Building AI Agents

MGMT 675: Generative AI for Finance

Kerry Back

Chatbot vs Agent

Chatbot

- Receives prompts
- Generates text responses
- That's it—just conversation
- Cannot take actions
- Cannot access external data

Agent

- Receives prompts
- Can generate text **or** request actions
- Has access to **tools**
- Can query databases, run code, search web
- Takes actions to accomplish goals

An agent is a chatbot with tools

What Are Tools?

Tools are functions the agent can call to interact with the outside world.

Example Tools

- Execute SQL queries
- Run Python code
- Read/write files
- Search the web
- Send emails
- Call APIs

How Tools Work

1. LLM decides to use a tool
2. Returns tool name + parameters
3. Agent executes the tool
4. Result sent back to LLM
5. LLM continues reasoning

Foundation: Calling an LLM from Code

The API: Talking to an LLM

An **API** (Application Programming Interface) lets your code communicate with an LLM service over the internet.

How It Works

1. Your code sends a request
2. Request includes your prompt
3. LLM processes the prompt
4. API returns the response
5. Your code uses the result

What You Need

- API endpoint (URL)
- API key (authentication)
- Model name to use
- Your prompt/messages

OpenRouter: One API, Many Models

What is OpenRouter?

- Unified API for 100+ models
- OpenAI, Anthropic, Google, Meta, etc.
- Single API key for all models
- Some models are free!

`openrouter.ai`

Free Models on Hugging Face

- Hugging Face hosts open models
- OpenRouter provides free access
- Examples:
  - `mistralai/mistral-7b-instruct:free`
  - `meta-llama/llama-3-8b-instruct:free`
  - `google/gemma-7b-it:free`

A Single API Call

Basic Structure

```
import requests

response = requests.post(
    "https://openrouter.ai/api/v1/chat/completions",
    headers={"Authorization": f"Bearer {API_KEY}"},
    json={
        "model": "mistralai/mistral-7b-instruct:free",
        "messages": [{"role": "user", "content": prompt}]
    }
)
answer = response.json()["choices"][0]["message"]["content"]
```

Message Format and Conversation History

Messages Are a List of Dictionaries

```
messages = [
    {"role": "system", "content": "You are a helpful
        finance tutor..."},
    {"role": "user", "content": "What is a P/E ratio?"},
    {"role": "assistant", "content": "A P/E ratio is..."},
    {"role": "user", "content": "How do I interpret it?"}
]
```

- Each API call is independent—LLM has no memory
- You must send the **entire conversation history** each time
- The **system prompt** (role `system`) defines the agent's behavior

The Agent Loop

The Agent Loop

The agent runs in a loop: LLM thinks → tool executes → result returns → LLM thinks again.

$$\begin{tikzcd}\boxed{\textbf{LLM}} \arrow[r, shift left, "\textbf{Tool Call}"] & \boxed{\textbf{Tool}} \arrow[l, shift left, "\textbf{Result}"]\end{tikzcd}$$

- LLM decides next action
- Returns tool name + parameters
- Agent executes the tool

- Result added to message history
- LLM sees result, reasons again
- Loop continues until task complete

Agent Loop Pseudocode

The Agent's Decision Loop

```
while not done:
    response = call_llm(messages, system_prompt, tools)

    if response.has_tool_call:
        result = execute_tool(response.tool_call)
        messages.append(tool_result)
    elif response.needs_user_input:
        answer = ask_user(response.question)
        messages.append(user_answer)
    elif response.is_final:
        done = True
        return response.content
```

The agent is part LLM intelligence, part traditional programming

Building an Agent Step by Step

The Key Pieces

To build an agent, you need five components:

1. **Define tools**: What actions can the agent take?
2. **Write a system prompt**: Describe available tools, schema, and rules
3. **Implement the agent loop**: Send messages, parse tool calls, execute, repeat
4. **Handle tool execution**: Safely run SQL, Python, etc.
5. **Manage context**: Keep message history, handle token limits

Or use an agent framework: LangChain, CrewAI, Claude Code SDK

Example: Database Analytics Agent

User request: “Analyze quarterly revenue trends for our top 5 customers and create a summary report.”

What the Agent Needs to Do

1. Write SQL to identify top 5 customers by revenue
2. Execute the SQL query against the database
3. Write SQL to get quarterly revenue for those customers
4. Execute that query
5. Write Python to analyze trends and create visualizations
6. Execute the Python code
7. Analyze the results
8. Generate a written report for the user

Step 1: User Prompt Arrives

Messages Sent to LLM

Role	Content
system	You are a data analyst with SQL and Python tools...
user	Analyze quarterly revenue trends for top 5 customers...

- System prompt defines capabilities
- Lists available tools
- Specifies database schema
- Sets response format

LLM Decides

"I need to first find the top 5 customers. I'll write a SQL query."

Step 2: LLM Requests SQL Tool

LLM Response (Not Text—A Tool Call)

```
{
  "tool": "execute_sql",
  "parameters": {
    "query": "SELECT customer_id, SUM(amount) as total
              FROM sales GROUP BY customer_id
              ORDER BY total DESC LIMIT 5"
  }
}
```

- LLM doesn't return text to user yet
- Instead, requests a tool execution
- Agent code intercepts this and runs the SQL
- Database returns results

Step 3: Tool Result Returns to LLM

Messages Now Include Tool Result

```
[
  {"role": "system", "content": "You are a data analyst..."},
  {"role": "user", "content": "Analyze quarterly revenue..."},
  {"role": "assistant", "tool_call": "execute_sql(...)"},
  {"role": "tool", "content": "customer_id,total\n
                               ACME,450000\nGlobex,380000\n..."}
]
```

The LLM sees the full history including tool results

Step 4: LLM Continues Reasoning

With the top 5 customers identified, the LLM decides what to do next.

LLM Thinks

"I have the top 5 customers: ACME, Globex, Initech, Umbrella, Wayne. Now I need quarterly data for each."

Next Tool Call

`execute_sql`: Get quarterly revenue for these 5 customers...

The loop continues: tool call → result → reasoning → next action

The Complete Message History

What the LLM Sees at Report Time

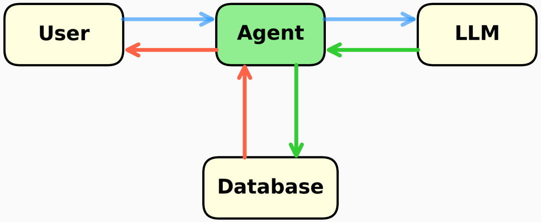
#	Message
1	system: You are a data analyst...
2	user: Analyze quarterly revenue...
3	assistant: [tool call: SQL for top 5]
4	tool: [results: ACME, Globex...]
5	assistant: [tool call: SQL for quarterly data]
6	tool: [results: Q1, Q2, Q3...]
7	assistant: [tool call: Python analysis]
8	tool: [output: stats, saved trends.png]
9	assistant: Here is my analysis... (final report)

Example: Rice Data Portal

Rice Data Portal: A Database Agent

- Prompt + System Prompt → LLM
- LLM → SQL code or a question for the user
- Agent → SQL code to database or → question to the user
- Eventually, SQL code → database
- Database → data or error message
- Error message → LLM
- Eventually data → user

Visit data-portal.rice-business.org
Get access token, Log in, Ask for data



How the Data Portal is Built

- System prompt provides the database schema and instructions to the LLM
- Agent logic is hard-coded as "if . . . then . . ." (**not AI**)
- The LLM writes SQL; the agent executes it
- If the database returns an error, the agent sends it back to the LLM to try again
- If the LLM needs clarification, the agent displays the question to the user

Architecture

Component	Implementation
LLM	OpenRouter API (various models)
Tool	SQL query execution against PostgreSQL
Agent Loop	Python if/else logic
UI	Streamlit web interface

User Interface and Deployment

Adding a User Interface

An agent doesn't need a web UI—it can run in a terminal or as a script. But a UI makes it accessible to non-technical users.

Streamlit

- Python library for web apps
- No HTML/CSS/JavaScript needed
- Built-in chat UI components
- Great for data apps and dashboards
- `streamlit.io`

Gradio

- Python library for ML demos
- Even simpler chat interface
- Built-in sharing via public links
- Popular in the ML community
- `gradio.app`

Both let you build a chat UI in a few lines of Python—no web development skills required

With or Without a UI

Without UI (Terminal)

- Agent runs in a Python script
- User types prompts at the command line
- Results printed to the terminal
- Simple, fast to develop
- Good for personal tools and scripts

With Streamlit/Gradio UI

- Agent wrapped in a web interface
- Chat bubbles, file uploads, charts
- Shareable via URL
- Accessible to non-technical users
- Good for team tools and demos

The **agent logic is the same** either way. Streamlit and Gradio only handle how the user interacts with the agent—they don't change how the agent works.

From Idea to Deployed Agent

The complete workflow to build and deploy an agent—all handled by Claude Code with natural language requests.

1. **Build the agent**: Define tools, system prompt, and agent loop
2. **Add a UI** (optional): Wrap with Streamlit or Gradio
3. **Create a GitHub repository**: Version control and collaboration
4. **Deploy**: Push to a cloud platform (e.g., Koyeb) for a public URL
5. **Iterate**: Push updates that auto-deploy

Ask Claude Code to do each step—no commands to memorize

Advanced: Orchestration

The Orchestration Layer

The agent's control logic coordinates everything: which LLM to call, which system prompt to use, and what to do with each response.

Different Tasks, Different Prompts

- SQL generation → database schema prompt
- Python analysis → data science prompt
- Report writing → communication prompt
- Each task gets specialized instructions

Different Tasks, Different LLMs

- Simple classification → fast, cheap model
- Complex reasoning → powerful model
- Code generation → code-specialized model
- Cost and speed optimization

Claude Code: A General-Purpose Agent

- Claude Code is itself an agent
- LLM: Claude Opus or Sonnet
- Built-in tools: file I/O, bash, code execution, web search
- Agent logic: built into Claude Code
- **Skills** customize it for specific tasks

Claude Code Architecture

Component	**Provider**
LLM	Claude Opus/Sonnet
Agent Logic	Claude Code app
Base Tools	Claude Code app
System Prompt	**Skill**
Custom Tools	**Skill scripts**

Summary

Key Concepts

- Agent = Chatbot + Tools
- LLM decides which tools to use
- Tool results feed back to LLM
- Agent loop orchestrates the flow
- Full message history provides context

Building Blocks

- API call to an LLM (OpenRouter)
- System prompt with tool definitions
- Agent loop (while not done)
- Tool execution (SQL, Python, etc.)
- Optional UI (Streamlit / Gradio)

Agents extend LLMs from conversation to action