AI Agents: From First Principles

A Practical Guide to Building AI Agent Systems in Python

Saad Khalid

February 14, 2026

Contents

Chapter 1

Introduction

This is a project about building AI agents from scratch. Not the kind where you `pip install` a framework and hope for the best. The kind where you write every piece yourself and understand what's actually happening.

Each file is self-contained. You can read any one of them without reading the others. They're numbered in the order you should probably learn them, but there's no shared library or base class tying them together.

1.1 Setup

```
python -m venv venv && source venv/bin/activate
pip install -r requirements.txt
```

Create a `.env` file:

```
OPENAI_API_KEY=sk-...
OPENROUTER_API_KEY=sk-or-...
```

Then run any script directly: `python 0_hello.py`.

1.2 The files

```
0_hello.py                                  API connectivity test
1_rag/
  1_rag.py                                Benchmark orchestrator (runs 1.1–1.3)
  1.1_in_context_qa.py                      In-context evaluation
  1.2_naive_rag_with_embeddings.py          Naive RAG with embeddings
  1.3_rag_with_query_rewording.py           RAG with query rewriting
  1.4_large_corpus_rag.ipynb                RAG at scale (10K+ docs)
  1.5_...query_rewriting.ipynb              Query rewriting at scale
  1.6_reranking.ipynb                     Two-stage retrieval with re-ranking
  1.7_hyde.ipynb                          Hypothetical document embeddings
  1.8_agentic_rag.ipynb                     Agentic RAG with feedback loop
2_tool_calls/
  2.1_single_tool_call.ipynb                Tool call lifecycle
  2.2_parallel_tool_calls.ipynb             Parallel tool execution
  2.3_chained_tool_calls.ipynb              Multi-turn chained tool calls
```

Chapter 2

Fundamentals

2.1 Hello World

Before you build anything complicated, you should verify the plumbing works. This script sends one message to an LLM and prints the response. If this doesn't work, nothing else will either.

```
.env (API key) ──> 0_hello.py ──> OpenAI API ──> "Hello, world!"
```

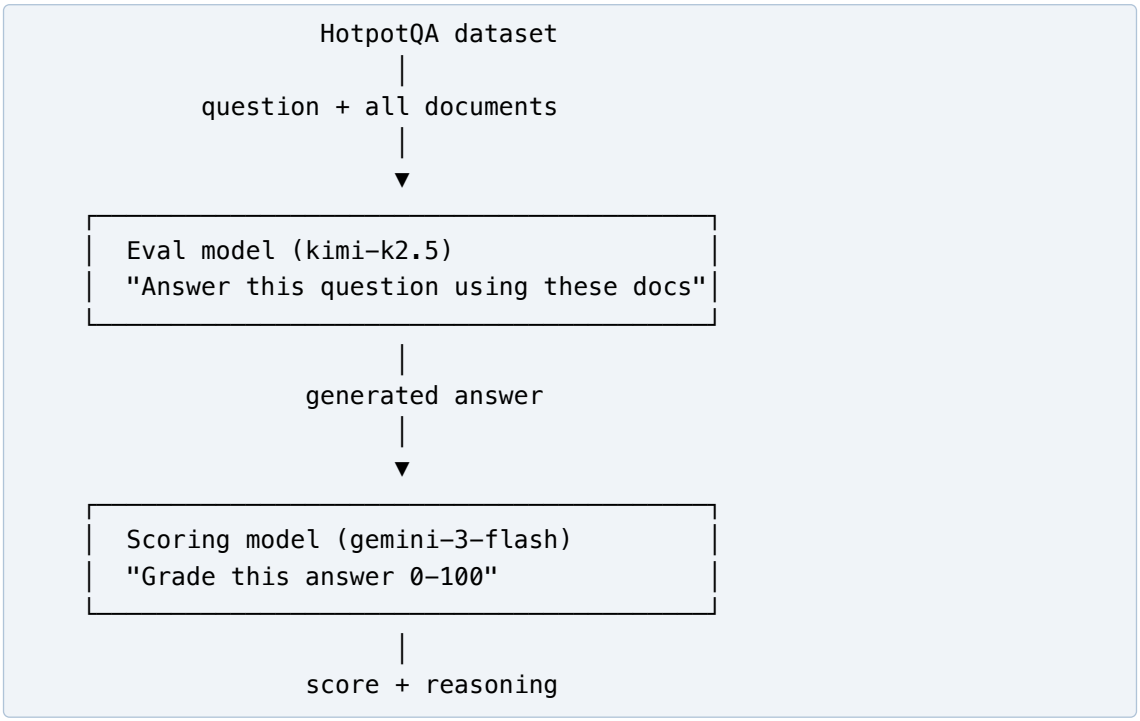
That's it. Twenty-eight lines. The interesting stuff starts next.

2.2 Measuring Quality

Here's the first real question: how do you know if your model is any good?

You can't just eyeball a few answers. You need a benchmark — a dataset with known questions and answers, and a systematic way to score responses. This script builds that evaluation framework from scratch, and every other script in the project reuses it.

The approach is called LLM-as-judge. A stronger model grades a weaker model's answers on four criteria: correctness, completeness, faithfulness, and clarity. Each criterion is worth 25 points, for a total of 0–100.



The dataset is HotpotQA — multi-hop questions that require combining facts from multiple documents. We evaluate 100 questions in parallel using 8 threads.

The key detail: we give the model *all* the documents. No retrieval, no searching. This is the control group. It tells us the ceiling — how well the model does when it has everything it needs.

Run it:

```
python 1_rag/1.1_in_context_qa.py
python 1_rag/1.1_in_context_qa.py --eval-model openai/gpt-4o --num-examples 50
```

Chapter 3

RAG

The interesting question isn't whether LLMs can answer questions from documents. They can. The interesting question is what happens when you stop handing them the documents and make them go find the right ones.

That's retrieval-augmented generation: embed documents into vectors, search for the ones that match the question, then feed only those to the model. The rest of this chapter is about what happens when you try it, what breaks, and how to fix it.

3.1 The Orchestrator

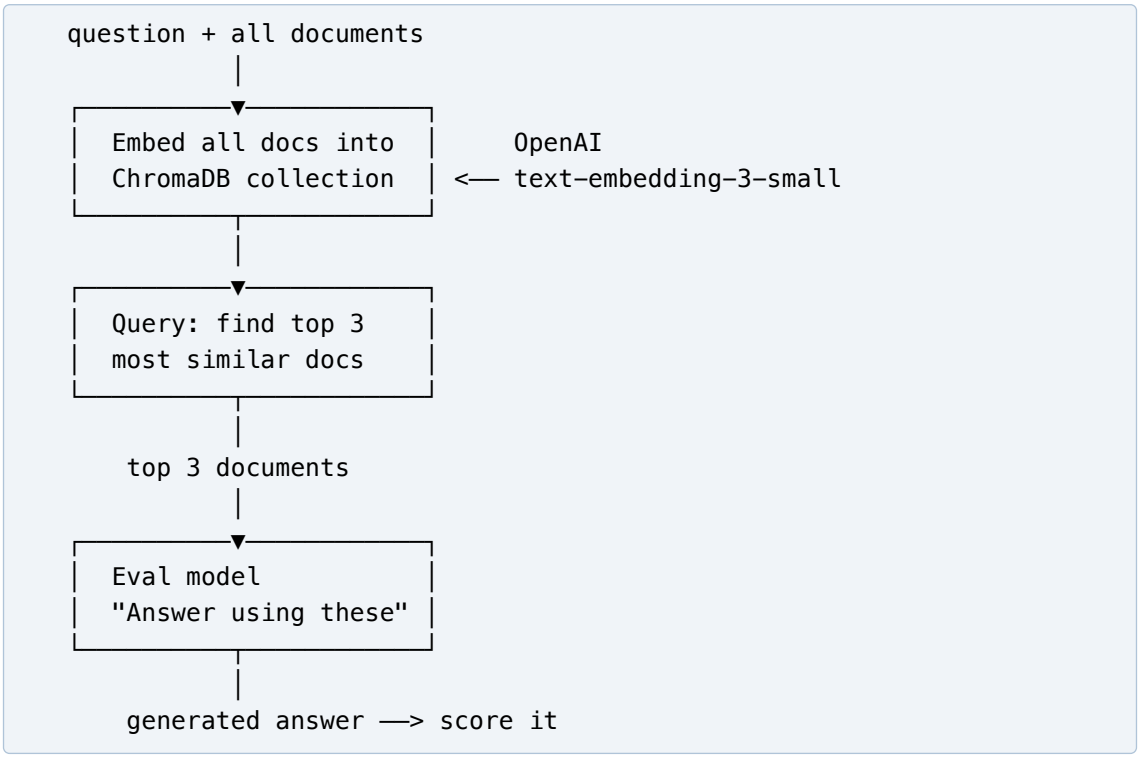
Before diving into individual scripts, it's worth mentioning the benchmark runner. It takes model parameters once and runs all the RAG approaches back-to-back:

```
python 1_rag/1_rag.py --eval-model openai/gpt-4o --num-examples 20
```

This produces a comparison table at the end so you can see how the approaches stack up against each other with identical settings.

3.2 Naive RAG

The simplest possible RAG pipeline: embed the documents, embed the question, find the top 3 matches, and feed them to the model.



The crucial difference from 1.1: the model no longer sees all the documents. It sees *only* what the retrieval found. If the embedding search misses a relevant document, the model can't use it.

Each question gets its own ephemeral ChromaDB collection with only its ~10 documents. This is "naive" because in real RAG systems you don't rebuild the index per question — you search a shared corpus. But for measuring the impact of retrieval vs. in-context, this controlled setup is useful.

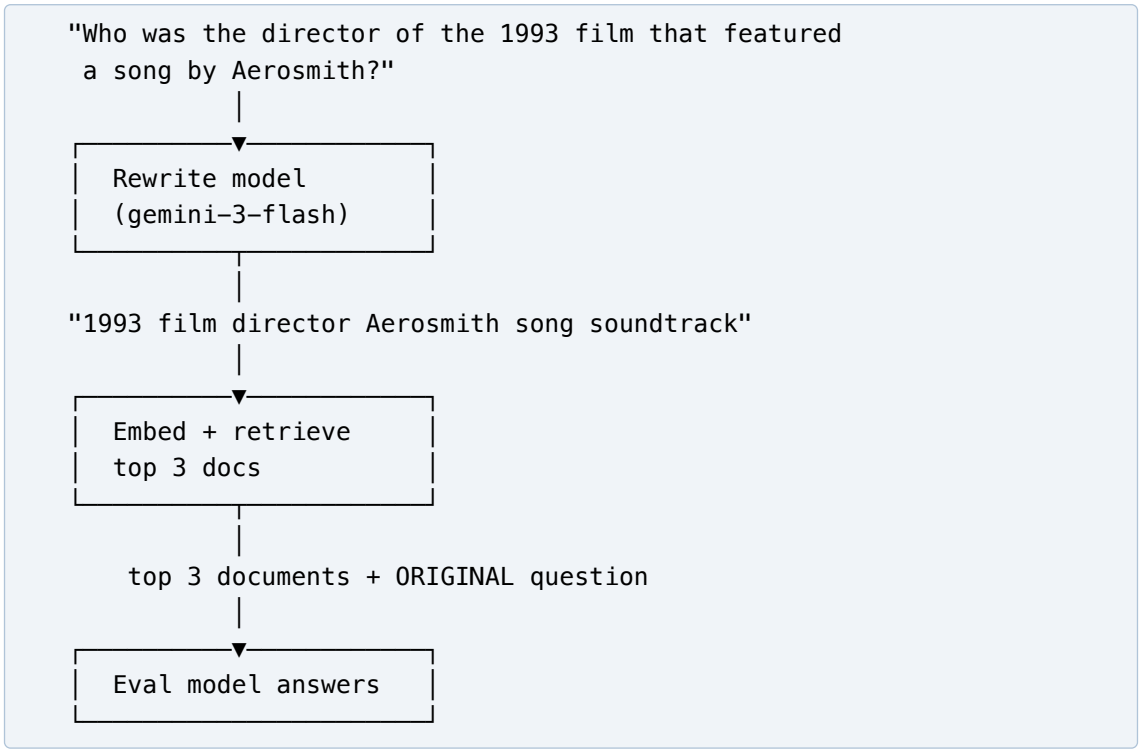
Run it:

```
python 1_rag/1.2_naive_rag_with_embeddings.py
python 1_rag/1.2_naive_rag_with_embeddings.py --eval-model openai/gpt-4o
```

3.3 Query Rewriting

Here's an insight that's easy to miss: the question the user asks isn't always the best query for embedding search.

User questions are conversational. Embedding search works better with keyword-rich, declarative statements. So before searching, we ask an LLM to rewrite the question into a search-optimized query.



Notice the subtle but important detail: we search with the *rewritten* query but answer with the *original* question. The rewrite is optimized for retrieval, not for comprehension.

Everything else is identical to 1.2. Same scoring, same dataset, same parallelism. The only change is one extra LLM call per question to rewrite the query before searching.

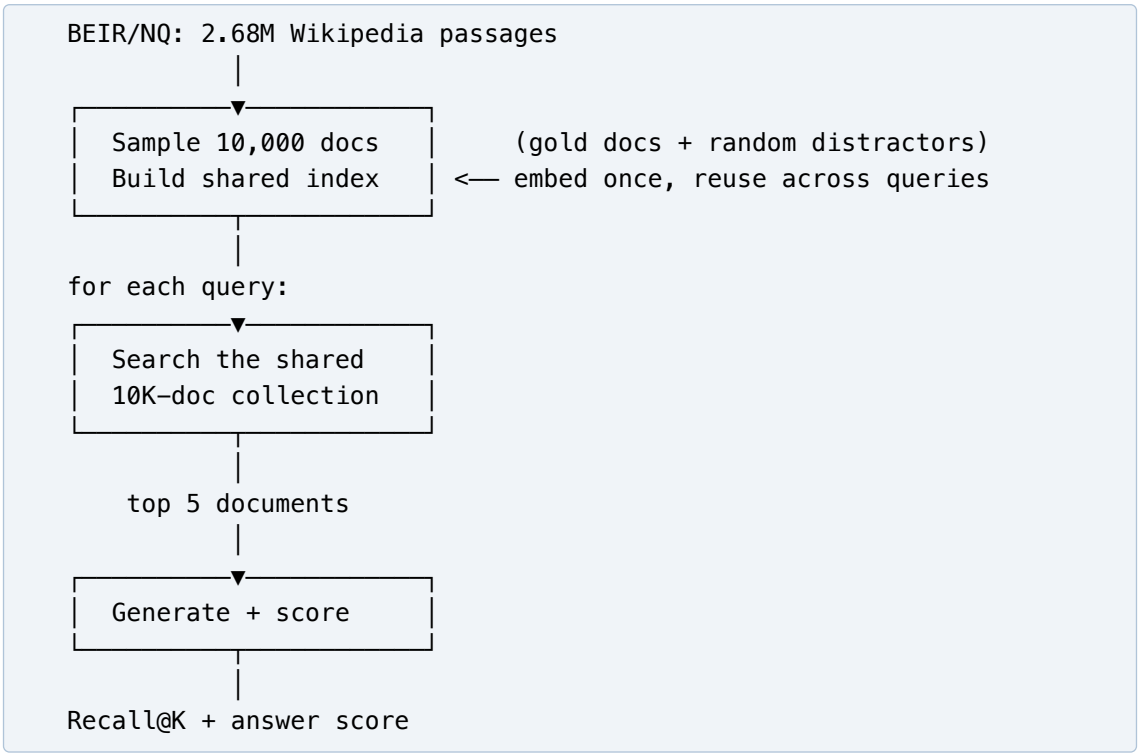
Run it:

```
python 1_rag/1.3_rag_with_query_rewording.py
python 1_rag/1.3_rag_with_query_rewording.py --rewrite-model openai/gpt-4o-mini
```

3.4 Scaling Up

Everything so far has a problem: the haystack is tiny. Each question in HotpotQA comes with about 10 documents. Finding the right one out of 10 isn't much of a test.

Real RAG systems search thousands or millions of documents. This notebook uses the BEIR Natural Questions dataset — 2.68 million Wikipedia passages and 3,452 questions originally from Google Search. We build one shared ChromaDB collection with 10,000+ documents and make every query search the same big corpus.



The key architectural difference: one persistent collection shared across all queries, instead of throwaway per-question collections. The collection is cached on disk (ChromaDB PersistentClient) so you don't re-embed on every run.

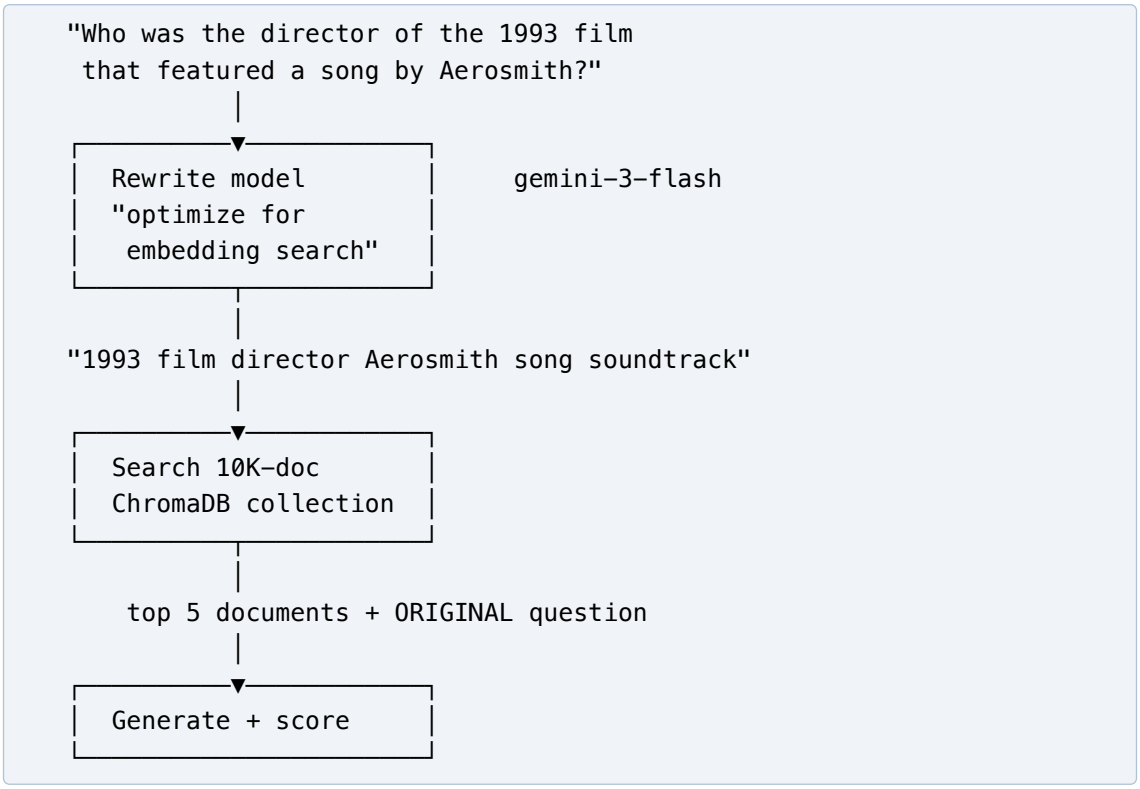
We also introduce a new metric: **Recall@K**. For each query, the BEIR dataset tells us which documents are actually relevant (ground-truth labels). Recall@K measures what fraction of those we found in our top-K retrieval. If the gold document is ranked 6th and K=5, recall is 0.

This is where you start to see the real challenge. When the haystack grows from 10 to 10,000 documents, retrieval quality drops and it drags answer quality down with it.

3.5 Query Rewriting at Scale

1.3 showed that query rewriting helps on small datasets. 1.5 tests whether the same trick works when the haystack is 10,000 documents instead of 10.

The approach is identical to 1.4, with one extra step: before searching, we rewrite the question into a search-optimized query. The intuition is the same as 1.3, but now we're testing it where it matters — at scale, where retrieval is actually hard.



The core rewriting logic:

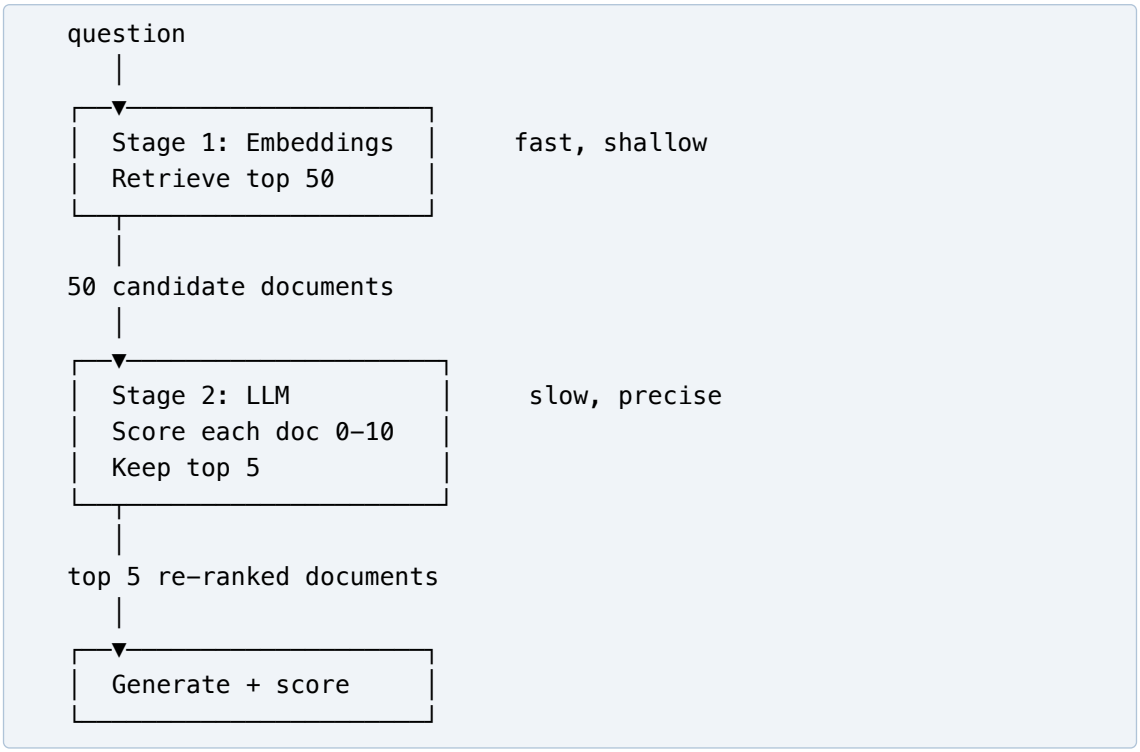
```python
def rewrite_query(question: str) -> str:
    response = client.chat.completions.create(
        model=REWRITE_MODEL,
        messages=[
            {"role": "system", "content": (
                "You are a search query optimizer. Rewrite the "
                "question as a keyword-rich, declarative query "
                "optimized for semantic similarity search. "
                "Remove filler words. Keep it concise. "
                "Output ONLY the rewritten query."
            )},
            {"role": "user", "content": question},
        ],
        max_tokens=100,
    )
    return response.choices[0].message.content.strip()
```

The subtle detail: we search with the *rewritten* query but answer with the *original* question. The rewrite is optimized for retrieval, not comprehension.

3.6 Re-ranking

Embedding similarity is fast but shallow. It matches on surface-level vector distance, which means a document about "apple pie recipes" might rank higher than one about "Apple Inc. revenue" for a query about Apple's stock price, just because the word "apple" is prominent.

Re-ranking fixes this by adding a second stage: retrieve broadly with embeddings, then have an LLM actually *read* each candidate and score its relevance.



The key trade-off: INITIAL_K=50 gives you broad coverage (you're unlikely to miss the right document), while RERANK_K=5 gives the answer model a focused, high-quality context. You're spending one LLM call per query to read 50 document snippets, but that's much cheaper than embedding-searching the entire corpus at higher precision.

The re-ranking function asks the model to score each document's relevance:

```python
def rerank_documents(question, doc_ids, doc_texts, top_k):
    doc_list = "\n\n".join(
        f"[Doc {i+1}] {text[:500]}"
        for i, text in enumerate(doc_texts)
    )
    response = client.chat.completions.create(
        model=RERANK_MODEL,
        messages=[{
            "role": "system",
            "content": (
                "Score each document's relevance to the "
                "question on a 0-10 scale. Respond with "
                "ONLY a JSON array: "
                '[{"doc": 1, "score": 8}, ...]'
            )
        }, {
            "role": "user",
            "content": f"Question: {question}\n\n"
                       f"Documents:\n{doc_list}"
        }],
        max_tokens=1000,
    )
    # Parse scores, sort descending, keep top_k
    scores = json.loads(response.choices[0].message.content)
    scored = sorted(
        [(s["score"], s["doc"] - 1) for s in scores],
        reverse=True,
    )
    top_indices = [idx for _, idx in scored[:top_k]]
    return (
        [doc_ids[i] for i in top_indices],
        [doc_texts[i] for i in top_indices],
    )
```

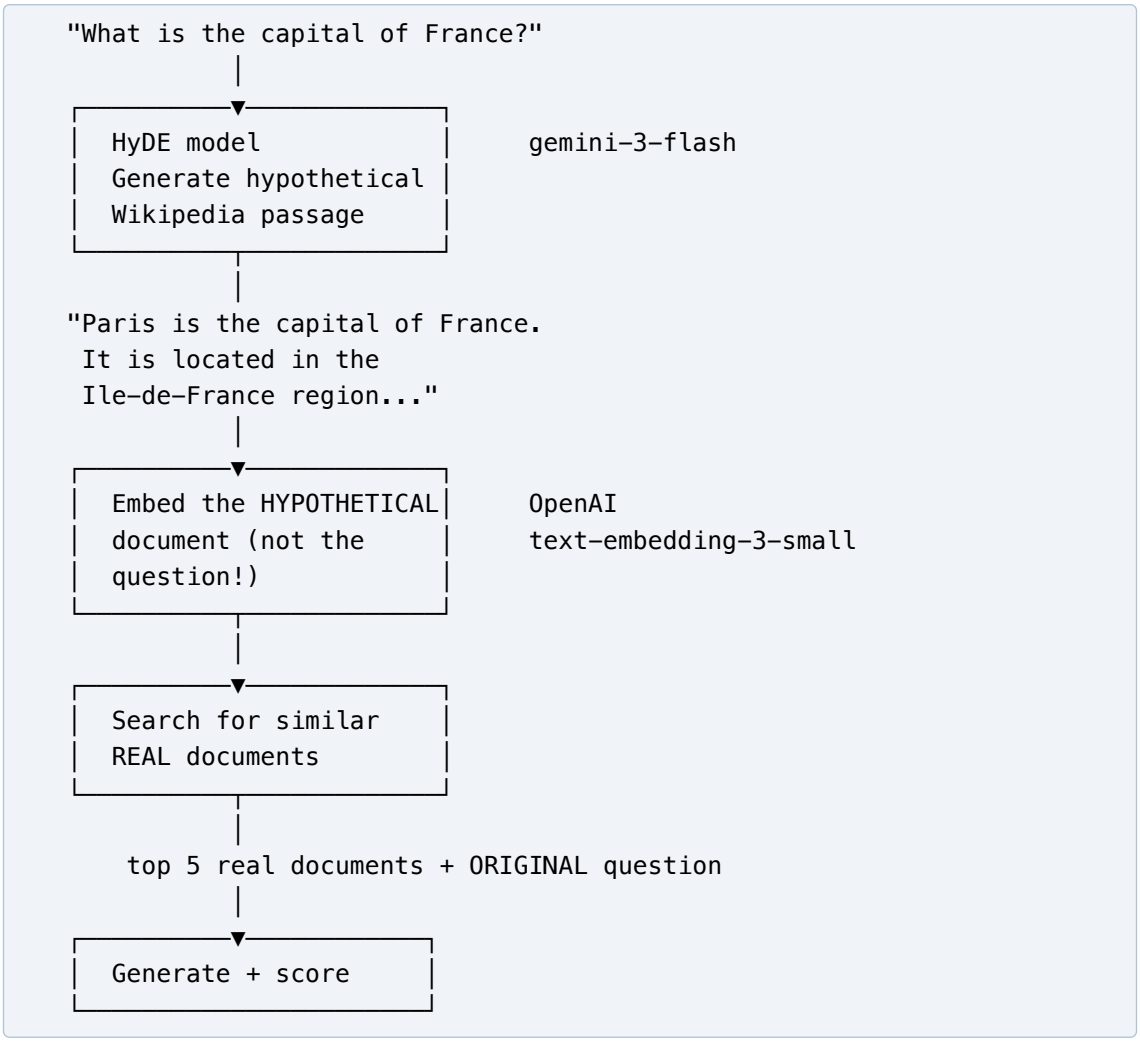
The evaluation tracks both `recall_before_rerank` (what embedding search found in the top 5) and `recall_at_k` (what's left after re-ranking). This tells you whether the LLM is helping or hurting — it's possible for re-ranking to *drop* a relevant document if the model misjudges its relevance.

3.7 HyDE — Hypothetical Document Embeddings

HyDE flips the retrieval problem. Instead of embedding the question and searching for similar documents, you generate a *hypothetical answer* and search for documents similar to *that*.

Why does this work? Consider the embedding space. A question like "what is the capital of France?" and a Wikipedia passage containing "Paris is the capital of France" live in different neighborhoods — one is a question, the other is a statement. But a hypothetical answer ("The capital of France is Paris, located in the Ile-de-France region") is much closer to the real document in embedding space.

The hypothetical answer doesn't need to be correct. It just needs to *sound like* the kind of document that would contain the real answer.



The implementation has two key functions:

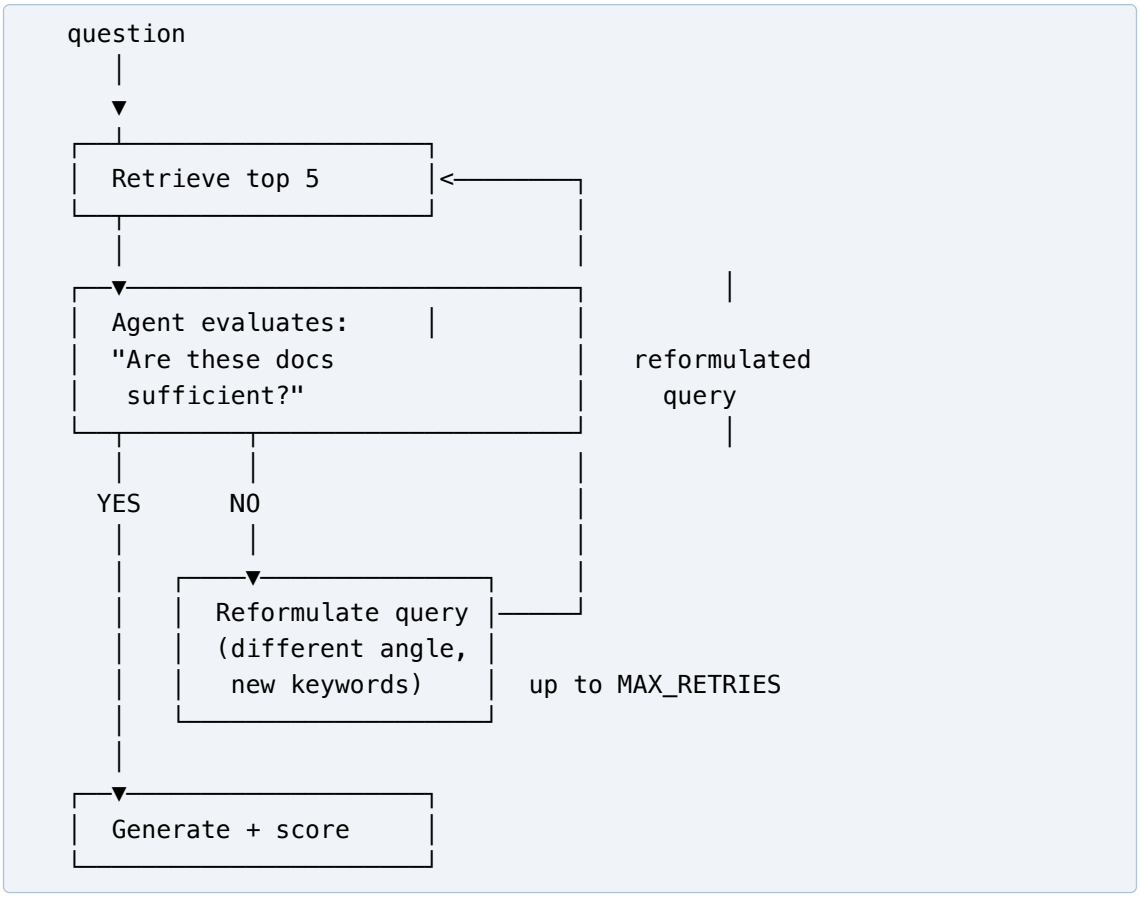
```python
def generate_hypothetical_document(question: str) -> str:
    """Generate a fake Wikipedia passage that would answer
    the question. Doesn't need to be correct -- just needs
    to sound like a real article about the topic."""
    response = client.chat.completions.create(
        model=HYDE_MODEL,
        messages=[{
            "role": "system",
            "content": "Write a short Wikipedia-style passage "
                       "that would answer the question. Keep "
                       "it under 150 words. Output ONLY the "
                       "passage."
        }, {"role": "user", "content": question}],
        max_tokens=200,
    )
    return response.choices[0].message.content.strip()

def hyde_retrieve(question, collection, top_k):
    hypo_doc = generate_hypothetical_document(question)
    # Embed the hypothetical doc, not the question
    hypo_embedding = openai_client.embeddings.create(
        model=EMBEDDING_MODEL, input=[hypo_doc]
    ).data[0].embedding
    # Search using the hypothetical doc's embedding
    results = collection.query(
        query_embeddings=[hypo_embedding],
        n_results=top_k,
    )
    return results["ids"][0], results["documents"][0], hypo_doc
```

Notice HyDE requires a direct OpenAI client for embedding (not through ChromaDB), since we're embedding arbitrary text rather than using ChromaDB's built-in query.

3.8 Agentic RAG

Every technique so far uses a fixed pipeline: retrieve once, answer once. If the retrieval misses, you're stuck.

Agentic RAG adds a feedback loop. After retrieving documents, an LLM evaluates whether they actually contain enough information to answer the question. If not, it reformulates the query and tries again. This is the first technique in our progression where the model makes its own decisions about the retrieval process.



The agent's evaluation step is the key innovation:

```python
def evaluate_retrieval(question, doc_texts):
    """Ask the LLM: do these docs contain enough
    info to answer the question?"""
    response = client.chat.completions.create(
        model=AGENT_MODEL,
        messages=[{
            "role": "system",
            "content": "Decide if the documents contain enough "
                       "information to answer the question. "
                       'Respond with JSON: {"sufficient": '
                       'true/false, "reformulated_query": '
                       '"new query if not sufficient"}'
        }, {
            "role": "user",
            "content": f"Question: {question}\n\n"
                       f"Documents:\n{doc_list}"
        }],
        max_tokens=200,
    )
    data = json.loads(response.choices[0].message.content)
    return data["sufficient"], data["reformulated_query"]
```

The retrieval loop merges results across attempts (deduplicating by document ID), so each retry *adds* to the pool of candidates rather than replacing them. After MAX_RETRIES=3 attempts, it answers with the best documents it has.

This trades latency for quality. If the first retrieval hits, you get an answer in one round. If it misses, you spend 2–3 extra LLM calls trying different angles. The evaluation tracks `retrieval_attempts` and `queries_tried` per example so you can see how often the agent actually retries.

3.9 What we learned

The progression from 1.1 to 1.8 tells a clear story about the retrieval problem and the different ways to attack it.

1.1 (all documents) is the ceiling. The model has everything it needs. If it can't answer, the problem is comprehension, not retrieval.

1.2 (naive RAG) shows that retrieval works surprisingly well when the haystack is small. With only 10 documents, even simple embedding search finds what you need most of the time.

1.3 (query rewriting) demonstrates that the question you ask isn't always the best search query. A quick rewrite step is cheap and often improves retrieval.

1.4 (large corpus) is where reality sets in. When the haystack grows to 10,000+ documents, naive top-K retrieval starts missing relevant content, and answer quality drops.

1.5 (query rewriting at scale) tests whether the trick from 1.3 still helps when the corpus is large. Same technique, harder problem.

1.6 (re-ranking) attacks the problem from the other end: retrieve broadly, then use an LLM to filter precisely. The cost is one extra LLM call per query to score 50 documents.

1.7 (HyDE) changes *what* gets embedded for the search. Instead of embedding the question, embed a hypothetical answer. This bridges the question-document gap in embedding

space.

1.8 (agentic RAG) gives the model autonomy over the retrieval process. Instead of a fixed pipeline, it decides whether to try again and how to reformulate. This is the most expensive approach but the most adaptive.

Each technique adds one idea. They can also be combined — you could rewrite the query (1.5), retrieve broadly (1.6 stage 1), re-rank (1.6 stage 2), and retry if the agent isn't satisfied (1.8). Production RAG systems often stack several of these together.

Chapter 4

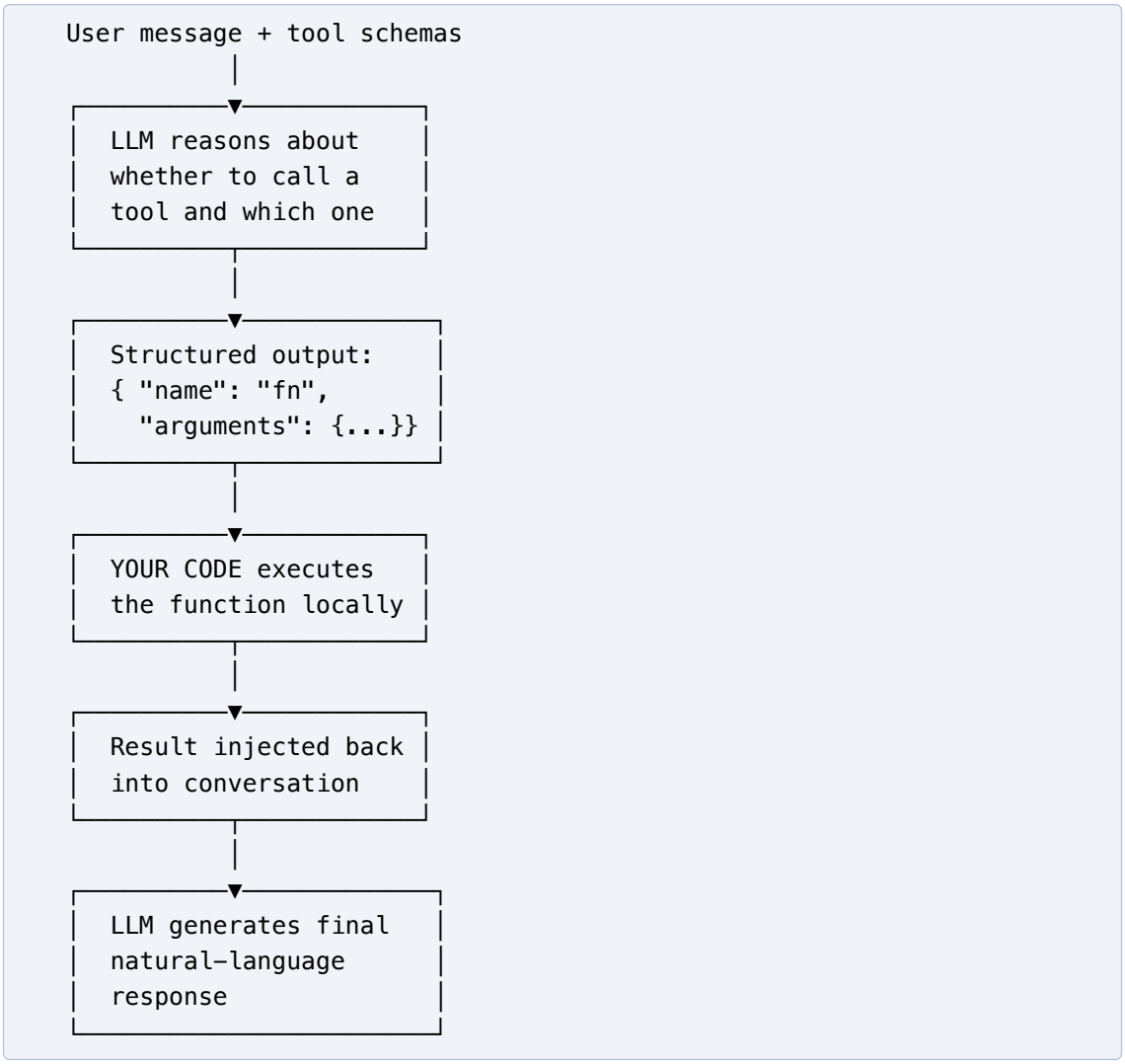
Tool Calling

Tool calling lets an LLM invoke external functions during a conversation — search the web, query a database, run code. Instead of generating text, the model outputs a structured request to call a specific function. You execute it and feed the result back. This loop is what turns a chatbot into an agent.

The key insight: the model never executes anything. It reads a JSON schema describing a function, decides when to call it, and emits structured arguments. Your code does the execution. This separation is what makes the pattern safe and composable.

4.1 The Lifecycle

Every tool call follows the same five-step lifecycle:



A tool is defined as a JSON schema with three parts: a name, a description (which the model reads to decide *when* to call it), and a parameters schema (which tells the model *how* to call it):

```
tools = [{
    "type": "function",
    "function": {
        "name": "get_weather",
        "description": "Get the current weather for a city.",
        "parameters": {
            "type": "object",
            "properties": {
                "city": {
                    "type": "string",
                    "description": "City name",
                },
            },
            "required": ["city"],
        },
    },
}]
```

The implementation lives in your code. The model never sees it:

```python
def get_weather(city: str) -> dict:
    return {"temp_f": 62, "conditions": "Foggy"}

available_functions = {"get_weather": get_weather}
```

The lifecycle then plays out in two API calls:

```python
# First call: model decides to use a tool
response = client.chat.completions.create(
    model=MODEL, messages=messages, tools=tools,
)
tool_call = response.choices[0].message.tool_calls[0]
# tool_call.function.name = "get_weather"
# tool_call.function.arguments = '{"city": "San Francisco"}'

# Execute locally
result = get_weather(**json.loads(tool_call.function.arguments))

# Second call: feed the result back
messages.append(response.choices[0].message)
messages.append({
    "role": "tool",
    "tool_call_id": tool_call.id,
    "content": json.dumps(result),
})
response = client.chat.completions.create(
    model=MODEL, messages=messages, tools=tools,
)
# response.choices[0].message.content = "The weather in San
#   Francisco is foggy with a temperature of 62 F..."
```

The `tool_call_id` links the result to the specific request. This is important when there are multiple tool calls — it tells the model which result corresponds to which call.

Here's the full conversation trace for a single tool call:

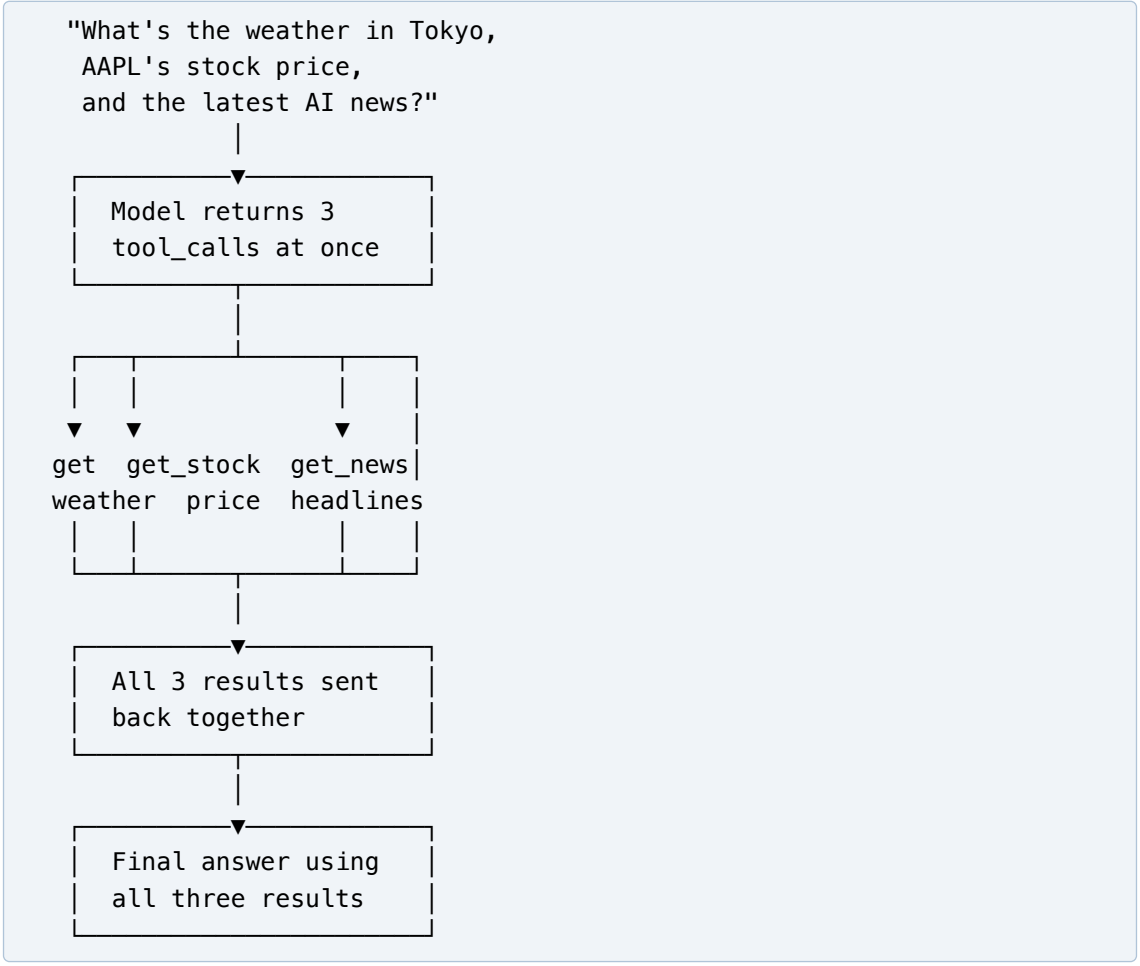
```
[0] role=user
    content: What's the weather in London?

[1] role=assistant
    tool_call: get_weather({"city":"London"})

[2] role=tool  tool_call_id=tool_get_weather_FP46U9...
    content: {"temp_f": 50, "conditions": "Rainy", "humidity": 85}

[3] role=assistant
    content: The weather in London is currently rainy with a
             temperature of 50 F and 85% humidity.
```

Four messages: user question, assistant tool request, tool result, assistant final answer. Two API round-trips. One local function execution in between.

When the model doesn't need a tool, it skips the whole mechanism and responds with text directly. The `tool_calls` field is `None`, and you get the answer in one round-trip.

4.2 Parallel Tool Calls

When a question requires multiple independent pieces of information, the model can request several tool calls in a single turn:



The model identifies which parts of the question are independent and batches them. You execute all calls concurrently:

```python
def execute_tool_calls_parallel(tool_calls):
    def execute_one(tc):
        fn = available_functions[tc.function.name]
        args = json.loads(tc.function.arguments)
        return {
            "role": "tool",
            "tool_call_id": tc.id,
            "content": json.dumps(fn(**args)),
        }

    with ThreadPoolExecutor(max_workers=len(tool_calls)) as ex:
        return list(ex.map(execute_one, tool_calls))
```

The speedup is significant. With three tools that each take 300ms:

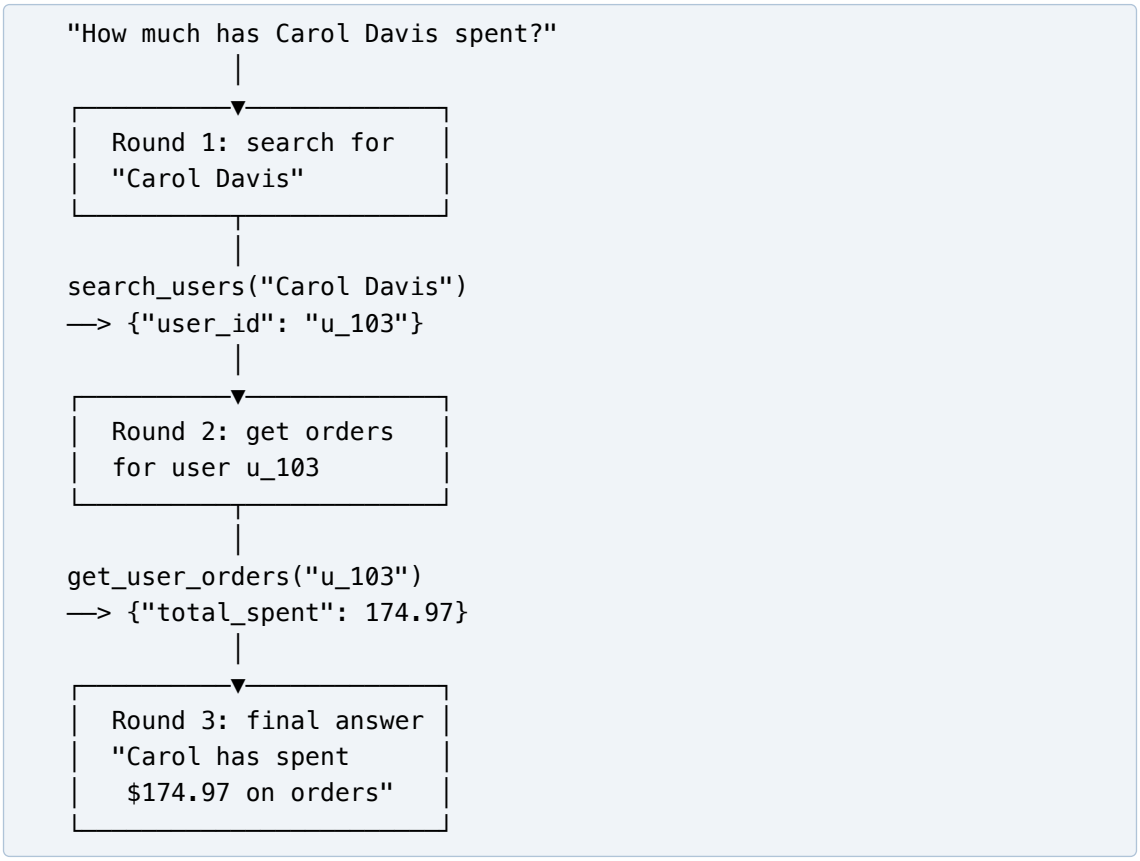
```
3 tools (each 300ms):
  Parallel:   0.305s
  Sequential: 0.916s
  Speedup:    3.0x
```

This is a latency optimization, not a capability one — you get the same answer either way. But in production, where tools hit real APIs with real latency, the difference between 300ms and 900ms matters.

4.3 Chained Tool Calls

Not all tool calls are independent. Sometimes the output of one feeds into the next. The model has to search for a user (to get their ID), then use that ID to look up their profile or orders. It can't skip ahead.

This is multi-turn tool use. The conversation grows with each round-trip:



The implementation is a loop that keeps calling the API until the model stops requesting tools:

```python
def run_chained(user_message, max_rounds=5):
    messages = [
        {"role": "system", "content": SYSTEM_PROMPT},
        {"role": "user", "content": user_message},
    ]

    for round_num in range(1, max_rounds + 1):
        response = client.chat.completions.create(
            model=MODEL, messages=messages, tools=tools,
        )
        assistant_msg = response.choices[0].message

        # No tool calls = final answer
        if not assistant_msg.tool_calls:
            return assistant_msg.content

        # Execute all tool calls for this round
        messages.append(assistant_msg)
        for tc in assistant_msg.tool_calls:
            fn = available_functions[tc.function.name]
            args = json.loads(tc.function.arguments)
            result = fn(**args)
            messages.append({
                "role": "tool",
                "tool_call_id": tc.id,
                "content": json.dumps(result),
            })
```

The interesting thing is that chaining and parallelism compose. When the model needs both a profile and order history after a search, it calls both in the same round:

```
Round 1: search_users("Bob Smith")
  ──> {"user_id": "u_102"}

Round 2: get_user_profile("u_102")     [parallel]
         get_user_orders("u_102")      [parallel]
  ──> profile + orders

Round 3: "Bob Smith has the Basic plan. His total
          spending is $129.99..."
```

Search is sequential (can't call profile without the user ID), but profile and orders are parallel (both just need the ID). The model figures this out on its own.

4.3.1 Conversation growth

Each round adds messages. A three-round chained query produces 8 messages:

```
[0] system:    "You are a helpful assistant..."
[1] user:      "What plan is Alice on and what are her orders?"
[2] assistant: tool_calls: [search_users]
[3] tool:      {"results": [{"user_id": "u_101", ...}]}
[4] assistant: tool_calls: [get_user_profile, get_user_orders]
[5] tool:      {"user_id": "u_101", "plan": "Premium", ...}
[6] tool:      {"user_id": "u_101", "orders": [...], ...}
[7] assistant: "Alice is on the Premium plan. Her orders..."
```

This is the fundamental tradeoff of chained tool calls: each round adds context that helps the model make better decisions, but also consumes more tokens. For complex multi-step tasks, the conversation can grow quickly.

4.4 The system prompt matters

One subtle finding: without a system prompt telling the model to use tools proactively, smaller models tend to ask clarifying questions instead of chaining. They'll search for a user, see the result, and respond "I found Alice. Would you like to see her profile?" instead of just calling `get_user_profile`.

The fix is simple — tell the model what you expect:

```
SYSTEM_PROMPT = """You are a helpful assistant with access to
a user database. When a user asks about a person, ALWAYS use
the available tools to find the answer. Do not ask clarifying
questions -- use search_users to find the user, then use
get_user_profile or get_user_orders to get the details they
asked about."""
```

This is a general lesson: tool definitions tell the model *what it can do*, but the system prompt tells it *how aggressively to do it*. Both matter.

4.5 What we learned

2.1 established the core lifecycle. Two API calls, one local function execution, four messages in the conversation. The model reads schemas, not code.

2.2 showed that independent tool calls can be batched and executed concurrently. The 3x speedup (0.3s vs 0.9s) is free — same answer, less wall time.

2.3 demonstrated that the model can drive multi-step workflows by chaining tool calls across rounds. It even combines chaining with parallelism when it recognizes independent sub-tasks within a round.

The progression from here: error handling (what happens when tools fail?), the ReAct reasoning loop (thinking before acting), tool selection at scale (picking from 20+ tools), and the Model Context Protocol (standardizing tool integration across systems).

Chapter 5

Multi-Agent Systems

Not yet implemented.

Instead of one LLM doing everything, you give different LLMs different jobs — one researches, one writes code, one reviews. They pass messages to each other through an orchestration layer. The interesting question is whether specialization actually helps, or whether one good model with a good prompt beats a committee.

Chapter 6

Advanced Patterns

Not yet implemented.

Planning, reflection, and memory. Agents that break tasks into steps, execute them, notice when something went wrong, and try a different approach. The hard part isn't any single technique — it's combining them without the system becoming too slow or too expensive to run.