

You built a multi-tool agent that orchestrates a complex workflow. The key lessons:

1. Tool Design is Critical
Good descriptions guide the agent. Bad descriptions confuse it. Spend time crafting clear, specific tool descriptions.

2. Agents Make Decisions
The agent chose which files to read and how to structure the output. You provided the goal, and it figured out the execution.

3. Iteration Limits Prevent Runaway
Always set maximum iterations for production agents. Infinite loops waste tokens and money.

4. Tool Chaining is Powerful
Using outputs from one tool as inputs to another enables complex workflows without hard-coding the sequence.

5. Prompting Shapes Output
Specific instructions in your prompt lead to better results. "Generate documentation" is vague. "Generate documentation with these five sections" is actionable.

. . .

What's Next

This documentation generator is useful, but it has a limitation: it forgets everything between runs. If you ask it to update the docs, it starts from scratch.

In Article 3, we'll add **memory and checkpoints**. Your agent will:

- Remember which files it already processed
- Track changes since the last run
- Only update what's necessary

- Rollback if something goes wrong

You'll also learn how to persist memory across sessions using Supabase, enabling agents that work on long-running tasks over days or weeks.

Before Article 3, try these experiments:

1. **Customize the output format** — Modify the prompt to generate different styles (API docs, user guides, technical specs)
2. **Add a code analysis tool** — Create a tool that extracts function signatures and exports
3. **Generate multiple documents** — Have the agent create separate files for different sections
4. **Handle larger codebases** — Test on a project with 20+ files and observe how the agent manages complexity

All code from this article is in the *GitHub repo*. Includes:

- Complete TypeScript and Python implementations
- Example projects to document
- Exercises with solutions
- Troubleshooting guides

See you in Article 3, where your agents learn to remember.

. . .

*Part of **The Agent Builder's Playbook** series. Read the full series | Follow for updates*

. . .

👉 Bookmark this post, share it with your team, and subscribe if you want to master *AI-driven Development with Claude Code.*

Or check out my new Master Guide for Spec-Driven Development:

👉 **Step 1:** Read this Master Guide — your evergreen hub for Spec-Driven Development resources.
👉 **Step 2:** Read Part 1: The Foundation to set up your memory system, constitution, and specification workflow.
👉 **Step 3:** Continue with Part 2: Execution and Scaling to turn specs into plans, tasks, and tested features.

About the Author

Alireza Rezvani is a Chief Technology Officer, Senior Full-stack architect & software engineer, and AI technology specialist with expertise in modern development frameworks, cloud native applications, and agentic AI systems. With a focus on *ReactJS*, *NextJS*, *Node.js*, and cutting-edge AI technologies and concepts of AI engineering, Alireza helps engineering teams leverage tools like *Gemini CLI*, and *Claude Code* or *Codex from OpenAI* to transform their development workflows.

Connect with Alireza at *alirezarezvani.com* for more insights on AI-powered development, architectural patterns, and the future of software engineering.

Looking forward to connecting and seeing your contributions — check out my *open source projects on GitHub*!

✨ Thanks for reading! If you'd like more practical insights on AI and tech, hit **subscribe** to stay updated.

I'd also love to hear your thoughts — drop a comment with your ideas, questions, or even the kind of topics you'd enjoy seeing here next. Your input really helps shape the direction of this channel.

Ai Agent · Claude Code · Software Development · Api Documentation Tool · Agentic Coding

Following

Written by Reza Rezvani

938 followers · 71 following

As CTO of a Berlin AI MedTech startup, I tackle daily challenges in healthcare tech. With 2 decades in tech, I drive innovations in human motion analysis.

No responses yet

Bgerby

What are your thoughts?

More from Reza Rezvani

In nginity by Reza Rezvani

I Let Claude Sonnet 4.5

IMAGINE this: It's 6 a.m., the kind of quiet dawn where the world's still wrapped in that soft, hazy light filtering through your blinds...

Sep 29

Reza Rezvani

"7 Steps" How to Stop Claude Code from Building the Wrong Thing (Part 1): The Foundation of...

Learn how to stop Claude Code from rewriting your architecture with vague prompts. This guide introduces Spec-Driven Development...

Sep 17

In nginity by Reza Rezvani

Claude AI and Claude Code Skills: Teaching AI to Think Like Your Best Engineer

Oct 19

Reza Rezvani

I Discovered Claude Code's Secret: You Don't Have to Build Alone

I've been coding long enough to know that the late-night debugging sessions aren't glamorous. They're just necessary.

Sep 19

See all from Reza Rezvani

Recommended from Medium

In Realworld AI Use Cases by Chris Dunlop

The complete guide to Claude Code's newest feature "skills"

Claude Code released a new feature called Skills and spent hours testing them so you don't have to. Here's why they are helpful

Oct 21

Manojkumar Vadivel

The .claude Folder: A 10-Minute Setup That Makes AI Code Smarter

If you're new to Claude Code, it's a powerful AI coding agent that helps you write, refactor, and understand code faster. This article…

Sep 15

•••

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 •••

In Coding Nexus by Code Coup

OpenSpec: A Spec-Driven Workflow for AI Coding Assistants (No API Keys Needed)

I've been using AI coding tools for a while — Claude, Cursor, even Copilot. They're fast, sometimes magical… but they also love to…

✦ Oct 18 •••

Vishal Mysore

Comprehensive Guide to Spec-Driven Development Kiro, GitHub Spec Kit, and BMAD-METHOD

What is Spec-Driven Development (SDD)?

Sep 30

Ayaan haider

Sonnet 4.5 vs Haiku 4.5 vs Opus 4.1—Which Claude Model Actually Works Best in Real Projects

Claude Code 2.0 gave us three models to work with—Haiku 4.5, Sonnet 4.5, and Opus 4.1.

Oct 18

See more recommendations