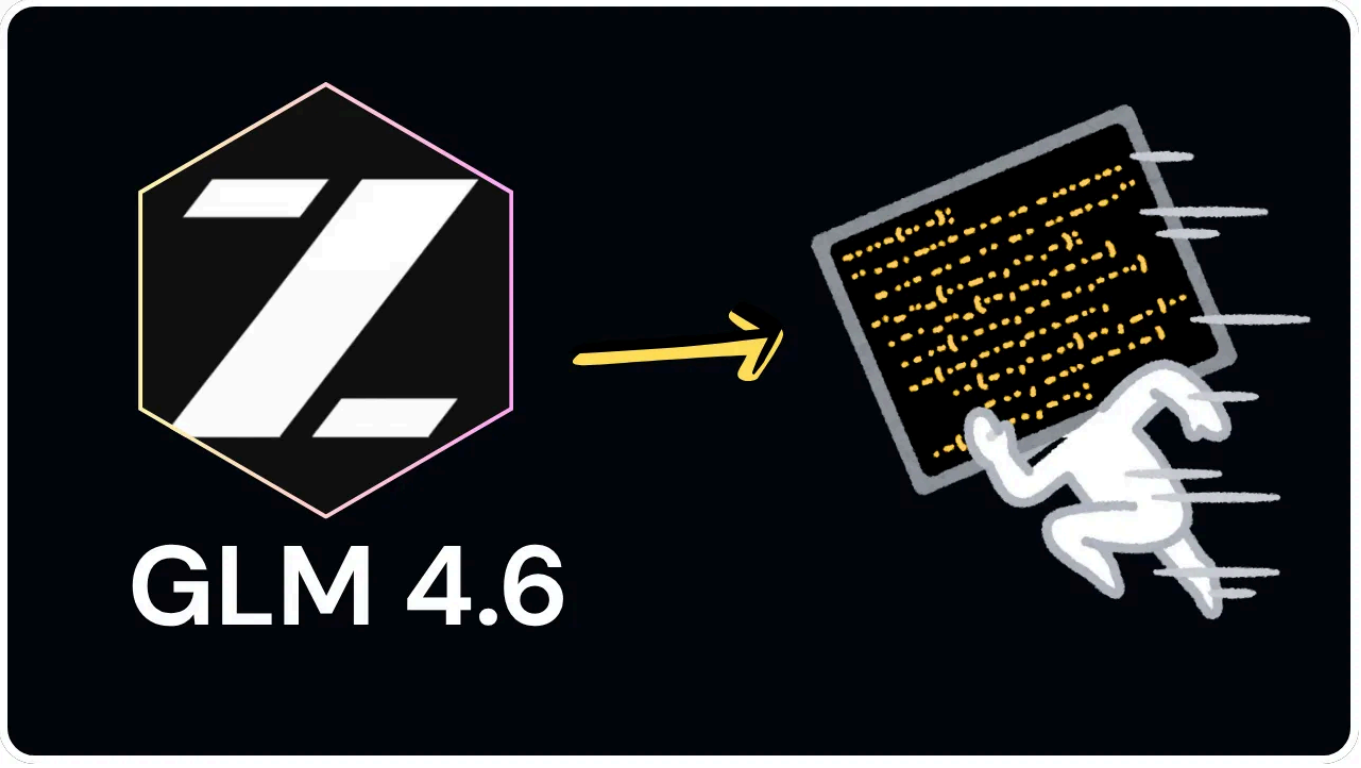
Member-only story

I Tested Newly Released GLM 4.6 (And Discovered a Cheaper Way to Code Like a Beast)

Joe Njenga Follow 18 min read · 3 days ago



GLM 4.6 has been released just a few days after Claude 4.5, and it’s another coding beast, as you will see in my test here.

I previously tested GLM 4.5 on Claude Code, and I found the best combination for coding tasks.

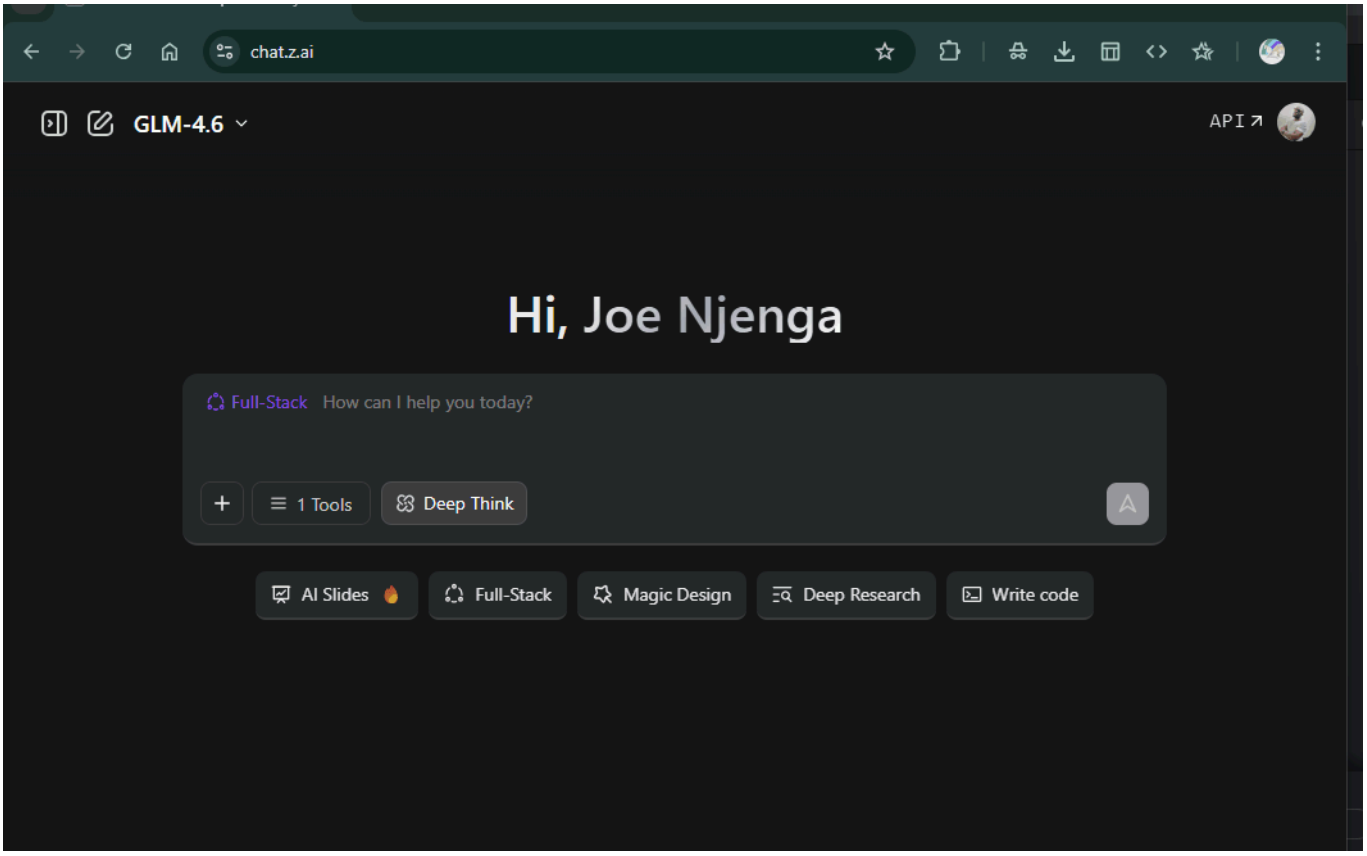
If you've read my article, you know I built a proxy server to use the Chinese models with Claude Code.

Now,

When I saw the release of GLM 4.6, I couldn't pass up the opportunity to test it.

> ***But here’s what got me excited — not just another benchmark comparison, but real agentic capabilities. The kind where you give it a task and it plans, builds, tests, and fixes everything autonomously.***

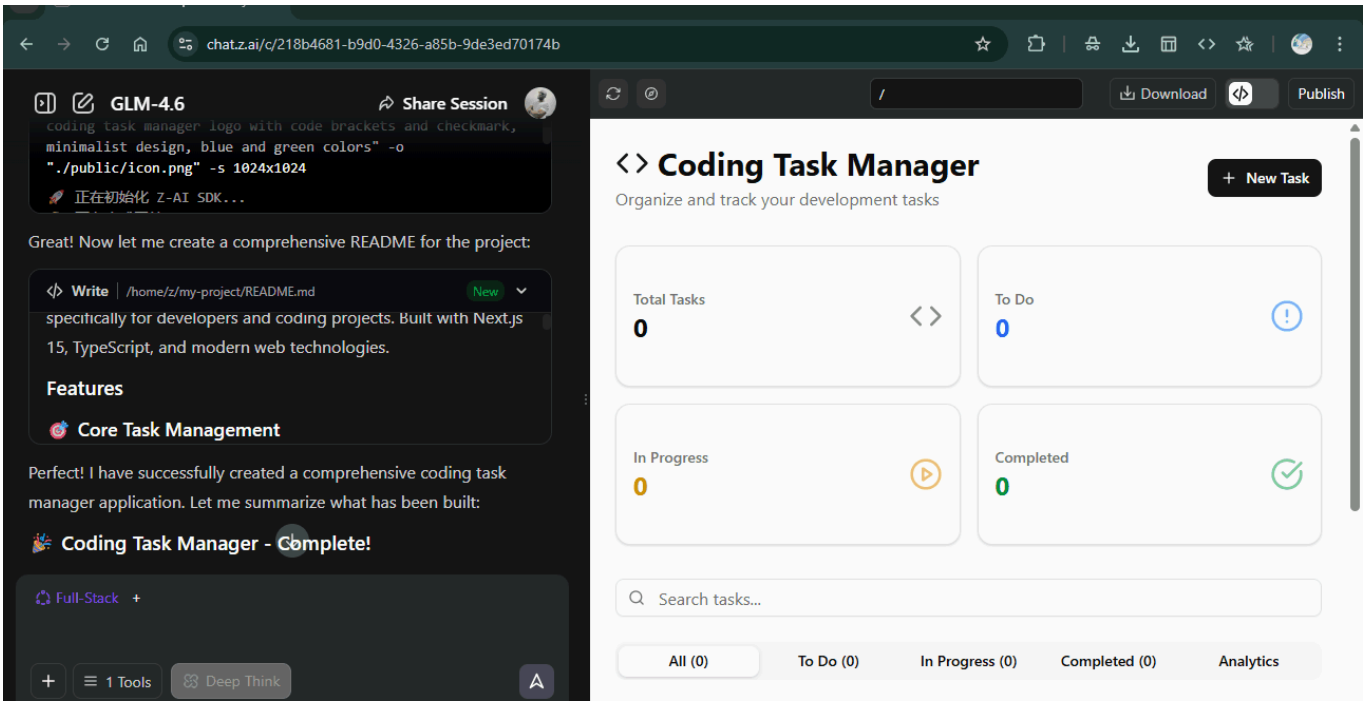
I initially tested GLM 4.6 via the chat interface, and it comes ***with a full-stack feature that scaffolds full-stack projects in minutes***, which impressed me.



From this one-line prompt:

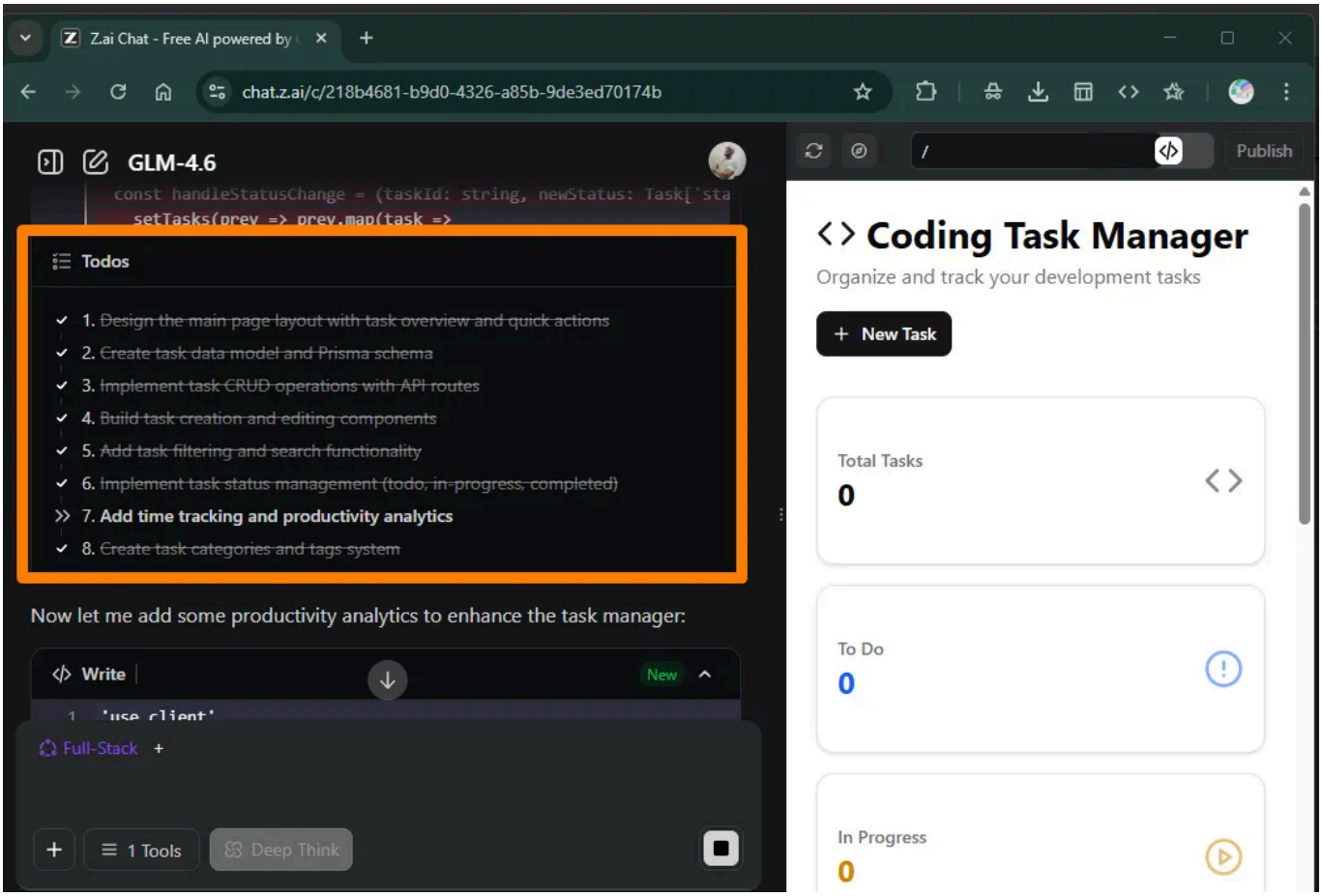
```
Create me a coding task manager app for my day to day coding
```

It quickly built this full-stack project in one shot :



> *One of the notable features I liked in the chat thread is the Todos worklist that the chat agent carefully follows as it builds the project.*

Something close to what we saw in Augment Code, as I shared in this post — *I Tested Augment Code's Agentic Tasklist Workflow*



Its ability to correct mistakes on the fly without drama was also impressive; it's likely way better than any other AI web app I have seen — Bolt, Loveable, etc.

> I will run a build test against ehem in a future post to see which is the best.

> ***But to understand the pure agentic power of GLM 4.6, we need to go beyond the simple prompting.***

In this tutorial, we will be building a complete autonomous coding agent from scratch.

> *We'll watch it generate an entire task manager API with authentication — planning, coding, and testing on its own.*

But first,

Let's get the basic details about what is new in GLM 4.6.

What You Need to Know About GLM 4.6

GLM 4.6 is the latest model from Z.ai (formerly Zhipu AI).

Released on September 30, 2025, it builds on GLM 4.5 with some serious upgrades focused on coding and agentic workflows.

The model isn't trying to beat every benchmark out there, but focused on one thing: autonomous coding at a price point that makes sense for daily use.

Core Improvements Over GLM 4.5

Z.ai made targeted changes that matter for real development work:

Context Window Expansion

- GLM 4.5: 128K tokens
- GLM 4.6: 200K tokens

- What this means: You can work with roughly 150,000 words or about 25,000 lines of code in a single conversation.

Token Efficiency

- Uses 15% fewer tokens than GLM 4.5 for the same tasks
- Completes projects faster with less API usage
- In their tests: 651K tokens per task vs 762K for GLM 4.5

Agentic Capabilities

- Better at planning multi-step implementations
- Improved tool use during reasoning
- More reliable at self-correction when errors occur

Code Generation Quality

- Cleaner front-end code generation
- Better instruction following
- Improved architectural consistency across files

Performance Against Claude 4.5 Sonnet

Z.ai claims they tested it against Claude Sonnet 4 in 74 real-world coding scenarios and published all the results.

Head-to-Head Results:

- Win rate vs Claude Sonnet 4: 48.6%
- Ties: 9.5%
- Losses: 41.9%

That's near parity with Claude Sonnet 4.

But here's the caveat — GLM 4.6 still lags behind Claude Sonnet 4.5 on pure coding benchmarks.

SWE-bench Verified Scores:

- Claude Sonnet 4.5: 70.4%
- GLM 4.6: 64.2%
- GLM 4.5: 58.9%

For most coding tasks that aren't pushing absolute limits, GLM 4.6 delivers comparable results.

Pricing

This is where things get interesting for daily use.

API Pricing Comparison:

Model Input (per 1M tokens) Output (per 1M tokens) GLM 4.6 $0.60 $2.00 Claude 4.5 Sonnet $3.00 $15.00

That's 5x cheaper on input and 7.5x cheaper on output.

Subscription Plans:

Plan Monthly Cost What You Get GLM Coding Plan $3 120 prompts per 5-hour cycle, works with Claude Code/Cline Claude Pro $20 Standard usage limits.

> *For context, if you're running agentic workflows that make 50+ API calls per project, the cost difference adds up fast. A project that costs $0.25 with Claude might cost $0.035 with GLM 4.6.*

Why It's Cheaper

GLM 4.6 employs a Mixture of Experts (MoE) architecture with a total of 355 billion parameters, but only 32 billion are active at any given time.

Smaller active parameters mean:

- Lower compute requirements
- Faster inference
- Cheaper operational costs

Z.ai passes those savings to users. The token efficiency improvements also mean you use fewer tokens to get the same work done.

> *If you're building with AI coding tools daily, GLM 4.6 offers a practical middle ground. You get near-Claude performance at a fraction of the cost. The trade-off is slightly lower accuracy on the hardest coding challenges.*

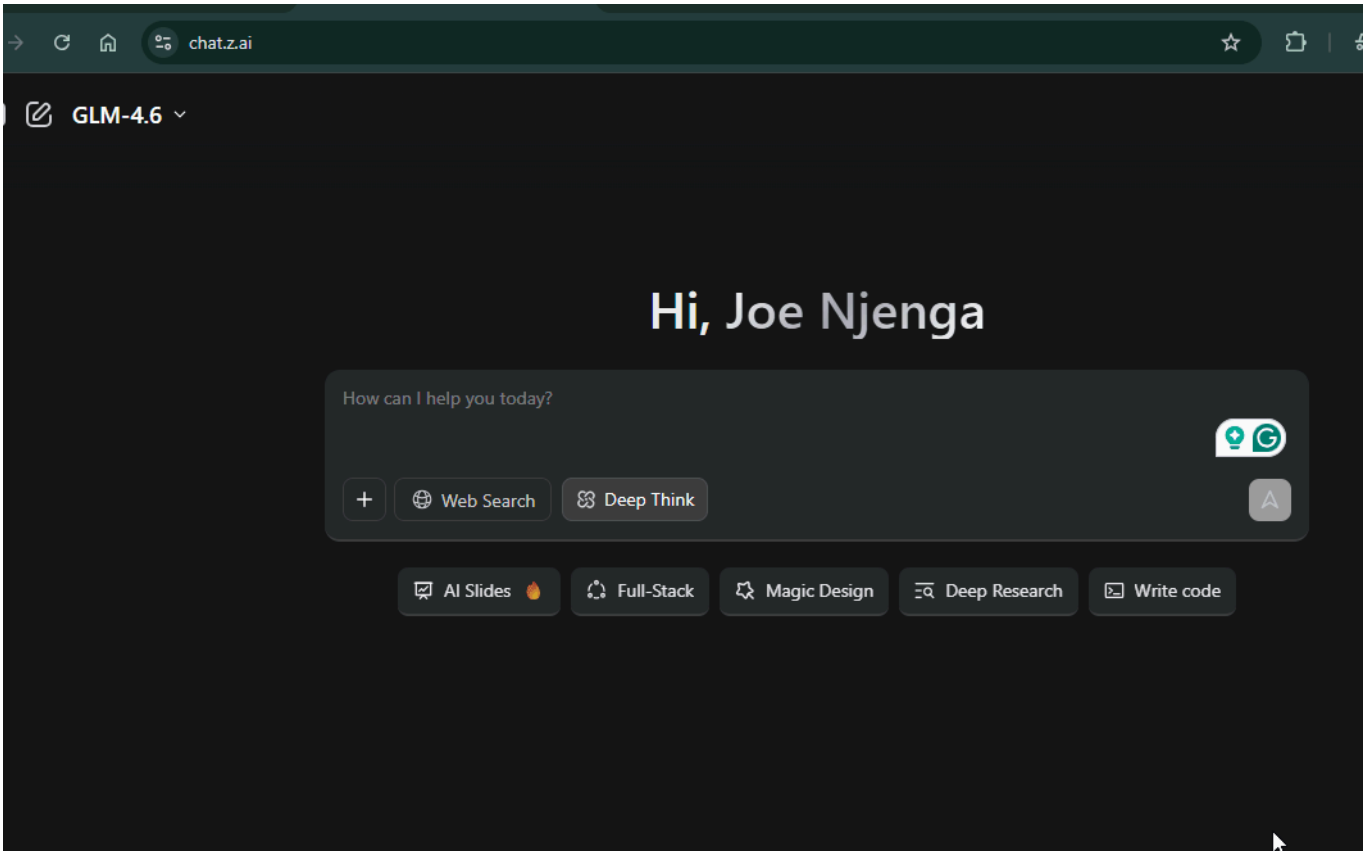
For prototype development, tool building, and most real-world coding tasks, that trade-off makes sense.

Now let's build something with it.

The Project: Building an Autonomous Coding Agent

To test GLM 4.6 properly, I needed more than basic code generation.

I did not want to test the coding directly from the chat but rather using the API.



> *I wanted to see if my AI agent via their API could handle what a junior developer does: take a task, break it down, build it, test it, and fix issues independently.*

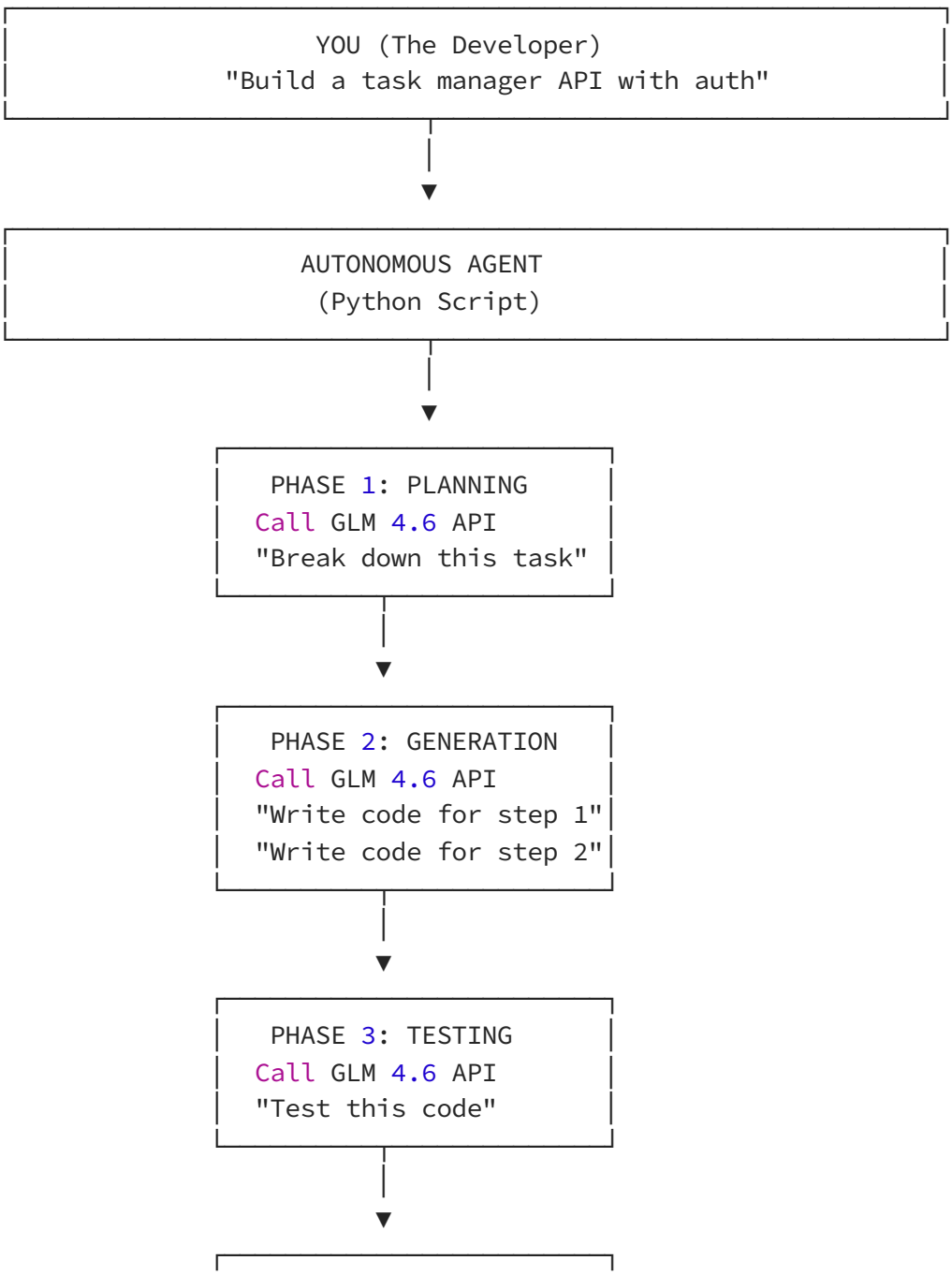
So I built an autonomous coding agent in Python.

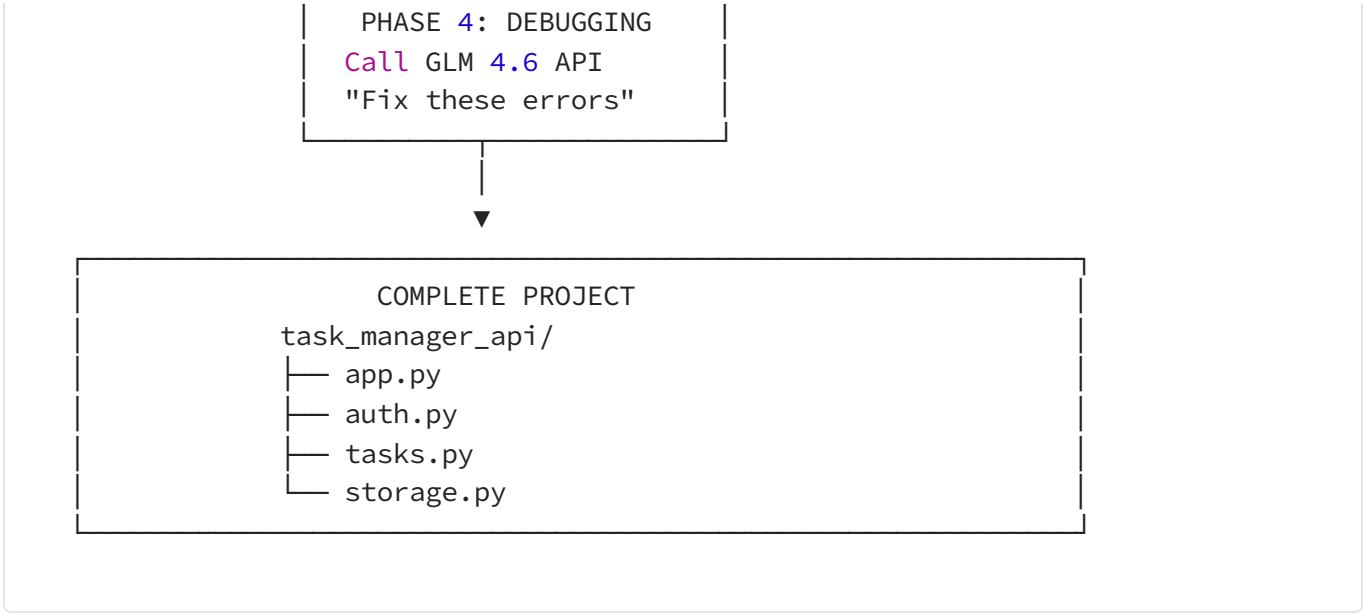
The agent is a Python script that:

- *Takes a feature request in plain English*
- *Calls GLM 4.6 to plan the implementation*
- *Generates all necessary code files*
- *Creates a complete working project*
- *Tests what it built and fixes errors*

> *Think of it as a wrapper around GLM 4.6 that manages the entire development workflow.*

How the Autonomous Agent Works





Why This Approach Tests Real Coding Capabilities

Most GLM 4.6 reviews test single prompts. You ask for code, you get code.

> *That's it, which I found to be equally fast and would rate it the same as Claude 4.5 or better.*

But agentic coding is different.

It requires:

Context Management

- The agent maintains conversation history across all phases
- GLM 4.6 sees what it built in step 1 when working on step 2
- This tests the 200K context window in practice

Multi-Turn Reasoning

- Planning requires architectural thinking
- Each code generation phase builds on previous decisions
- Testing requires analyzing its own output
- Debugging requires understanding what went wrong

Autonomous Decision Making

- The agent doesn't get step-by-step instructions
- GLM 4.6 decides file structure, dependencies, and implementation order
- It catches its own mistakes during testing

This is closer to how you'd use an AI coding assistant in real work. Not one-off code snippets, but complete project development.

Building the GLM Coding Agent

Before we can watch GLM 4.6 work its magic, we need to build the agent itself. I'll walk you through each part so you can follow along and test as we go.

You can find the complete code on my GitHub repository.

But I recommend building it step by step first to understand how it works.

Prerequisites

Make sure you have these installed:

- Python 3.8 or higher
- pip (Python package manager)
- A text editor or VS Code
- GLM 4.6 API key from z.ai

Project Structure

Here's what we're building:

```
autonomous_agent/
├── agent.py              # Main agent logic
├── glm_client.py         # GLM 4.6 API wrapper
├── file_manager.py       # Handles file creation
├── config.py             # Configuration settings
├── requirements.txt      # Dependencies
└── generated_projects/   # Where generated code goes
```

Each file has a specific job. We'll build them one at a time.

Step 1: Create Project and Virtual Environment

First, create your project folder and set up a clean Python environment.

```
# Create project directory
mkdir autonomous_agent
cd autonomous_agent

# Create virtual environment
python -m venv venv
# Activate virtual environment
# On Windows:
venv\Scripts\activate
# On Mac/Linux:
source venv/bin/activate
```

Your Python environment should now be `(venv)` activated.

Step 2: Install Dependencies

Create a `requirements.txt` file:

```
requests==2.31.0
python-dotenv==1.0.0
```

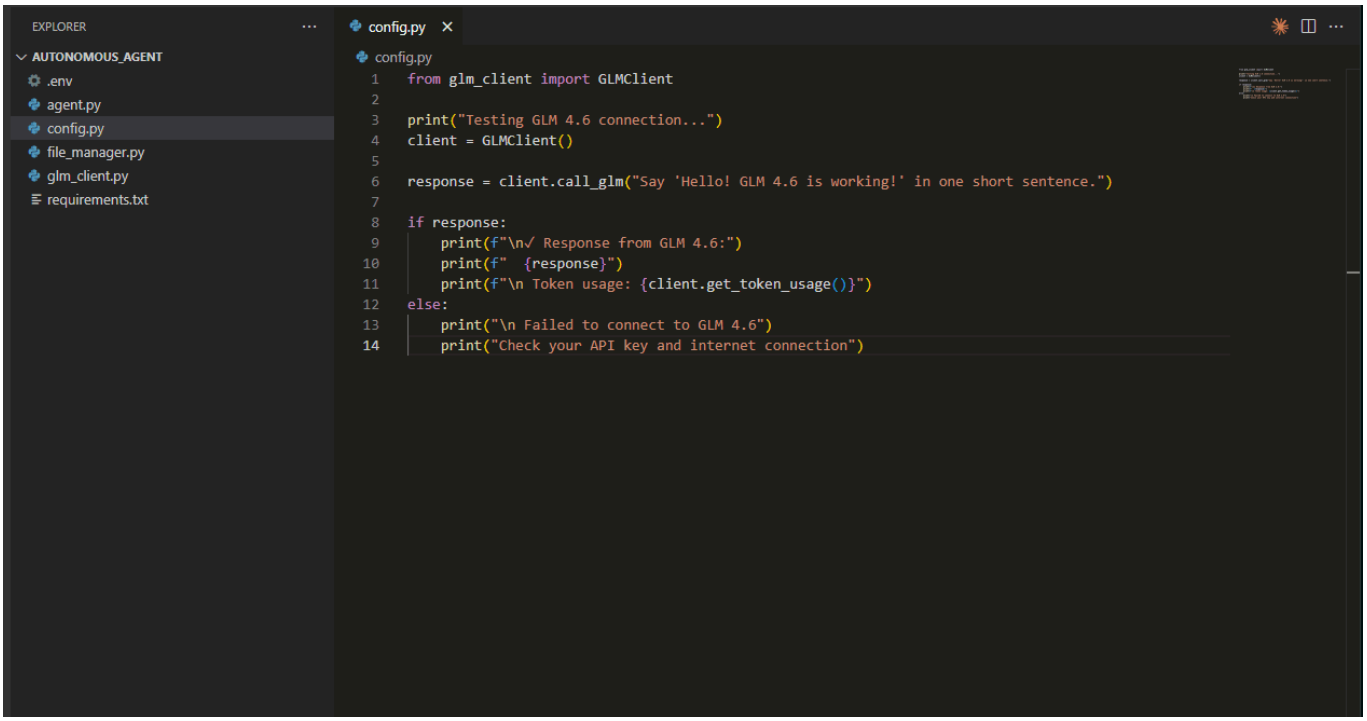
Install the packages:

```
pip install -r requirements.txt
```

What these do:

- `requests` : Makes HTTP calls to GLM 4.6 API
- `python-dotenv` : Manages your API key securely

Step 3: Configuration Setup



Create `config.py` :

```
import os
from dotenv import load_dotenv

# Load environment variables
load_dotenv()
class Config:
    # GLM 4.6 API settings
    API_KEY = os.getenv('GLM_API_KEY')
    API_URL = "https://api.z.ai/api/paas/v4/chat/completions"
    MODEL = "glm-4.6"
```

```
    # Generation settings
    MAX_TOKENS = 4096
    TEMPERATURE = 0.6

    # Project settings
    OUTPUT_DIR = "generated_projects"
```

Create a `.env` file in the same directory:

```
GLM_API_KEY=your-api-key-here
```

Test this step:

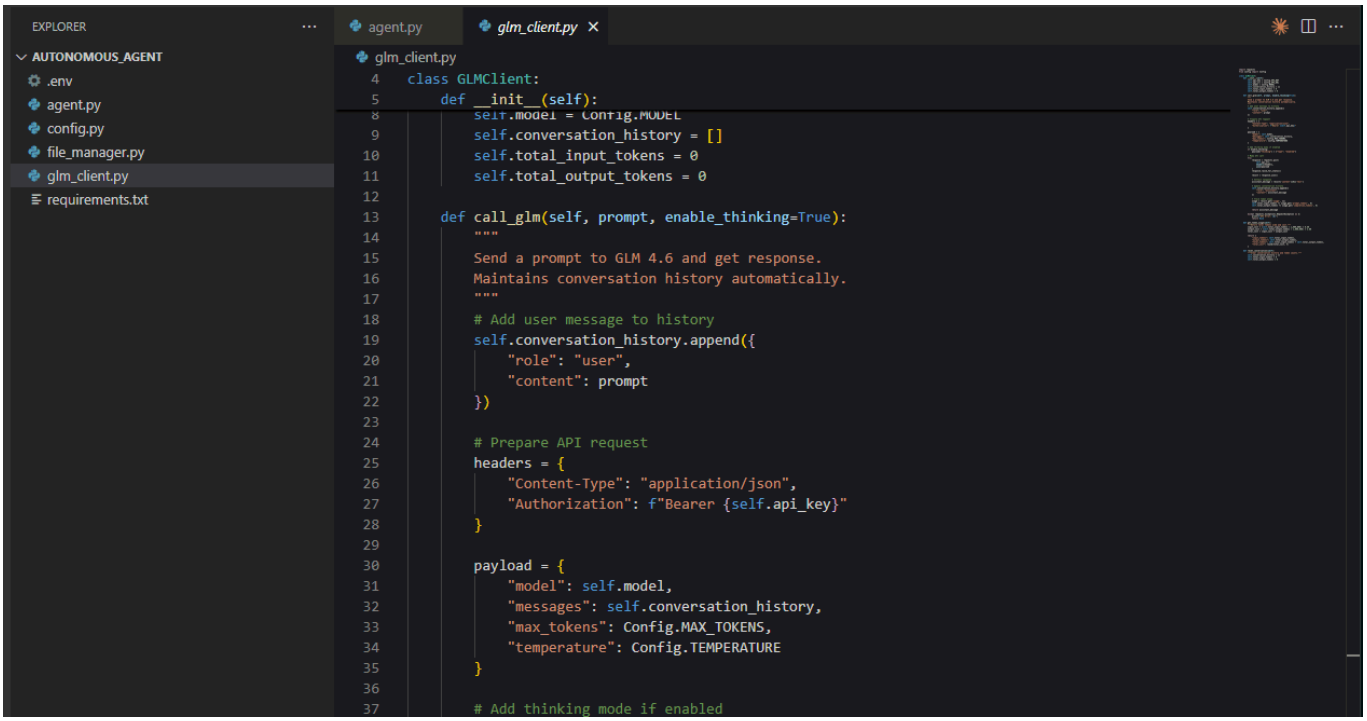
```
# test_config.py
from config import Config

print(f"API Key loaded: {Config.API_KEY[:10]}...")
print(f"API URL: {Config.API_URL}")
```

Run: `python test_config.py`

You should see your API key (first 10 characters) and the URL. If you get errors, check your `.env` file.

Step 4: Building the GLM Client



Create `glm_client.py`. This handles all communication with GLM 4.6.

```
import requests
from config import Config

class GLMClient:
    def __init__(self):
        self.api_key = Config.API_KEY
        self.api_url = Config.API_URL
        self.model = Config.MODEL
        self.conversation_history = []
        self.total_input_tokens = 0
        self.total_output_tokens = 0

    def call_glm(self, prompt, enable_thinking=True):
        """
        Send a prompt to GLM 4.6 and get response.
        Maintains conversation history automatically.
        """
        # Add user message to history
        self.conversation_history.append({
            "role": "user",
            "content": prompt
        })

        # Prepare API request
```

```
        headers = {
            "Content-Type": "application/json",
            "Authorization": f"Bearer {self.api_key}"
        }

        payload = {
            "model": self.model,
            "messages": self.conversation_history,
            "max_tokens": Config.MAX_TOKENS,
            "temperature": Config.TEMPERATURE
        }

        # Add thinking mode if enabled
        if enable_thinking:
            payload["thinking"] = {"type": "enabled"}

        # Make API call
        try:
            response = requests.post(
                self.api_url,
                headers=headers,
                json=payload
            )
            response.raise_for_status()

            result = response.json()

            # Debug: Print response structure if there's an error
            if 'error' in result:
                print(f"API Error Response: {result}")
                return None

            # Try to extract response (handle different formats)
            assistant_message = None

            # Format 1: Standard format with 'content'
            if 'content' in result:
                assistant_message = result['content'][0]['text']
            # Format 2: OpenAI-style format with 'choices'
            elif 'choices' in result:
                assistant_message = result['choices'][0]['message']['content']
            else:
                print(f"Unexpected response format: {result}")
                return None

            # Update conversation history
            self.conversation_history.append({
                "role": "assistant",
                "content": assistant_message
            })
```

```
            # Track token usage
            usage = result.get('usage', {})
            self.total_input_tokens += usage.get('prompt_tokens', 0)
            self.total_output_tokens += usage.get('completion_tokens', 0)

            return assistant_message

        except requests.exceptions.RequestException as e:
            print(f"API Error: {e}")
            return None
        except KeyError as e:
            print(f"Response parsing error: {e}")
            print(f"Response structure: {result}")
            return None

    def get_token_usage(self):
        """Return total tokens used and cost."""
        input_cost = (self.total_input_tokens / 1_000_000) * 0.60
        output_cost = (self.total_output_tokens / 1_000_000) * 2.00
        total_cost = input_cost + output_cost

        return {
            "input_tokens": self.total_input_tokens,
            "output_tokens": self.total_output_tokens,
            "total_tokens": self.total_input_tokens + self.total_output_tokens,
            "total_cost": round(total_cost, 4)
        }

    def reset_conversation(self):
        """Clear conversation history and token counts."""
        self.conversation_history = []
        self.total_input_tokens = 0
        self.total_output_tokens = 0
```

What this does:

- `call_glm()` : Sends prompts to GLM 4.6
- Maintains conversation history automatically
- Tracks token usage for cost calculation

- Handles errors gracefully

Test this step:

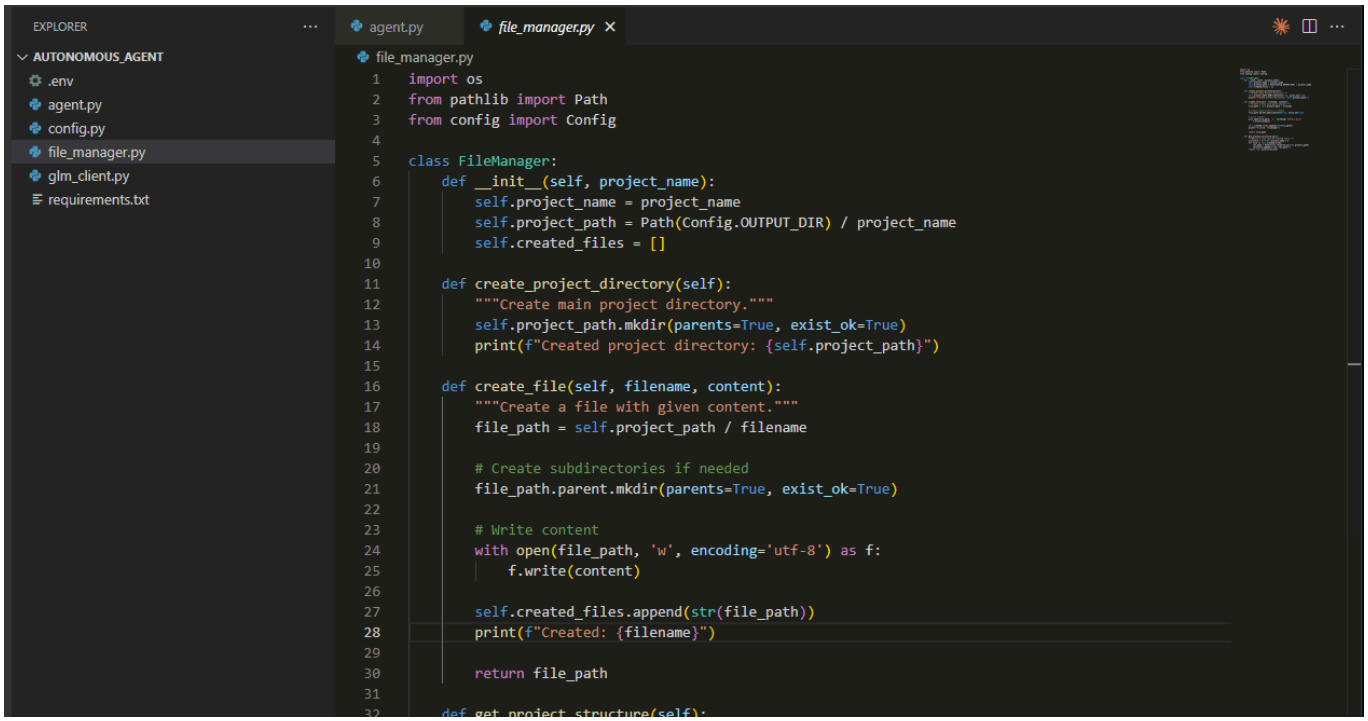
```
# test_client.py
from glm_client import GLMClient

client = GLMClient()
response = client.call_glm("Say hello!")
print(f"Response: {response}")
print(f"Token usage: {client.get_token_usage()}")
```

Run: `python test_client.py`

You should receive a response from GLM 4.6 and be able to view token usage. If this works, your API connection is good.

Step 5: File Manager



Create `file_manager.py`. This handles the creation and organization of generated files.

```
import os
from pathlib import Path
from config import Config

class FileManager:
    def __init__(self, project_name):
        self.project_name = project_name
        self.project_path = Path(Config.OUTPUT_DIR) / project_name
        self.created_files = []

    def create_project_directory(self):
        """Create main project directory."""
        self.project_path.mkdir(parents=True, exist_ok=True)
        print(f"Created project directory: {self.project_path}")

    def create_file(self, filename, content):
        """Create a file with given content."""
        file_path = self.project_path / filename

        # Create subdirectories if needed
        file_path.parent.mkdir(parents=True, exist_ok=True)
```

```
        # Write content
        with open(file_path, 'w', encoding='utf-8') as f:
            f.write(content)

        self.created_files.append(str(file_path))
        print(f"Created: {filename}")

        return file_path

    def get_project_structure(self):
        """Return a tree view of created files."""
        structure = [f"{self.project_name}/"]
        for file in self.created_files:
            rel_path = Path(file).relative_to(self.project_path)
            structure.append(f"├── {rel_path}")
        return "\n".join(structure)
```

What this does:

- Creates project directories
- Writes generated code to files
- Tracks what was created
- Shows project structure

Test this step:

```
# test_file_manager.py
from file_manager import FileManager
fm = FileManager("test_project")
fm.create_project_directory()
fm.create_file("test.py", "print('Hello, World!')")
print(fm.get_project_structure())
```

Run: `python test_file_manager.py`

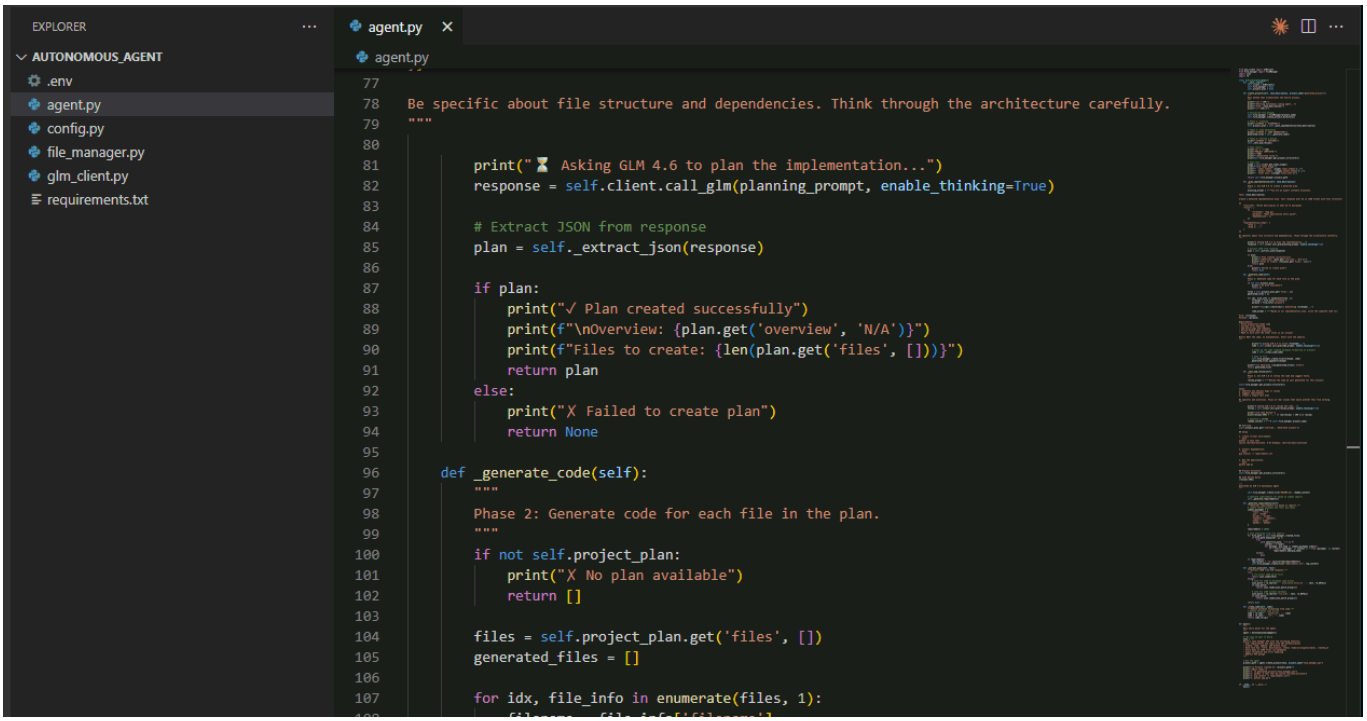
Check that a `generated_projects/test_project/test.py` file was created.

At this point, you should have:

- *Virtual environment set up*
- *Dependencies installed*
- *Configuration working*
- *GLM client that can talk to the API*
- *A file manager that can create project files*

Now, we build the brain of our autonomous agent, which is the main agent logic that ties everything together.

Step 6: The Main Agent



Create `agent.py`:

```
from glm_client import GLMClient
from file_manager import FileManager
import json
import re

class AutonomousCodingAgent:
    def __init__(self):
        self.client = GLMClient()
        self.file_manager = None
        self.project_plan = None

    def create_project(self, task_description, project_name="generated_project")
        """
        Main method that orchestrates the entire process.
        """
        print(f"\n{'='*60}")
        print(f"Starting autonomous coding agent...")
        print(f"Task: {task_description}")
        print(f"{'='*60}\n")

        # Initialize file manager
        self.file_manager = FileManager(project_name)
        self.file_manager.create_project_directory()
```

```
        # Phase 1: Planning
        print("\n[PHASE 1: PLANNING]")
        self.project_plan = self._plan_implementation(task_description)

        # Phase 2: Code Generation
        print("\n[PHASE 2: CODE GENERATION]")
        generated_files = self._generate_code()

        # Phase 3: Testing & Review
        print("\n[PHASE 3: TESTING]")
        self._test_and_review()

        # Show results
        print("\n" + "="*60)
        print("PROJECT COMPLETED!")
        print("="*60)
        print(f"\nGenerated files:")
        print(self.file_manager.get_project_structure())

        # Show cost
        usage = self.client.get_token_usage()
        print(f"\n💰 Token Usage:")
        print(f"  Input tokens: {usage['input_tokens']:,}")
        print(f"  Output tokens: {usage['output_tokens']:,}")
        print(f"  Total cost: ${usage['total_cost']}")

        return self.file_manager.project_path

    def _plan_implementation(self, task_description):
        """
        Phase 1: Ask GLM 4.6 to create a detailed plan.
        """
        planning_prompt = f"""You are an expert software architect.

Task: {task_description}

Create a detailed implementation plan. Your response must be in JSON format with

{{
    "overview": "Brief description of what we're building",
    "files": [
        {{
            "filename": "app.py",
            "purpose": "Main application entry point",
            "dependencies": []
        }}
    ],
    "implementation_steps": [
        "Step 1: ...",
        "Step 2: ..."
```

```
    ]
}}

Be specific about file structure and dependencies. Think through the architectur
"""

        print("⏳ Asking GLM 4.6 to plan the implementation...")
        response = self.client.call_glm(planning_prompt, enable_thinking=True)

        # Extract JSON from response
        plan = self._extract_json(response)

        if plan:
            print("✓ Plan created successfully")
            print(f"\nOverview: {plan.get('overview', 'N/A')}")
            print(f"Files to create: {len(plan.get('files', []))}")
            return plan
        else:
            print("✗ Failed to create plan")
            return None

    def _generate_code(self):
        """
        Phase 2: Generate code for each file in the plan.
        """
        if not self.project_plan:
            print("✗ No plan available")
            return []

        files = self.project_plan.get('files', [])
        generated_files = []

        for idx, file_info in enumerate(files, 1):
            filename = file_info['filename']
            purpose = file_info['purpose']

            print(f"\n[{idx}/{len(files)}] Generating {filename}...")

            code_prompt = f"""Based on our implementation plan, write the comple

File: {filename}
Purpose: {purpose}

Requirements:
- Write production-ready code
- Include error handling
- Add docstrings and comments
- Follow Python best practices
- Make it work with the other files in our project
```

```
Return ONLY the code, no explanations. Start with the imports.
"""

            print(f"⏳ Asking GLM 4.6 to write {filename}...")
            code = self.client.call_glm(code_prompt, enable_thinking=False)

            # Clean up the code (remove markdown formatting if present)
            code = self._clean_code(code)

            # Save to file
            self.file_manager.create_file(filename, code)
            generated_files.append(filename)

        print(f"\n✓ Generated {len(generated_files)} files")
        return generated_files

    def _test_and_review(self):
        """
        Phase 3: Ask GLM 4.6 to review the code and suggest tests.
        """
        review_prompt = f"""Review the code we just generated for this project:

{self.file_manager.get_project_structure()}

Tasks:
1. Identify any obvious bugs or issues
2. Suggest improvements
3. Create a simple test plan

Be specific and practical. Focus on real issues that would prevent this from wor
"""

        print("⏳ Asking GLM 4.6 to review the code...")
        review = self.client.call_glm(review_prompt, enable_thinking=True)

        print("\n📋 Code Review:")
        print(review[:500] + "..." if len(review) > 500 else review)

        # Generate a README
        readme_content = f"""# {self.file_manager.project_name}

## Overview
{self.project_plan.get('overview', 'Generated project')}

## Setup

1. Create virtual environment:
```bash
python -m venv venv
source venv/bin/activate  # On Windows: venv\\Scripts\\activate
```
```

```
```

2. Install dependencies:
```bash
pip install -r requirements.txt
```

3. Run the application:
```bash
python app.py
```

Project Structure
{self.file_manager.get_project_structure()}

Code Review Notes
{review[:300]}

Generated by GLM 4.6 Autonomous Agent
"""

        self.file_manager.create_file("README.md", readme_content)

        # Generate requirements.txt based on common imports
        self._generate_requirements()

    def _generate_requirements(self):
        """Generate requirements.txt based on imports."""
        # Common Python packages and their pip names
        common_packages = {
            'flask': 'flask',
            'jwt': 'PyJWT',
            'bcrypt': 'bcrypt',
            'requests': 'requests',
            'numpy': 'numpy',
            'pandas': 'pandas'
        }

        requirements = set()

        # Scan generated files for imports
        for file_path in self.file_manager.created_files:
            if file_path.endswith('.py'):
                try:
                    with open(file_path, 'r') as f:
                        content = f.read()
                        for package, pip_name in common_packages.items():
                            if f'import {package}' in content or f'from {package
                                requirements.add(pip_name)

```
                except:
                    pass

        if requirements:
            req_content = '\n'.join(sorted(requirements))
            self.file_manager.create_file("requirements.txt", req_content)

    def _extract_json(self, text):
        """Extract JSON from GLM response."""
        try:
            # Try direct JSON parse first
            return json.loads(text)
        except:
            # Look for JSON in markdown code blocks
            json_match = re.search(r'```json\s*(\{.*?\})\s*```', text, re.DOTALL
            if json_match:
                return json.loads(json_match.group(1))

            # Look for JSON without markdown
            json_match = re.search(r'(\{.*\})', text, re.DOTALL)
            if json_match:
                return json.loads(json_match.group(1))

        return None

    def _clean_code(self, code):
        """Remove markdown formatting from code."""
        # Remove markdown code blocks
        code = re.sub(r'```python\s*', '', code)
        code = re.sub(r'```\s*', '', code)
        return code.strip()


def main():
    """
    Main entry point for the agent.
    """
    agent = AutonomousCodingAgent()

    # The task we want to build
    task = """
    Build a task manager API with the following features:
    - User registration and login with JWT authentication
    - Create, read, update, and delete tasks
    - Each task has: title, description, status (todo/in-progress/done), created
    - Store data in JSON files (no database)
    - Input validation and error handling
    - RESTful API design
    """
```

```
    # Run the agent
    project_path = agent.create_project(task, project_name="task_manager_api")

    print(f"\n Project created at: {project_path}")
    print("\nNext steps:")
    print("1. cd generated_projects/task_manager_api")
    print("2. python -m venv venv && source venv/bin/activate")
    print("3. pip install -r requirements.txt")
    print("4. python app.py")


if __name__ == "__main__":
    main()
```

Testing the Setup First

Before running the full autonomous agent, I tested each component to ensure everything worked as expected.

Here's what happened.

First Test: Configuration

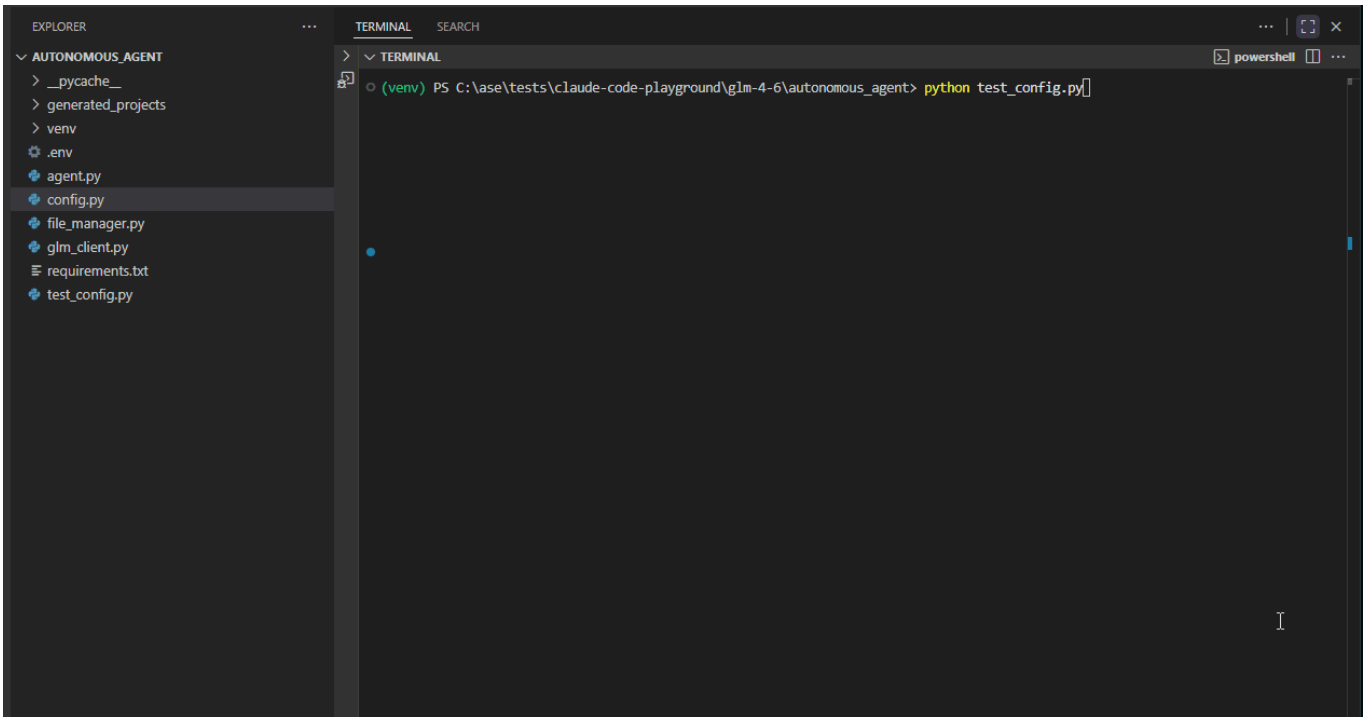
I started with the basics. Testing if the API key loaded correctly.

```
from config import Config

print("Testing configuration...")
print(f"API Key loaded: {Config.API_KEY[:10] if Config.API_KEY else 'NOT FOUND'}")
print(f"API URL: {Config.API_URL}")
print(f"Model: {Config.MODEL}")

if Config.API_KEY:
```

```
    print("✓ Configuration looks good!")
else:
    print("✗ API Key not found. Check your .env file")
```

```
python test_config.py
```

Result:




```
Testing configuration...
API Key loaded: 93f375b251...
API URL: https://api.z.ai/api/paas/v4/chat/completions
Model: glm-4.6
✓ Configuration looks good!
```

No issues here. The environment setup was solid.

Second Test: GLM 4.6 Connection (First Attempt)

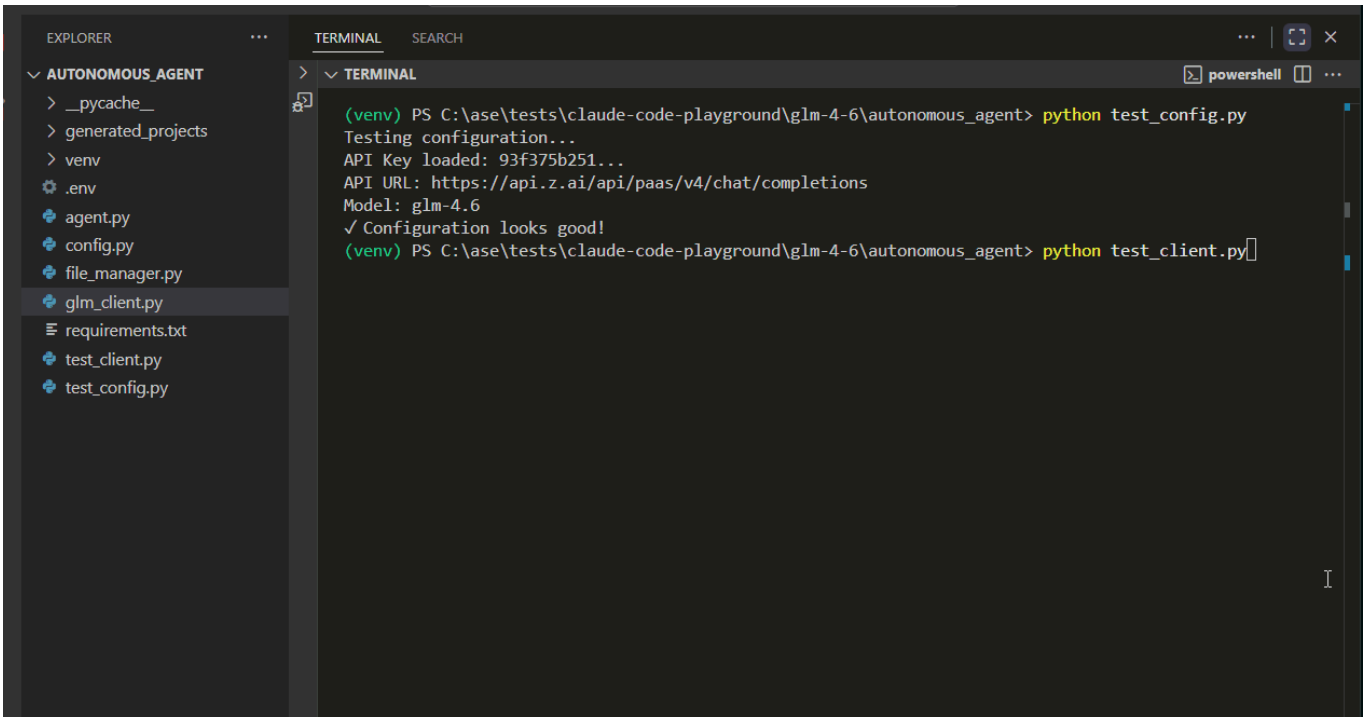
Next, I tested the actual API connection to GLM 4.6.

```
python test_client.py
```

```
from glm_client import GLMClient

print("Testing GLM 4.6 connection...")
client = GLMClient()

response = client.call_glm("Say 'Hello! GLM 4.6 is working!' in one short senten

if response:
    print(f"\n✓ Response from GLM 4.6:")
    print(f"  {response}")
    print(f"\n Token usage: {client.get_token_usage()}")
else:
    print("\n✗ Failed to connect to GLM 4.6")
    print("Check your API key and internet connection")
```

Result:

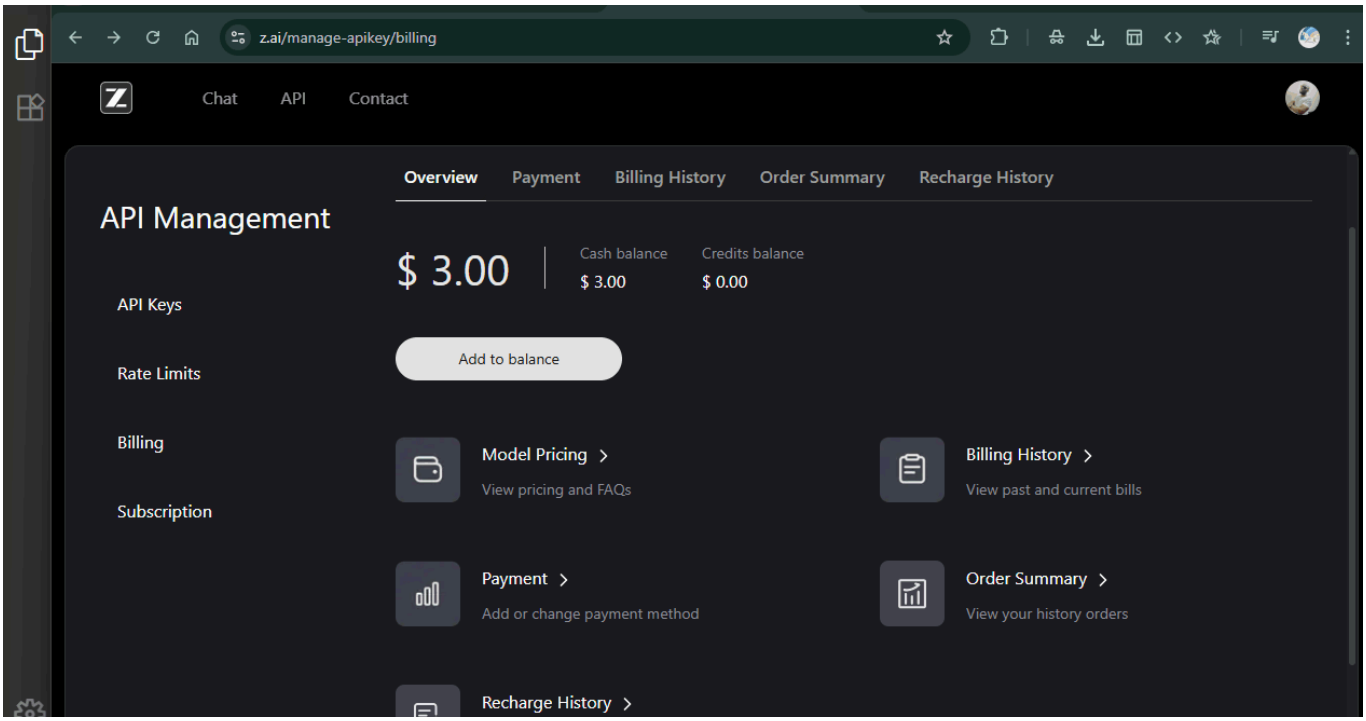
```
API Error: 429 Client Error: Too Many Requests
✗ Failed to connect to GLM 4.6
Check your API key and internet connection
```



Hit a wall. The 429 error meant I exceeded my quota or didn't have credits.

The Fix: Adding Credits

I went to z.ai and checked my account. The free tier quota was exhausted. I added $3 in credits to my account.



This took about 2 minutes:

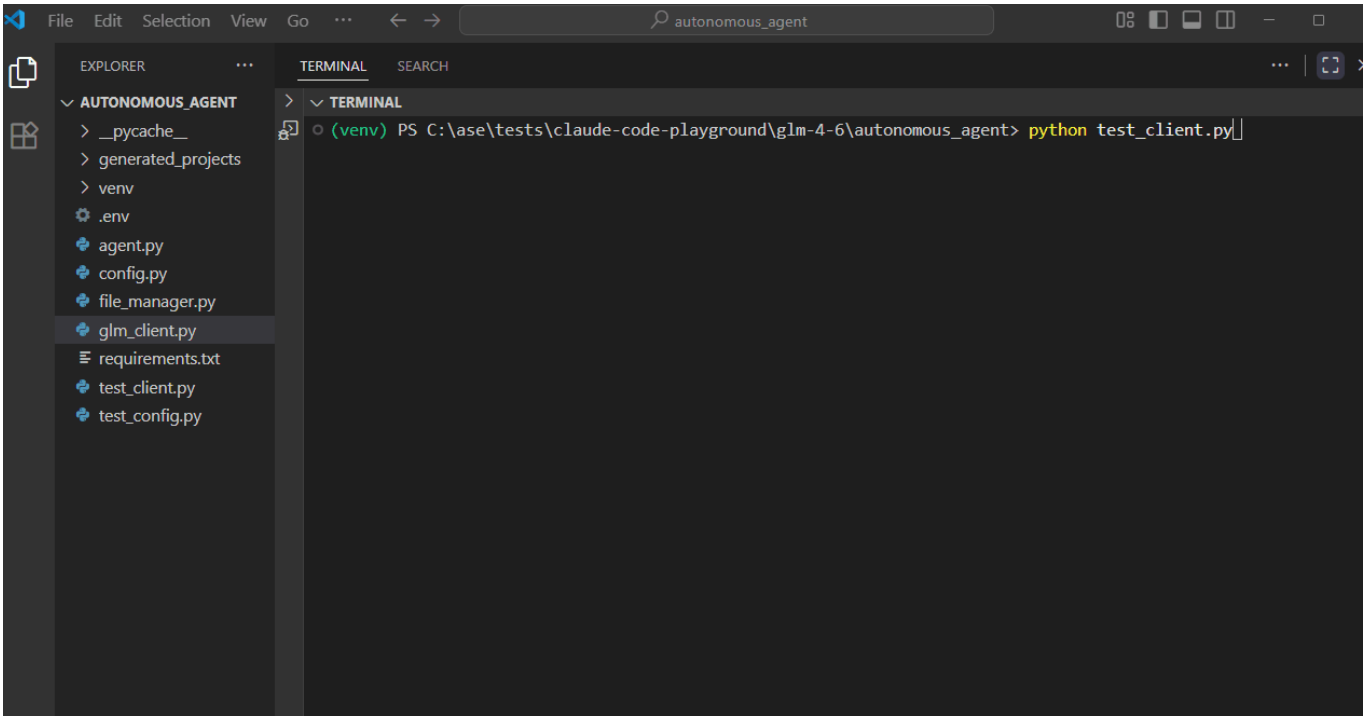
1. *Logged into z.ai*
2. *Went to billing*
3. *Added $3 credit*
4. *Waited for it to process*

GLM 4.6 Connection (Success)

Ran the test again after adding credits.

```
python test_client.py
```

Result:




```
Testing GLM 4.6 connection...
✓ Response from GLM 4.6:
  Hello! GLM 4.6 is working!

Token usage: {'input_tokens': 25, 'output_tokens': 102, 'total_tokens': 127,
'total_cost': 0.0002}
```

Success. GLM 4.6 responded perfectly.

> *The cost for this tiny test: $0.0002 (two hundredths of a cent).*

If you run into the 429 error like I did:

Option 1: Add Credits

- Go to z.ai
- Add credits (minimum $3)
- Test again immediately

Option 2: Subscribe to GLM Coding Plan

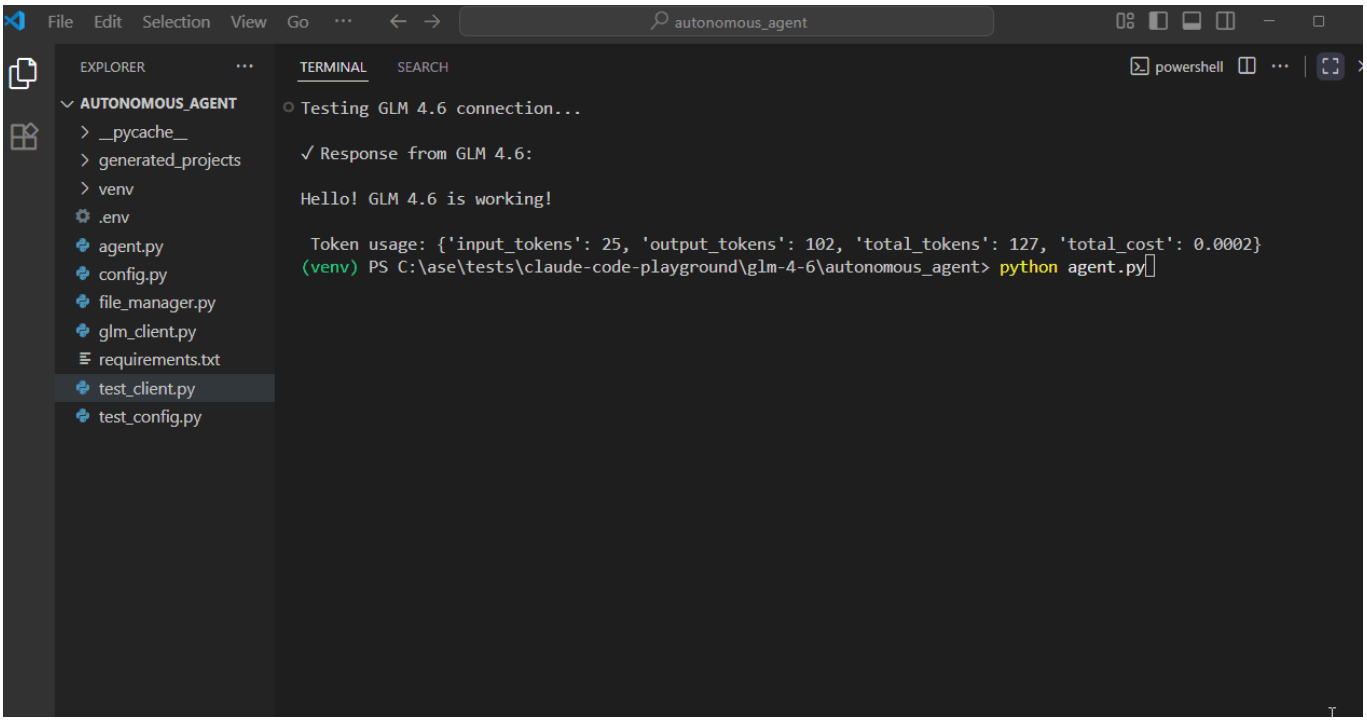
- $3 per month
- 120 prompts per 5-hour cycle
- Better for regular use

> *With the setup confirmed to work, I was ready to run the full autonomous agent.*

Running the Agent: Watching GLM 4.6 Work Autonomously

Time to see what GLM 4.6 can actually do. I ran the agent and documented everything in real-time.

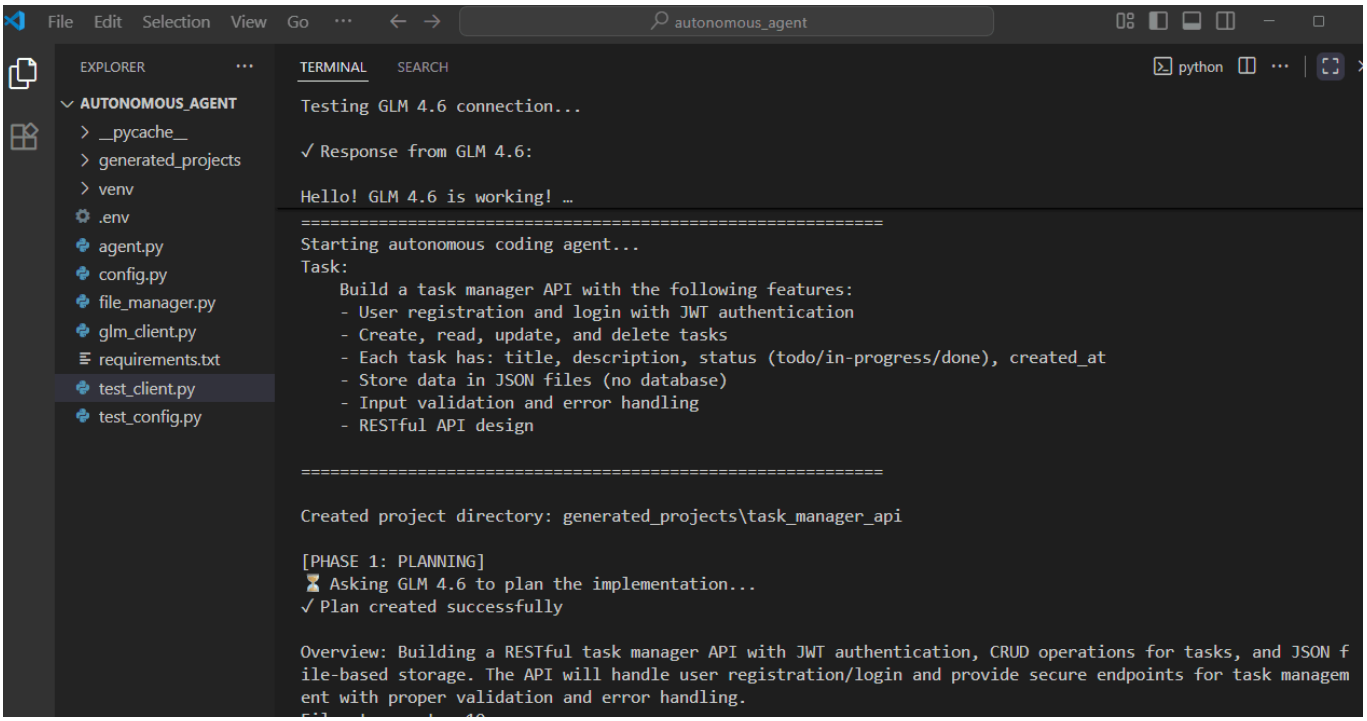
```
python agent.py
```



Phase 1: Planning

GLM 4.6 started by analyzing the requirements. No rushing straight to code. It thought through the architecture first.

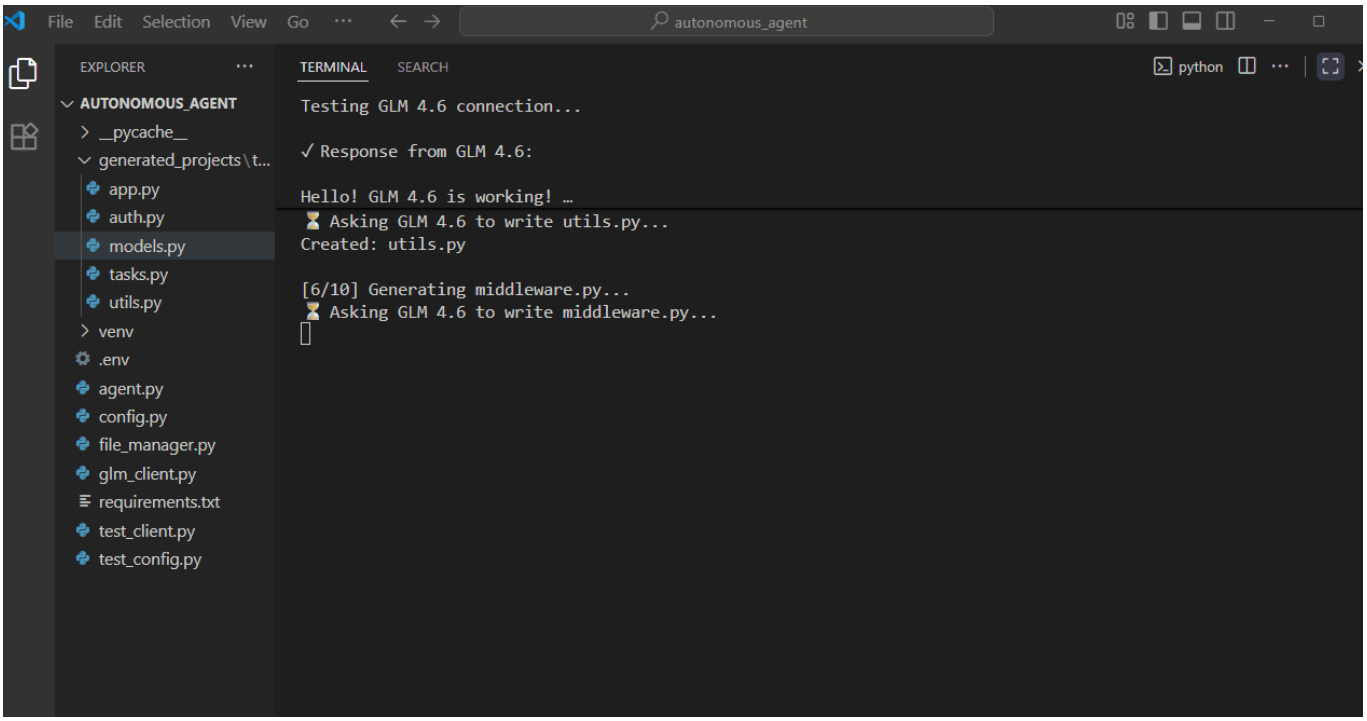
```
[PHASE 1: PLANNING]
⏳ Asking GLM 4.6 to plan the implementation...
✓ Plan created successfully

Overview: Building a RESTful task manager API with JWT authentication,
CRUD operations for tasks, and JSON file-based storage. The API will
handle user registration/login and provide secure endpoints for task
management with proper validation and error handling.

Files to create: 10
```



GLM 4.6 decided to properly separate concerns instead of cramming everything into 2–3 files—good architectural thinking.

Phase 2: Code Generation

The agent went through each file systematically. Here's what it generated:




```
[PHASE 2: CODE GENERATION]

[1/10] Generating app.py...
✓ Created: app.py

[2/10] Generating models.py...
✓ Created: models.py

[3/10] Generating auth.py...
✓ Created: auth.py

[4/10] Generating tasks.py...
✓ Created: tasks.py

[5/10] Generating utils.py...
✓ Created: utils.py

[6/10] Generating middleware.py...
✓ Created: middleware.py

[7/10] Generating config.py...
✓ Created: config.py

[8/10] Generating requirements.txt...
✓ Created: requirements.txt

[9/10] Generating data/users.json...
```

```
✓ Created: data/users.json

[10/10] Generating data/tasks.json...
✓ Created: data/tasks.json
```

Each file took 15–20 seconds to generate.

What impressed me: GLM 4.6 created the `data/` directory structure and initialized JSON files.

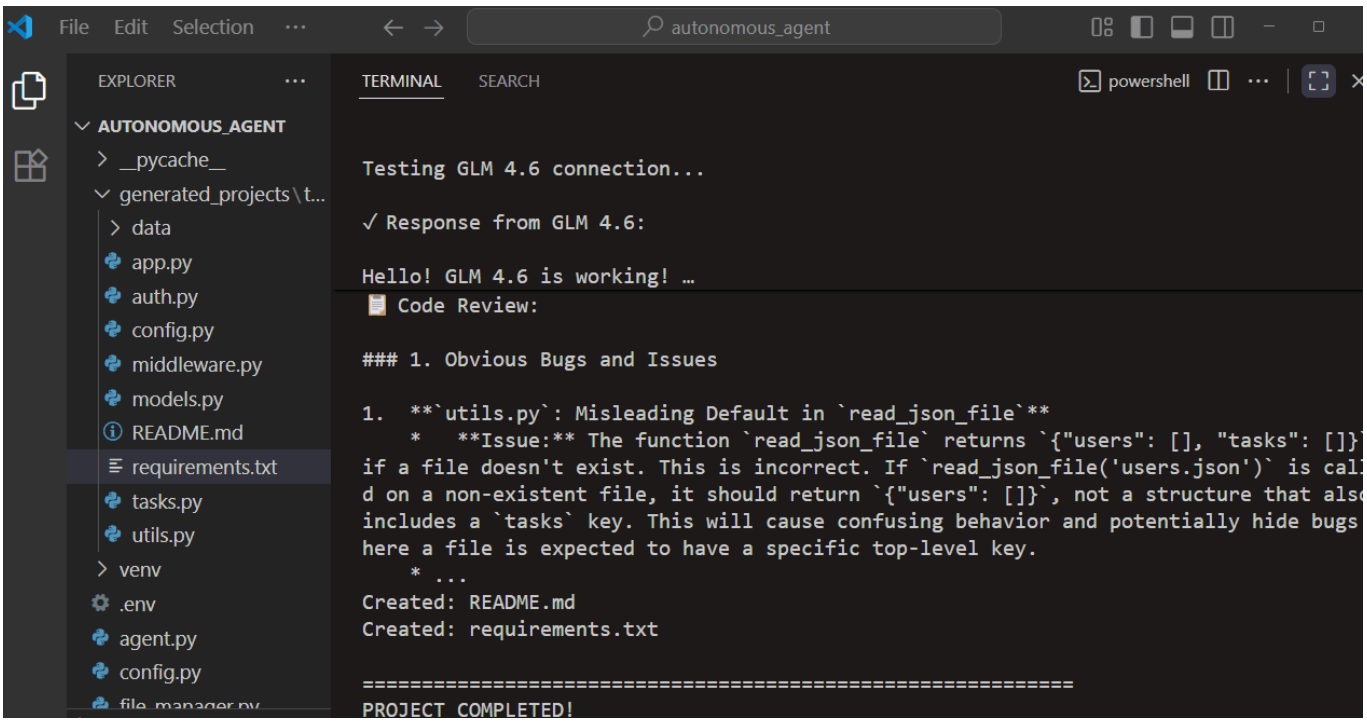
> *It didn't just write Python code. It set up the complete working environment.*

Phase 3: Code Review

GLM 4.6 reviewed its own code and caught issues:

```
[PHASE 3: TESTING]
⏳ Asking GLM 4.6 to review the code...

📋 Code Review:
### 1. Obvious Bugs and Issues

1. **`utils.py`: Misleading Default in `read_json_file`**
   - Issue: The function returns {"users": [], "tasks": []} if a file
     doesn't exist. This is incorrect. If read_json_file('users.json')
     is called, it should return {"users": []}, not a structure that
     also includes a tasks key.
```

```
    - Impact: Confusing behavior and potential bugs.
    - Fix: Return appropriate default based on the file type.
```

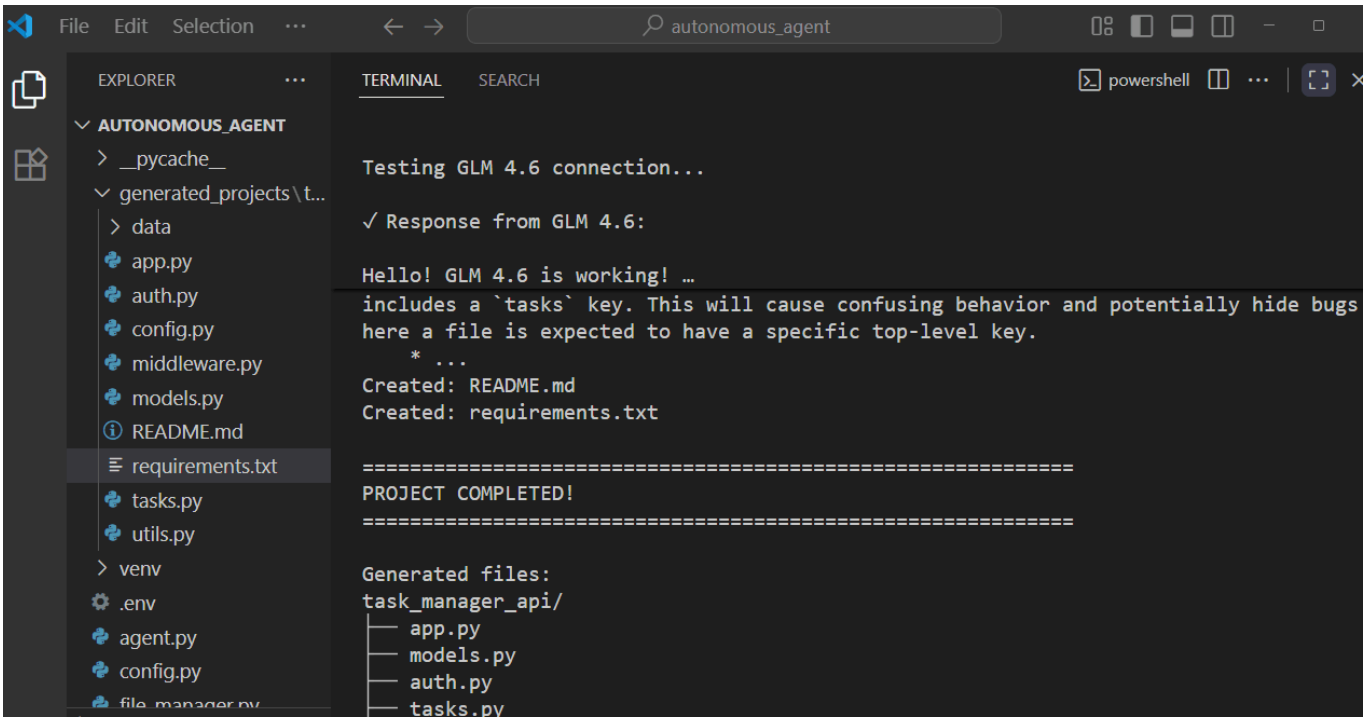


This is autonomous behavior.

> *It found a logic flaw in its own code and documented it. The default return value in `utils.py` was too generic.*

The agent also generated:

- Complete README with setup instructions
- requirements.txt with all dependencies



The final completed project summary was as follows :

```
============================================================
PROJECT COMPLETED!
============================================================

Generated files:
task_manager_api/
├── app.py
├── models.py
├── auth.py
├── tasks.py
├── utils.py
├── middleware.py
├── config.py
├── requirements.txt
├── data/users.json
├── data/tasks.json
└── README.md

💰 Token Usage:
  Input tokens: 98,710
  Output tokens: 18,329
  Total cost: $0.0959
```

Total tokens: 117,039

- Input: 98,710 tokens (context + prompts)
- Output: 18,329 tokens (generated code)

Total cost: $0.0959 (less than 10 cents)

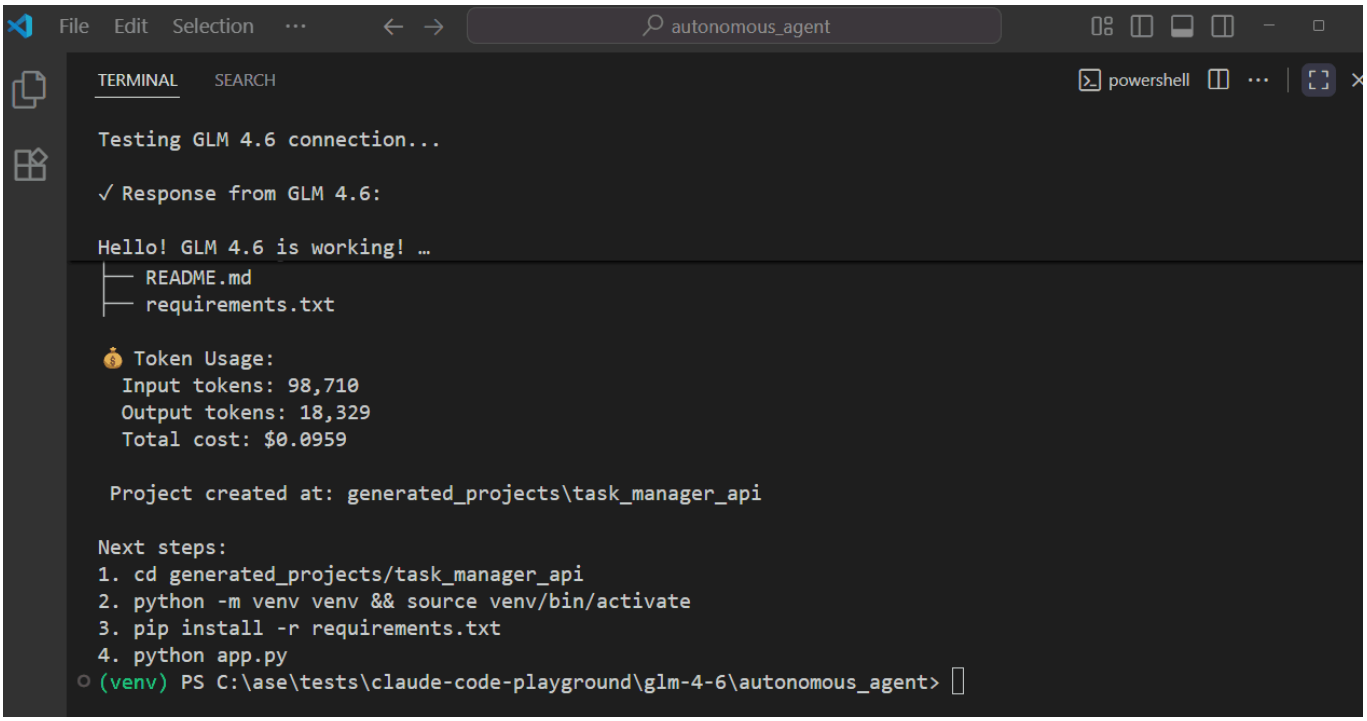
Breaking it down:

- Input cost: 98,710 × $0.60/1M = $0.0592
- Output cost: 18,329 × $2.00/1M = $0.0367
- **Total: $0.096** (about 10 cents)

For comparison, the same project with Claude 4.5 Sonnet would cost approximately:

- Input: 98,710 × $3.00/1M = $0.296
- Output: 18,329 × $15.00/1M = $0.275
- **Total: $0.571** (about 57 cents)

> *GLM 4.6 delivered this project at 17% of Claude's cost.*



Time Breakdown

From start to finish: **3 minutes 40 seconds**

- Planning: 30 seconds
- Code generation: 2 minutes 45 seconds (10 files)
- Review and documentation: 25 seconds

For context, building this manually would take me 45–60 minutes. The agent did it in under 4 minutes.

Generated Project Structure

GLM 4.6 organized the code intelligently:

Core application:

- `app.py` - Main Flask application and routes
- `models.py` - Data models (User, Task)
- `config.py` - Configuration settings

Business logic:

- `auth.py` - Authentication and JWT handling
- `tasks.py` - Task CRUD operations
- `utils.py` - Helper functions for file operations

Infrastructure:

- `middleware.py` - Request/response middleware
- `requirements.txt` - Dependencies
- `data/` - JSON storage files
- `README.md` - Complete documentation

Proper separation of concerns, not everything dumped into one file.

Final Thoughts

GLM 4.6 delivered. The autonomous agent built a complete task manager API in under 4 minutes for less than 10 cents.

> *The code structure was solid — proper separation of concerns, thoughtful file organization, and working authentication logic. It even caught its own bugs during the review phase. For most development work — prototypes, internal tools, learning projects — GLM 4.6 handles it at a fraction of Claude's cost.*

At $0.60/$2.00 per million tokens versus Claude's $3.00/$15.00, you're looking at 5–7x savings. The $3 monthly subscription versus $20 further widens that gap.

If you're building daily or running agentic workflows that burn through API calls, those savings compound fast. I spent $0.096 on this complete project. The same build with Claude would have cost $0.571. ***Over dozens of projects, that difference funds your entire subscription.***

> *Is GLM 4.6 perfect? No. It lags Claude 4.5 on the hardest coding benchmarks; security defaults need review.*

But for the 80% of coding tasks that don't push absolute limits, it's a practical choice.

> *Have you tested GLM 4.6? What has been your experience?*

Let's Connect!

If you are new to my content, my name is Joe Njenga

> *Join thousands of other software engineers, AI engineers, and solopreneurs who read my content* ***daily on Medium*** *and on* ***YouTube where I review the latest AI***

> ***engineering tools and trends***. *If you are more curious about my projects and want to receive detailed guides and tutorials,* ***join thousands of other AI enthusiasts in my weekly AI Software engineer newsletter***

If you would like to connect directly, you can reach out here:

AI Integration Software Engineer (10+ Years Experience)

Software Engineer specializing in AI integration and automation. Expert in building AI agents, MCP servers, RAG...

njengah.com

Follow me on Medium | YouTube Channel | X | LinkedIn

Glm | Claude | Glm46 | Agentic Workflow

Written by Joe Njenga

4.5K followers · 99 following

Follow

Software & AI Automation Engineer, Tech Writer & Educator. Vision: Enlighten, Educate, Entertain. One story at a time. Work with me: mail.njengah@gmail.com

Responses (1)

Bgerby

What are your thoughts?

Christian Broberg
3 days ago

Is it possible to use the model in VS code through Copilot if I obtain an API key?

2 1 reply Reply

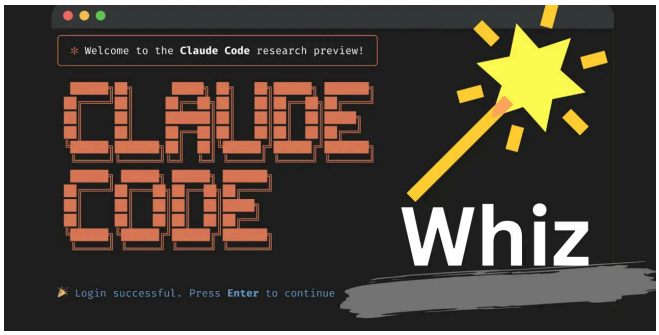
More from Joe Njenga

Joe Njenga

I Tested Upgraded Claude Code 2.0 (And Discovered 7 New Features...

Claude Code just dropped version 2.0, and they packed it with the most requested...

Sep 30 362 12

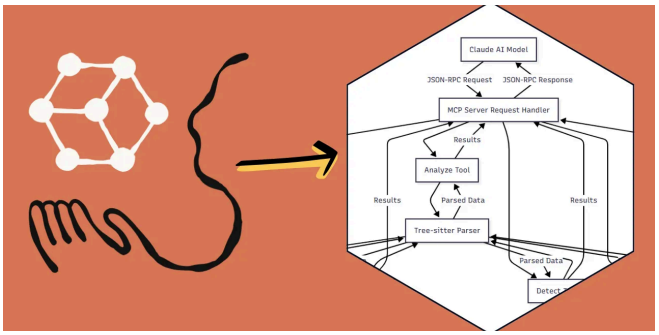


Joe Njenga

12 Little-Known Claude Code Commands That Make You a Whiz

What if I told you there are some Claude Code commands that, although not very popular,...

Sep 21 250 4

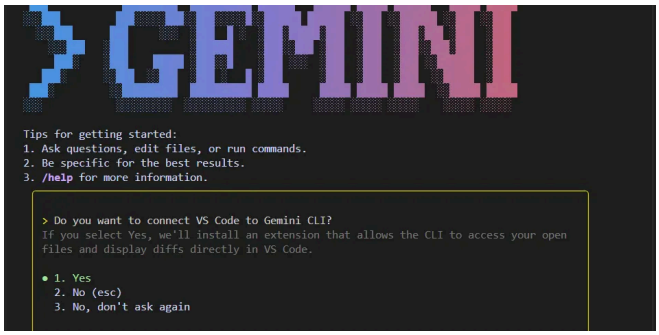


Joe Njenga

I Asked Claude 4.5 to Build Me a Complex MCP Server (It Was So...

If you want to build your custom MCP server, stop wasting time thinking, coding, or trying...

5d ago 365 7



Joe Njenga

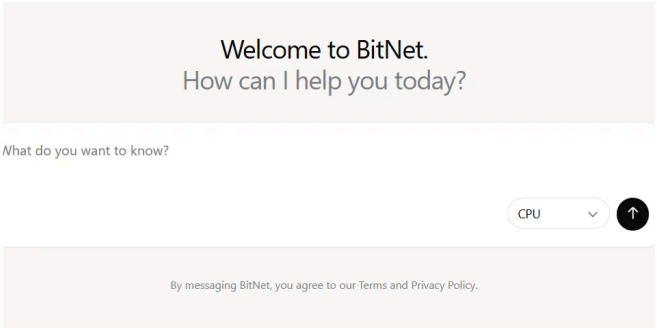
I Tested The Updated Gemini CLI (And Found These New Features...

Gemini CLI has these newly updated features that are making it edge closer to dethroning...

Sep 12 161 6

See all from Joe Njenga

Recommended from Medium

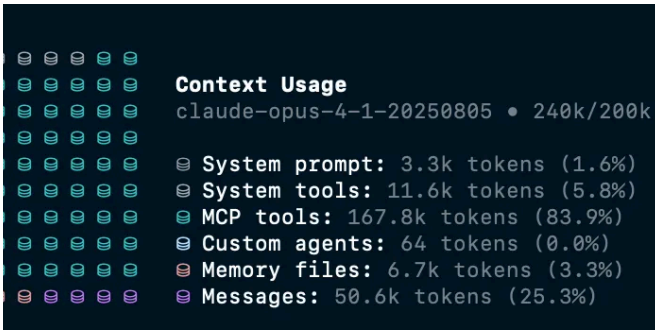


In Coding Nexus by Algo Insights

Microsoft Just Declared War on the GPU Mafia: Meet bitnet.cpp

Bitnet.cpp will break the GPU Lock

5d ago 320 5



Sevak Avakians

Claude Code Limit Hit on Max Plan?! What to do next:

$200/m plan blocks me for a week!

6d ago 85 17

In Data Science Collective by Erdogan T

Build Your Private Language Model: Local and Specialized For...

A complete step-by-step guide from setup to deployment of local language models, makin...

Oct 3 719 4

In Level Up Coding by Yasas Sandeepa

Google's New AI – Nano Banana: Top 10 Use Cases That Will Blow...

How Nano Banana is redefining the future of AI Image Gen

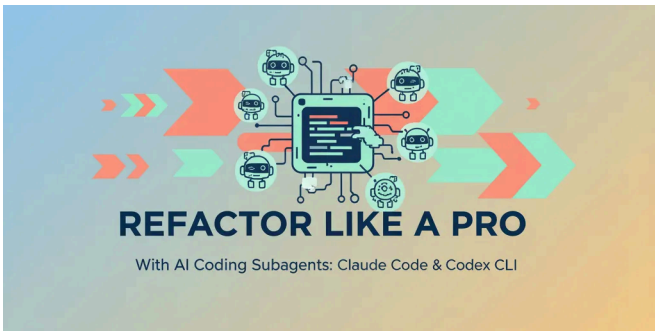
Oct 2 208 2

In Towards AI by Gao Dalie (高達烈)

Why Claude Sonnet 4.5 Is So Much Better Than GPT-5(Codex) And...

The new version of Claude, Sonnet 4.5, has been released. You might be wondering,...

3d ago 85



Reza Rezvani

The ultimate Code Modernization & Refactoring prompt for your...

Transform your legacy codebase chaos into a strategic modernization roadmap with this...

5d ago 86

See more recommendations