Building an AI Coding Agent from Scratch: A Complete Tutorial

Table of Contents

Introduction and Conceptual Foundation {#introduction}

This tutorial will guide you through building a Python-based AI coding agent using Google's Gemini Flash API. Unlike simple chatbots that just generate responses, this agent can interact with its environment by reading files, modifying code, and executing programs. The agent operates in an iterative loop, much like a human developer would, examining code, making changes, running tests, and refining its approach until it solves the problem at hand.

What You'll Build

By the end of this tutorial, you'll have created an AI agent capable of:

- Scanning directory structures and understanding project layouts
- Reading and analyzing source code files
- Writing and modifying files with new content
- Executing Python code and interpreting the results
- Iteratively debugging and fixing issues through multiple passes

This is essentially a "toy version" of sophisticated tools like GitHub Copilot or Cursor, but building it yourself will give you deep insights into how these tools actually work under the hood.

Learning Objectives

Beyond just building a functional agent, this project will help you:

- Understand the fundamental difference between stateless chatbots and stateful agents
- Learn how Large Language Models (LLMs) can be augmented with external tools through function calling
- Practice designing secure systems that limit AI access to authorized resources only
- Gain experience with API integration and token management
- Develop multi-directory Python projects with proper package structure
- Explore functional programming patterns in Python

Understanding AI Agents vs. Chatbots {#agents-vs-chatbots}

The Chatbot Paradigm

When you interact with a standard chatbot like ChatGPT, the conversation follows a simple pattern: you provide input, the model generates a response, and the conversation ends there. The model has no ability to verify its answers, test its code suggestions, or interact with the world beyond generating text. Each interaction is isolated, and the model relies entirely on its training data and your prompts.

The Agent Paradigm

An AI agent, by contrast, is fundamentally different in several key ways. First, an agent can take actions in its environment. In our case, this means the agent can read files, write code, and execute programs. Second, an agent operates in a loop rather than a single pass. It can propose a solution, test it, see the results, and iterate based on what it learns. Third, an agent maintains context across multiple steps, building up a conversation history that helps it understand what it has tried and what worked or didn't work.

Think of the difference this way: a chatbot is like asking a friend for advice on fixing your car over the phone. They can give you suggestions based on what you describe, but they can't actually look at the car themselves. An agent is like having that friend come to your garage, where they can inspect the car, try different solutions, and see the results of their work directly.

The Agentic Loop

The core of any agent is its agentic loop, which follows this pattern:

1. Receive an initial goal or problem from the user
2. Analyze the current state by examining available information
3. Decide on an action to take using one of its available tools
4. Execute that action and observe the results
5. Update its understanding based on the results
6. Determine if the goal is achieved; if not, return to step three
7. Provide a final response to the user

This loop allows the agent to be self-correcting and adaptive. If its first approach doesn't work, it can try something different. If it encounters an error, it can analyze the error message and adjust its strategy.

Project Architecture and Design Philosophy {#architecture}

High-Level Architecture

The agent system consists of three main layers that work together:

The first layer is the CLI interface, which handles user input and command-line arguments. This layer is responsible for parsing the user's prompt, handling flags like verbose mode, and displaying results in a user-friendly format.

The second layer is the agent core, which manages the interaction with the Gemini API. This layer maintains the conversation history, determines when to call functions versus when to return a final answer, and orchestrates the overall agentic loop. It's essentially the "brain" of the operation, deciding what to do next based on the LLM's responses.

The third layer consists of the tool functions themselves. These are the concrete Python functions that actually perform actions like reading files or executing code. Each tool function is paired with a schema that describes it to the LLM, allowing the model to understand what tools are available and how to use them.

Design Decisions and Trade-offs

Several important design decisions shape this project, each made for specific pedagogical and practical reasons.

Manual vs. Automatic Function Calling: The project uses manual function calling rather than Gemini's built-in automatic function calling feature. While automatic function calling is more convenient, it can have parsing issues with complex type signatures. More importantly, manual function calling gives us direct control over the process and helps students understand exactly how function calling works under the hood. You can see each step: the LLM decides to call a function, we parse that decision, we execute the function, and we feed the results back to the LLM.

Security-First Design: Every tool function includes extensive security checks before performing any operation. This isn't just defensive programming; it's an essential recognition that we're giving an AI model the ability to modify and execute code on our system. While the LLM is generally helpful, it could potentially be manipulated by carefully crafted prompts to access files it shouldn't. By restricting all operations to a designated working directory and validating every path, we create a sandbox environment where the agent can work freely without risking the host system.

Token Budget Management: File reading is limited to 10,000 characters, and code execution has a 30-second timeout. These limits exist because API calls cost money based on tokens processed, and we want to avoid accidentally consuming large amounts of tokens by reading massive files or getting stuck in infinite loops. In a production system, you might adjust these limits based on your needs and budget.

Multi-Directory Structure: The calculator example project deliberately uses a multi-directory package structure with a pkg subdirectory. This isn't just to show off Python packaging; it's to demonstrate that the agent can navigate and understand complex project layouts, not just flat file structures.

File Structure Overview

The project is organized as follows:

```
ai_agent/
├── main.py                    # Entry point and CLI handling
├── functions/
|   └── get_files_info.py     # All tool functions and schemas
├── calculator/                # Working directory for the agent
|   ├── main.py
|   ├── tests.py
|   └── pkg/
|       ├── calculator.py
|       └── render.py
├── .env                       # API key storage (not in version control)
└── pyproject.toml            # Project dependencies
```

Prerequisites and Setup {#setup}

System Requirements

Before starting, ensure you have Python 3.12 or later installed on your system. You can verify your Python version by running `python --version` in your terminal. The project specifically requires Python 3.12 or later because we use some modern Python features and type hints that aren't available in earlier versions.

You'll also need the UV package manager, which is a fast, Rust-based alternative to pip and poetry. UV handles dependency management and virtual environments automatically, making project setup much simpler. If you don't have UV installed, you can get it from the official UV website or install it via your package manager.

Finally, you'll need a Unix-like shell environment. If you're on macOS or Linux, your default terminal will work fine. If you're on Windows, we strongly recommend using Windows Subsystem for Linux (WSL) rather than trying to adapt the file path handling for Windows-style paths.

Getting Your Gemini API Key

Google's Gemini API provides the intelligence behind our agent. The free tier is more than sufficient for this project, offering generous rate limits that will accommodate all your learning and experimentation.

To get your API key, navigate to Google AI Studio in your web browser. You'll need to sign in with a Google account if you're not already signed in. Once in AI Studio, look for the option to create an API key. The interface will guide you through creating a new API key for the Gemini API specifically.

When you receive your API key, it will be a long string of characters that looks something like `AIzaSyD...` (truncated for security). This key is essentially a password that allows your code to access the Gemini API, so treat it with care.

Secure API Key Storage

Never, under any circumstances, commit your API key to version control or share it publicly. If your API key is exposed, anyone who finds it can use your quota and potentially rack up charges if you exceed the free tier.

The proper way to store your API key is in a `.env` file that lives in your project root but is explicitly excluded from version control. Create a file named `.env` (note the leading dot) in your project directory and add this line:

```
GEMINI_API_KEY=your_actual_api_key_here
```

Then, make absolutely certain that your `.gitignore` file includes `.env` on its own line. This tells Git to never track or commit this file, keeping your secrets safe.

In your Python code, you'll load this environment variable using a library like `python-dotenv`, which reads the `.env` file and makes those values available through `os.getenv()`. This pattern is standard practice in professional development for managing secrets and configuration that varies between environments.

Initial Project Setup

Clone the repository and navigate into the project directory. Create your `.env` file as described above. Then run `uv sync` to install all dependencies and set up the virtual environment. UV will create a `.venv` directory containing an isolated Python environment with all the required packages. You don't need to manually activate this environment; UV commands automatically use it.

Core Concepts: LLMs, Tokens, and APIs {#core-concepts}

What is a Large Language Model?

A Large Language Model like Gemini is a neural network trained on massive amounts of text data. Through this training, the model learns patterns in language, coding conventions, common reasoning patterns, and general knowledge about the world. When you give it a prompt, the model predicts what text should come next based on those learned patterns.

Think of it like an incredibly sophisticated autocomplete. But instead of just suggesting the next word, it can generate entire paragraphs, code files, or reasoning chains that are contextually appropriate and logically sound.

Understanding Tokens

When you interact with an LLM API, you're charged based on "tokens" rather than characters or words. A token is roughly equivalent to four characters on average, but the exact tokenization depends on the specific model and language. For example, common English words might be a single token, while rare words or code might take multiple tokens.

This matters because every API call includes both input tokens (your prompt plus any conversation history) and output tokens (the model's response). Reading a large file and including its contents in your prompt could consume thousands of tokens, which is why we implement the 10,000 character limit on file reading.

The model also has a context window, which is the maximum number of tokens it can consider at once. For Gemini Flash, this window is quite large, but in a long agentic loop with extensive conversation history, you could theoretically hit this limit. In practice, for our use case, this is unlikely to be an issue.

How API Calls Work

When you make an API call to Gemini, you're sending an HTTP request to Google's servers with your prompt and any configuration. The server processes your request using the LLM, generates a response, and sends it back to you. This all happens in a few seconds for typical requests.

The basic flow looks like this:

```
import google.generativeai as genai

# Configure the API client with your key
genai.configure(api_key=your_api_key)

# Create a model instance
model = genai.GenerativeModel("gemini-2.0-flash-exp")

# Generate a response
response = model.generate_content("Write a Python function to calculate factorial")

# Extract the text from the response
print(response.text)
```

This simple interaction is synchronous and stateless. Each call is independent, and the model doesn't remember previous interactions unless you explicitly include that history in your next request.

Function Calling: Extending the LLM

The real power of our agent comes from function calling, which is a feature that allows the LLM to request that specific functions be executed rather than just generating text. Here's how it works conceptually.

When you configure the model, you provide schemas that describe available functions. These schemas tell the model the function name, what it does, what parameters it takes, and what types those parameters should be. During generation, if the model determines that calling a function would be helpful, it can include a function call in its response instead of (or in addition to) regular text.

Your code then detects this function call, executes the actual Python function with the specified parameters, and sends the result back to the model as a new message in the conversation. The model can then use that information to continue its reasoning or make additional function calls as needed.

This is transformative because it means the LLM isn't limited to its training data. It can access real-time information, interact with external systems, and verify its own work by executing code and seeing the actual results.

Building the Agent: Step-by-Step {#building}

Step 1: The Main Entry Point

Let's start by understanding `main.py`, which serves as the entry point for our agent. This file handles command-line arguments, initializes the Gemini client, and manages the overall flow of the program.

The file begins by importing necessary dependencies and loading environment variables. We use `python-dotenv` to automatically load variables from the `.env` file, making the API key available through `os.getenv()`. This separation of configuration from code is a best practice that makes your code more maintainable and secure.

The command-line argument parsing uses Python's `argparse` module to define two arguments: the user's prompt (a required positional argument) and an optional `--verbose` flag. The verbose flag is particularly useful during

development because it shows you exactly what's being sent to the API, how many tokens are being used, and what responses are coming back. This transparency helps you understand what's happening at each step and debug issues when things don't work as expected.

Step 2: Initializing the Gemini Client

The Gemini client initialization is straightforward but important. After loading the API key from environment variables, we configure the `genai` library and create a model instance. The model string `"gemini-2.0-flash-exp"` specifies which version of Gemini we're using. The "Flash" variant is optimized for speed and cost-effectiveness, making it perfect for an application that might make many API calls in a single session.

When creating the model, we pass in the function declarations. These declarations are the schemas that describe our tools to the model. By including them at model initialization time, we're telling Gemini, "These are the tools you have available. You can call any of these functions when you think they would help you accomplish the user's goal."

Step 3: Understanding Function Schemas

Function schemas are JSON-like structures that describe each tool in a way the LLM can understand. A schema includes the function name, a description of what it does, and a parameters object that details each parameter.

Let's look at an example schema for the `get_files_info` function:

```
{
    "name": "get_files_info",
    "description": "List all files and subdirectories in a directory with metadata",
    "parameters": {
        "type": "object",
        "properties": {
            "directory": {
                "type": "string",
                "description": "Path to the directory (relative to working directory)"
            }
        },
        "required": ["directory"]
    }
}
```

The description is crucial because it's how the model understands when to use this function. A vague description like "gets files" might not be clear enough, whereas "List all files and subdirectories in a directory with metadata" tells the model exactly what this function does and when it would be appropriate to use.

The parameters section uses JSON Schema format to describe the expected inputs. Here we're saying the function takes one parameter called `directory`, which is a string representing a path. The `required` array indicates that this parameter must be provided.

The model uses these schemas to determine not only which function to call but also what arguments to pass. When it decides to call `get_files_info`, it will generate a function call with a `directory` argument that makes sense in the context of the user's request.

Step 4: Implementing the Tool Functions

Each tool function follows a similar pattern: validate inputs, perform security checks, execute the operation, and return results in a structured format. Let's examine each function in detail.

get_files_info: This function lists the contents of a directory, returning information about each file including its name, size, and whether it's a directory or a regular file. The implementation uses `os.scandir()` which is more efficient than `os.listdir()` because it provides file metadata without additional system calls.

The security check is critical here. Before listing anything, the function resolves the requested path to an absolute path and verifies it's within our working directory. The check uses `os.path.commonpath()` to ensure the resolved path is a subdirectory of the working directory. This prevents path traversal attacks where someone might try to access files

outside the working directory using paths like `"../../etc/passwd"`.

get_file_content: Reading file contents is straightforward, but again we start with security validation. The function resolves the path, checks it's within our working directory, and verifies the file exists and is actually a file (not a directory or special file).

The 10,000 character limit is implemented here. If a file is longer than 10,000 characters, we read only the first 10,000 and append a message indicating the file was truncated. This prevents token budget overruns while still giving the model enough context to understand the file's contents. In most cases, 10,000 characters is more than enough to understand what a file does.

write_file: Writing files is where the agent can make changes. This function can both create new files and overwrite existing ones. An important feature is the automatic creation of parent directories using `os.makedirs(exist_ok=True)`. This means if the model wants to create a file at `"pkg/utils/helper.py"` and the directories don't exist, they'll be created automatically.

Again, security checks ensure we're only writing within the working directory. We use `'w'` mode which will create the file if it doesn't exist or overwrite it if it does. The function returns a simple success message that the model can read to confirm the operation worked.

run_python_file: This is perhaps the most powerful and dangerous tool. It allows the model to execute Python code and see the results. The function first validates that the file path ends with `.py` (preventing execution of arbitrary files) and that it's within our working directory.

The execution uses `subprocess.run()` with several important parameters. `capture_output=True` captures both stdout and stderr so we can return any output or errors to the model. `text=True` ensures we get string output rather than bytes. `timeout=30` prevents the agent from getting stuck if it accidentally creates an infinite loop or very slow code. The function also handles `TimeoutExpired` exceptions gracefully, returning an error message instead of crashing.

One subtle but important detail: we execute the file using `sys.executable` to ensure we're using the same Python interpreter that's running the agent. This matters because different Python versions might behave differently.

Step 5: The Function Dispatcher

The `call_function()` function in `get_files_info.py` acts as a dispatcher that maps function names to actual Python functions. This abstraction is important because it separates the agent's decision to call a function from the implementation details of how to call it.

The function takes a `function_call_part` from the API response, extracts the function name and arguments, and uses a dictionary mapping to find and call the appropriate Python function. This dictionary pattern is cleaner and more maintainable than a long chain of if-elif statements.

Error handling here is important. If the function name isn't recognized, we return an error message rather than crashing. If the function raises an exception during execution, we catch it and return the error message so the model can see what went wrong and potentially adjust its approach.

Step 6: The Agentic Loop

The heart of the agent is the agentic loop in `main.py`. This loop is what transforms a simple API client into a true agent capable of multi-step reasoning and action.

The loop maintains a conversation history as a list of messages. Each message has a role (either "user", "model", or "function") and content. The initial user message is the prompt provided at the command line.

The loop proceeds through iterations up to a maximum count (typically set to 10). In each iteration, we send the current conversation history to the model. The model then responds with either a text message (indicating it's done) or a function call (indicating it wants to take an action).

When the model requests a function call, we extract the function details, execute the corresponding Python function, and create a new function response message. This message includes both the function name and its result, allowing the

model to see what happened. We append this message to the conversation history and loop again.

This continues until either the model returns a text response (indicating it has the final answer) or we hit the maximum iteration count. The maximum is a safety measure to prevent infinite loops if something goes wrong with the model's decision-making.

Step 7: System Prompts and Model Guidance

While not always visible in the code, the system prompt plays a crucial role in guiding the model's behavior. A system prompt is a special message that carries more weight than regular user messages and helps establish the model's personality, capabilities, and constraints.

A good system prompt for this agent might include instructions like: "You are a helpful coding assistant with access to file system tools. When the user asks you to fix bugs, read the relevant files first, understand the issue, make necessary changes, and then run any tests to verify your fixes work. Always explain your reasoning as you work through problems."

The system prompt would also explain what each function does and when to use it, although the function schemas themselves provide much of this information. The key is to give the model enough context to make intelligent decisions about which tools to use and in what order.

Security Considerations {#security}

Why Security Matters for AI Agents

When you give an AI model the ability to read, write, and execute files, you're essentially giving it programmatic access to your system. While LLMs like Gemini are generally helpful and follow instructions, they can also be manipulated through carefully crafted prompts or might make mistakes in their reasoning.

Consider what could happen without proper security:

- A malicious prompt could trick the agent into reading sensitive files like password databases
- The agent could accidentally overwrite important system files
- Executed code could spawn processes that persist after the agent finishes
- The agent could be used to exfiltrate data from your system

By implementing security boundaries, we create a sandbox where the agent can work freely while being unable to cause harm to the broader system.

The Working Directory Restriction

The cornerstone of our security model is restricting all operations to a designated working directory. This is implemented through path validation in every tool function.

The validation process works like this: when the agent requests an operation on a path like `"../../../etc/passwd"`, we first resolve it to an absolute path. The `os.path.abspath()` function follows all symbolic links and resolves relative references. We then check if this absolute path starts with the working directory path using `os.path.commonpath()`.

If the resolved path is outside our working directory, we reject the operation entirely and return an error message. The model sees this error and (hopefully) understands it needs to work within the allowed directory.

This approach protects against several types of attacks:

- Path traversal attacks using `..` to escape the directory
- Symbolic link attacks that point outside the working directory
- Absolute path specifications that bypass relative path restrictions

File Type Restrictions

For the `run_python_file` function, we add an additional check that the file ends with `.py`. This prevents execution of arbitrary executables or scripts in other languages that might have different security models.

While this check is simple, it's effective. Even if an attacker somehow got a malicious binary into the working directory, they couldn't execute it through our agent.

Execution Timeouts

The 30-second timeout on code execution serves multiple purposes. First, it prevents denial of service if the agent accidentally creates infinite loops. Second, it limits the potential damage from resource-intensive operations. Third, it keeps API costs manageable by ensuring we don't wait forever for results.

When a timeout occurs, the agent sees an error message and can try a different approach. This is preferable to hanging indefinitely or consuming excessive resources.

Content Limits

The 10,000 character limit on file reading prevents token budget overruns but also provides a mild security benefit. It means the agent can't dump the entire contents of a massive file into the API, which could potentially leak large amounts of data if the agent were compromised.

Defense in Depth

Notice how we implement multiple layers of security rather than relying on any single check. Even if one security measure failed, others would still provide protection. This "defense in depth" approach is a best practice in security engineering.

Testing and Debugging {#testing}

The Calculator Test Project

The calculator directory serves as our test case for the agent. This isn't a contrived example; it's a real multi-file Python project with a package structure, unit tests, and deliberate bugs for the agent to find and fix.

The calculator implements a simple postfix (Reverse Polish Notation) calculator that can handle basic arithmetic operations. The package structure includes a main entry point, a calculator module with the core logic, a rendering module for output, and a comprehensive test suite.

Running the Tests

You can run the tests directly without the agent using: `cd calculator && python -m unittest tests.py`. This shows you what the agent should be able to achieve. The test suite includes nine tests covering basic operations, complex expressions, and error handling.

When you ask the agent to "run the tests and fix any failures," it should:

1. Use `get_files_info` to explore the project structure
2. Read the test file to understand what's being tested
3. Run the tests using `run_python_file`
4. Read the relevant source files to understand the implementation
5. Identify the bug causing test failures
6. Modify the code to fix the bug
7. Run the tests again to verify the fix works

Verbose Mode for Debugging

The `--verbose` flag is your best friend when debugging. It shows you:

- The exact prompt being sent to the API
- Token counts for each request
- Function calls the model is making
- Results of those function calls
- The model's reasoning at each step

When something goes wrong, verbose output helps you understand where in the process things broke down. Did the model call the wrong function? Did it pass incorrect arguments? Did it misinterpret the error message?

Common Issues and Solutions

Issue: "ValueError: Failed to parse the parameter function_call_part"
This error occurred in earlier versions of the project and led to the decision to use manual function calling. The issue arises when Gemini's automatic function calling returns function calls in a format that's difficult to parse reliably. By switching to manual detection and parsing, we gain more control and can handle edge cases better.

Issue: Function calls timeout
If code execution times out, check for infinite loops or very slow operations. The agent might have introduced a bug that causes the code to hang. Use verbose mode to see what code was being executed.

Issue: Path access denied
This usually means the agent is trying to access files outside the working directory. Check the exact path being used and verify it's relative to the working directory. The security system is working as intended by blocking this access.

Issue: Agent makes wrong decisions
Sometimes the model will make suboptimal choices, like reading all files before running tests when it should run tests first to see what fails. This is where prompt engineering becomes important. You can guide the agent by being more specific in your initial prompt: "First run the tests to see what fails, then investigate the failing tests."

Advanced Topics and Extensions {#advanced}

Extending the Agent with New Tools

Adding new tools to the agent follows a straightforward pattern. First, define the function schema describing the new tool. Second, implement the Python function with proper security checks and error handling. Third, add the function to the `function_map` in `call_function()`. Finally, update the schema list passed to the model.

For example, you might add a tool to search for text within files, or a tool to run command-line programs other than Python scripts. Each addition makes the agent more capable but also requires careful thought about security implications.

Token Usage Optimization

As you use the agent more, you'll notice that conversation history can grow quite long, consuming more tokens with each iteration. Strategies to optimize token usage include:

- Summarizing older messages in the conversation history
- Removing function call details after they're no longer relevant
- Using more efficient function designs that return only necessary information
- Setting more aggressive character limits on file reading for large codebases

Improving Error Recovery

The current agent can handle errors by seeing error messages and adjusting, but you could make this more robust. For example, you might track repeated failures and suggest to the model that it try a fundamentally different approach. Or you could implement a "reflection" step where after several iterations, the agent summarizes what it's learned and plans its next steps explicitly.

Multi-Model Architectures

An interesting extension would be using different models for different tasks. You might use a fast, cheap model for simple operations like deciding which files to read, and a more powerful model for complex reasoning about bugs. This would optimize both cost and latency while maintaining high quality where it matters.

Human-in-the-Loop

For production use, you'd likely want to add approval steps before the agent executes certain operations. For example, before overwriting a file or running code, the agent could show you what it plans to do and wait for confirmation. This maintains safety while still providing the benefits of automation.

Expanding Beyond Python

While this agent focuses on Python, the same principles apply to other languages. You'd need to implement appropriate execution functions for each language (with their own security considerations) and train the model to understand the conventions of those languages through your system prompt and examples.

Conclusion

Building this AI agent from scratch provides deep insights into how modern AI-powered development tools actually work. You've learned about the fundamental concepts of agents versus chatbots, implemented function calling to extend an LLM's capabilities, designed security boundaries to protect your system, and created an iterative loop that allows the AI to take multiple passes at solving problems.

The principles you've learned here extend far beyond this specific project. Function calling is a general technique applicable to any task where you want to augment an LLM with external capabilities. The agentic loop pattern works for any iterative problem-solving scenario. And the security considerations apply anytime you're building systems that interact with the real world based on AI decisions.

As you continue exploring, consider how you might apply these concepts to other domains. Could you build an agent that interacts with databases? One that controls web browsers? One that manages cloud infrastructure? The possibilities are vast, and you now have the foundational knowledge to explore them.