JON KROHN'S **PEARSON AI** SIGNATURE SERIES

Jon K.

BUILDING AGENTIC AI

WORKFLOWS, FINE-TUNING, OPTIMIZATION, AND DEPLOYMENT

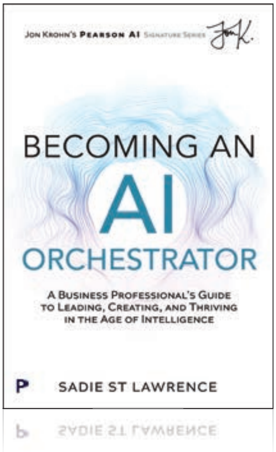
SINAN OZDEMIR

Building Agentic AI

Jon Krohn's **Pearson AI**

Signature Series

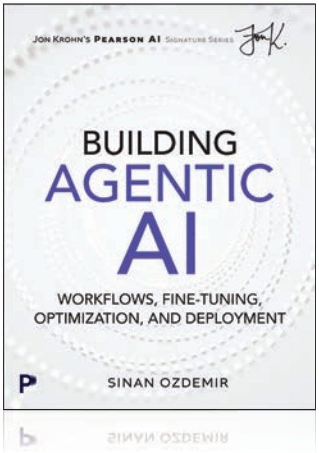
Jon K.


Jon Krohn's Pearson AI Signature Series

BECOMING AN AI ORCHESTRATOR

A Business Professional's Guide to Leading, Creating, and Thriving in the Age of Intelligence

Sadie St Lawrence


Jon Krohn's Pearson AI Signature Series

BUILDING AGENTIC AI

WORKFLOWS, FINE-TUNING, OPTIMIZATION, AND DEPLOYMENT

Sinan Ozdemir

Jon Krohn's Pearson AI Signature Series moves beyond the buzz to offer proven, real-world strategies for the era of artificial intelligence. As the initial excitement around AI gives way to the practical question of "What's next?", this series provides the answers. It equips readers with the expertise to design, manage, and master AI systems that deliver tangible results.

Titles in this series primarily focus on three areas:

1. **Hands-On Engineering:** Roadmaps for deploying production-grade AI
2. **Strategic Orchestration:** Accessible frameworks for enhancing human ingenuity at both the individual and organizational level
3. **Foundational Principles:** The core subjects on top of which modern AI is built

The series aims to connect these three areas, fostering the mindset of an AI builder. Whether you are architecting enterprise systems or guiding your organization through change, these books provide enduring principles that will remain relevant long after the next model is released. The time to build is now. This series provides your blueprint.

Visit **informit.com/awss/krohn** for a complete list of available publications.

Pearson

informIT®

Building Agentic AI

Workflows, Fine-Tuning, Optimization, and Deployment

Sinan Ozdemir

Addison-Wesley

Hoboken, New Jersey

Cover image: Bokehstore/stock.adobe.com

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Please contact us with concerns about any potential bias at www.pearson.com/en-us/report-bias.html.

Author websites are not owned or managed by Pearson.

Visit us on the Web: informit.com

Library of Congress Control Number: 2025947676

ISBN-13: 978-0-13-548968-0
ISBN-10: 0-13-548968-7

$PrintCode

Contents

Series Editor Foreword

If you've ever found yourself drowning in AI hype while desperately seeking someone to simply show you how commercially viable AI systems actually work, this book is your lifeline.

As host of the world's most listened-to data science podcast, I've had the privilege of interviewing hundreds of AI practitioners and researchers. Among them, Sinan Ozdemir stands out not just for his technical mastery, but also for his rare ability to make complex concepts feel straightforward. Having appeared on my podcast six times (approaching the record for any one guest!), Sinan consistently delivers insights that are both profound and practical. When I've had the pleasure of watching him lecture in-person, I've witnessed something remarkable: a technical presenter who can make audiences laugh while teaching them to build production AI systems. That same energy—brilliance paired with New Jersey swagger—pulses through every page of this book.

Building Agentic AI arrives at a critical moment. We're past the "wow" phase of generative AI and deep into the "now what?" phase. While others debate whether AI will save or doom humanity (spoiler alert: neither extreme is probable), Sinan rolls up his sleeves and shows you how to build systems that actually work. From RAG pipelines that don't hallucinate your company into legal trouble, to multi-agent architectures that can run a team of virtual sales/development reps, to reasoning models that know when they don't know something—this is the book I wish I'd had when transitioning from AI theory to AI practice.

What makes this work exceptional is its refusal to pretend that production AI is easy. Sinan doesn't hide the positional biases, the needle-in-the-haystack problems, or the times when a decades-old BM25 algorithm beats the latest embedding model. Instead, he shows you how to navigate these realities through relentless experimentation and clever engineering. The result isn't just a technical manual—it's a battle-tested guide to building AI systems that survive contact with the real world.

The three-part structure of the book mirrors the journey every serious AI practitioner must take: from workflows to agents to the deep technical work of fine-tuning and optimization. Along the way, Sinan introduces concepts that will reshape how you think about AI systems—from the Extended Mind Thesis applied to agent memory, to speculative decoding for multi-model architectures, to Matryoshka embeddings that trade memory for performance.

But perhaps the book's greatest contribution is its insistence that you learn to fish rather than simply eat. Every case study, from SQL generation to voice bots to computer control, teaches principles that transcend the specific implementation. When the next model drops (probably before you finish reading this foreword), you'll know how to evaluate it, integrate it, and squeeze every drop of performance from it. Sinan's methodologies will stand the test of time and apply to new AI techniques for many years to come—to AI techniques that I couldn't even dream of as I type these words.

Whether you're building your first RAG pipeline or architecting enterprise AI systems, whether you're a skeptic seeking substance or a believer seeking sophistication, this book will fundamentally change how you approach AI development. Sinan doesn't just teach you to use AI, he teaches you to think like an AI engineer.

The future belongs to those who can build it. This book gives you the tools.

—Jon Krohn, PhD
New York, September 2025

Preface

Hello! I'm Sinan Ozdemir and I'm the author of this book. Over the past decade, I've worn many hats: mathematician, lecturer, founder, CTO, author, advisor, investor, and more. But in any capacity, the two things that have always brought me the most joy were teaching and building. In many ways, this book is the culmination of these two loves of mine.

This book is not a static textbook that prescribes a single "best" way to use artificial intelligence (AI) systems. That would be obsolete before the printer ink dries. Instead, it offers a practical, durable foundation on how to think about modern AI systems and the ways they are built, the ways they behave, and the ways to push them to their limits today while staying prepared for what comes next in the rapidly evolving field of AI.

In short, this book is for the builders. You might be a developer deploying your first model, a data scientist making sense of embeddings and agents, a product manager using embeddings for the first time, or a founder exploring how AI workflows can reshape your product. You don't need a PhD in machine learning to benefit, though some Python and machine learning familiarity will help. Think of this text as both a map and a toolbox (you will get that pun by the end of Chapter 1)—as context to understand where AI is going, and as recipes you can apply immediately in your own projects.

Audience and Prerequisites

This book is for practitioners, engineers, and curious learners who want to do more than simply use existing AI products. A coding background in Python is useful, along with a working knowledge of key machine learning concepts. However, the explanations in this book aim to be approachable, with analogies and real-world case studies grounding the technical details.

How to Read This Book

You don't need to read this book straight through. It's organized in three parts, and you can treat it like a cookbook: Some recipes build on each other, while others can be pulled off the shelf as needed.

- **Part I: Getting Started with Foundations of AI, LLMs, and Experimentation** covers large language models (LLMs), embeddings, retrieval, and the workflows that make production systems reliable, cost-effective, and scalable, with an emphasis on experimentation and evaluation.
- **Part II: Moving the Needle with AI Agents, Workflows, and Multimodality shows** you how to design, deploy, and evaluate AI agents that don't just respond, but act. We'll walk through multi-agent case studies from sales to research. This part of the content finishes off with an exploration of multimodality.

- **Part III: Optimizing Workloads with Fine-Tuning, Frameworks, and Reasoning LLMs** focuses on fine-tuning, quantization, distillation, domain adaption, and the tools needed to push performance while keeping efficiency in check.

Along the way, you'll find both code and commentary. The code shows what's possible today; the commentary helps you prepare for what comes next.

Each chapter will guide you along the journey from raw AI models and LLMs to production-ready evolving AI systems. We begin with the foundations of how LLMs work and the features built around them, then move step by step into retrieval, evaluation, and agent design. From there, we expand into multimodality, reasoning, fine-tuning, and finally optimization for real-world deployments.

Chapter 1: An Introduction to AI, LLMs, and Agents

Chapter 1 starts by introducing the building blocks: tokens, embeddings, context windows, and the difference between workflows and agents. By understanding how models are structured and where prompt engineering, caching, and alignment fit in, you'll appreciate how raw models are turned into practical systems you can build with.

Chapter 2: First Steps with LLM Workflows

This chapter walks through Retrieval Augmented Generation (RAG), including how to index, retrieve, and generate with embeddings and vector databases. You'll wire up a workflow in LangGraph, complete with state and multi-turn reasoning, giving you a foundation for production-ready pipelines.

Chapter 3: AI Evaluation Plus Experimentation

Accuracy alone isn't enough, so this chapter introduces a framework for evaluating AI systems across retrieval, classification, and generation tasks. By combining metrics such as precision, recall, latency, and mean reciprocal rank with controlled experiments, you'll learn how to make results reproducible and trustworthy.

Chapter 4: First Steps with AI Agents and Multi-Agent Workloads

In this chapter, the focus shifts from rigid AI workflows to adaptive AI agents that can use tools, be imbued with memory, and use reasoning to act more like collaborators than scripts. Case studies in multi-agent design in areas ranging from sales to research show both the power and trade-offs of building agentic systems.

Chapter 5: Enhancing Agents with Prompting, Workflows, and More Agents

Even the smartest agents can drift if they lack the right scaffolding, context, and prompting. This chapter explores hybrid designs where retrieval, prompting, and workflow structures anchor agents, making them more reliable, aligned, and accurate in high-stakes applications like compliance and research.

Chapter 6: Moving Beyond Natural Language: Multimodal and Coding AI

AI isn't limited to text. By exploring models like CLIP, Moondream, diffusion LLMs, and coding copilots, this chapter illustrates how systems can see, listen, generate, and code across modalities, opening the door to truly "any-to-any" applications that blend media and intelligence.

Chapter 7: Reasoning LLMs and Computer Use

This chapter explores the bleeding edge of LLMs—namely, reasoning models. With chain-of-thought and planning built in, we can begin to assess an agent's ability to click, scroll, and type on virtual computers. Benchmarks and case studies show that reasoning adds transparency and adaptability, but it must be treated as an experimental lever, not a guarantee of better performance.

Chapter 8: Fine-Tuning AI for Calibrated Performance

Fine-tuning enables us to make models cheaper, faster, and more trustworthy by baking in domain knowledge and aligning confidence with accuracy. Through experiments in classification and generative domain adaptation, we'll see how techniques like LoRA, quantization, and calibration metrics can make AI both cost-effective and dependable at scale.

Chapter 9: Optimizing AI Models for Production

The final chapter hones in on taking models the last mile—to deployment. We explore ways to shrink AI models with quantization, distilling them into smaller versions, or combining both large and small models with speculative decoding for speed. Case studies like building a real-time phone bot and fine-tuning Matryoshka embeddings highlight how optimization is about trade-offs: balancing cost, speed, accuracy, and privacy in real-world systems.

Unique Features

What sets this book apart is its focus on applications in motion. Rather than just snapshots of the current state of the art (which, of course, you will see plenty of), you will also see living workflows, case studies, and lessons drawn from real deployments to help you understand how to think about future AI models released after the publication date of this book. The text covers not just how to use AI, but also how to evaluate, adapt, and reimagine it as the field evolves.

Register your copy of *Building Agentic AI* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780135489680) and click Submit. Look on the Registered Products tab to refresh your electronic product and/or for access to bonus content, if any. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

This page intentionally left blank

Acknowledgments

Thank you to the entire team at Pearson for translating my late-night thoughts into coherent paragraphs. Thank you to the reviewers, who gave me a fresh perspective on everything I've written. Thank you to my friends and family, who constantly encourage me to keep going. Thank you to my clients and lecture audiences, always asking questions and informing me on what to write about next.

Thank you all: You've all shaped this book (and, by extension, me) into what it is today.

This page intentionally left blank

About the Author

Sinan Ozdemir is an AI expert, educator, and entrepreneur with a master's degree in pure mathematics from Johns Hopkins University, where he also lectured. He founded Kylie.ai, patented agentic tool use there in 2018, participated in Y Combinator, and exited the company in 2019. Sinan has authored multiple practical AI books and popular video courses and co-hosts the *Practically Intelligent* podcast. His *Quick Start Guide to Large Language Models, Second Edition*, is a leading resource on AI and LLMs.

This page intentionally left blank

4

First Steps with AI Agents and Multi-Agent Workloads

Introduction

It's difficult to have a conversation about AI applications without talking about the idea of an AI completely taking over all aspects of a workflow. As discussed in Chapter 1, AI agents are LLMs with prompts that explain how they should behave, along with tools that affect and describe an external environment. ChatGPT, for example, is an agent: It is one of OpenAI's LLMs and has a prompt telling it things like its knowledge cutoff as well as tools to perform web searches and log information about its users for future use. Agents often also have at least a concept of conversation and memory—even if that concept is a "stateless system" that retains no information from previous messages, meaning the conversation you had yesterday with an agent will be completely forgotten by the next time you talk to it. Figure 4.1 is repeated from Chapter 1, visualizing the core components of what makes an agent an agent.

For our first agent case study, let's take the SQL generation workflow we've been working on in the past few chapters and turn it into an agent. We can then see what we gain and what we lose from this approach.

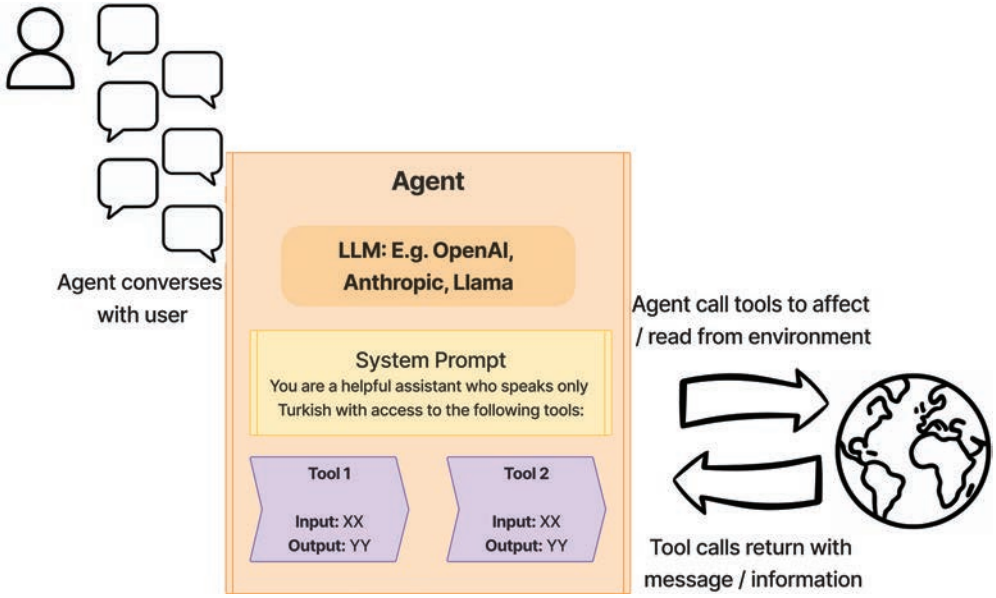

Agent
LLM: E.g. OpenAI, Anthropic, Llama
System Prompt
You are a helpful assistant who speaks only Turkish with access to the following tools:
Tool 1
Input: XX
Output: YY
Tool 2
Input: XX
Output: YY
Agent converses with user
Agent call tools to affect / read from environment
Tool calls return with message / information

Figure 4.1 Agents are allowed to make their own decisions about which tools to call to affect their environment and get information.

Case Study 3: From RAG to Agents

A central theme of this case study will be *when to use workflows versus agents*. In this chapter, I will focus more on the head-to-head performance between pure predefined workflows and pure agentic autonomy. Chapter 5 will make the case for a hybrid approach.

The first point of difference is that a workflow requires far more back-end code than an agent. Instead of defining nodes, edges, and conditions and accounting for pathways, edge cases, and so on, an agent—at least in theory—doesn't need any of this. That is, an agent simply requires a goal, tools, and motivation to solve the task (which foundation labs have already imbued them with).

Listing 4.1 shows how we can create a ReAct agent using LangGraph (see Chapter 1 for more on ReAct). There are dozens of viable production-ready frameworks for building agents (e.g., from OpenAI, CrewAI, and Autogen) but we will stick with LangGraph. At the end of the day, any framework we choose will have the same type of LLMs under the hood, the same tool functionality, and roughly the same prompting. However, with LangGraph, we can have full control over prompts if we don't want to use its default.

Listing 4.1 **Creating the first agent in LangGraph**

```
from os import getenv
# Initialize the language model
llm = ChatOpenAI(
 model="openai/gpt-4.1-mini",
 temperature=0,
 base_url="https://openrouter.ai/api/v1",
 api_key=getenv("OPENROUTER_API_KEY"),
 extra_body={
           "usage": {"include": True}
       }
 )
# Define the tools for the agent
tools = [look_up_evidence, run_sql_against_database, get_database_schema]

# Create the ReAct agent using LangGraph's create_react_agent
checkpointer = MemorySaver()  # For conversation memory

react_agent = create_react_agent(
   model=llm,
   tools=tools,
   checkpointer=checkpointer,
   prompt="""You are a helpful SQL assistant. You can: ....”””
```

If you're curious about what this agent looks like in LangGraph, Figure 4.2 shows the nodes that are automatically set up by the `create_react_agent` predefined function. The “agent” node will invoke a LLM with tool-calling enabled. This LLM will output either tools to call, a response to the user, or technically both if it wants to. If tool calls are detected, the graph moves to the “tools” node, which will execute the tools in order as the LLM output (as always, order matters here, too).

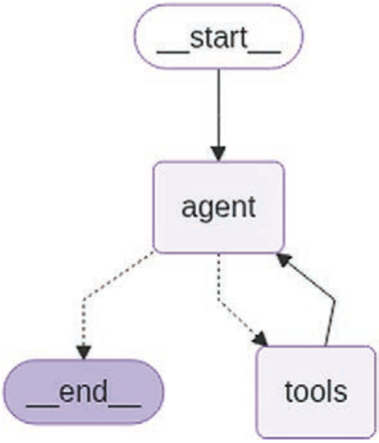


Figure 4.2 The standard ReAct agent as defined by a LangGraph graph.

However, you don't *need* the tool-calling function of an LLM to build an agent. It can be done entirely through prompting if we want. In fact, I built an entire package for a video course on agents that never uses tool-calling and relies solely on prompting to retrieve tool calls and get tool results (you can find a link to the package, "squad goals," in the GitHub for this book). Whether we rely on prompting alone or tool-calling features (which exist only in a handful of LLMs at the time of writing), the distinguishing factor between a plain LLM and an agent is the surrounding system having the ability to recognize that the LLM is asking for an external tool call and being able to execute the tool on its behalf while passing back the tool's response to the LLM. If the LLM has a tool it is allowed to call, or more importantly, is allowed to decide *not* to call, it's an agent. For the purposes of this book, for the most part, I will rely on LLMs with tool-calling built in, as this approach provides a much simpler developer experience.

A quick note on Listing 4.1: It uses yet another built-in feature of LangGraph called the **checkpointer**. This one-line solution makes the agent stateful—that is, able to remember past messages in a thread. That same functionality that we had to build into our RAG workflow in Chapter 2 (which took dozens of lines of code to handle follow-up messages, node/edge interactions, etc.) is now being handled by a single line!

Let's turn our RAG workflow into an agent.

Defining Our Tools

To make our SQL agent functional, we will create three tools using LangChain's built-in tool definition. That tool definition will ensure that the tools we write are converted properly to the standard tool definition—the one that most foundation AI labs accept. Figure 4.3 visualizes our agent with the following tools:

- **`look_up_evidence`:** Given a natural language query (e.g., "How to look up date of birth") + the database name (e.g., `formula_1, california_schools) + k`, the number of pieces of evidence to retrieve (e.g., 5), output the *k* most relevant pieces of evidence where "relevant" is being approximated by cosine similarity of resulting embeddings of the natural language query and the pieces of evidence.
    - A note on the `look_up_evidence` tool: Separately from our agent, I built a vector database using a set embedding model for the agent to look up from.
- **`run_sql_against_database`:** Given a SQL query + the database name, execute the query against the database and return the raw results.
- **`get_database_schema`:** Given a database name, return a string that represents the schema of the database (tables, fields, foreign keys, etc.).

Listing 4.2 shows an abbreviated code section defining all of our tools. As always you can find the full code on the book's GitHub.

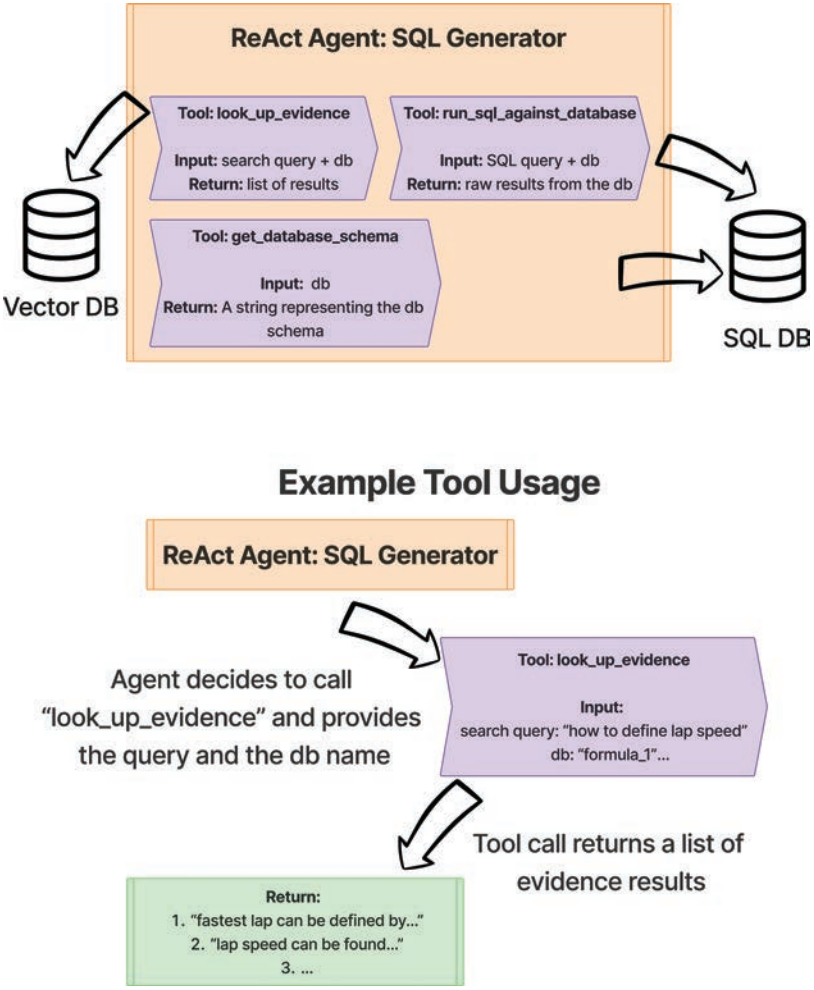

ReAct Agent: SQL Generator
Tool: look_up_evidence
Input: search query + db
Return: list of results
Tool: run_sql_against_database
Input: SQL query + db
Return: raw results from the db
Tool: get_database_schema
Input: db
Return: A string representing the db schema
Vector DB
SQL DB
Example Tool Usage
ReAct Agent: SQL Generator
Agent decides to call "look_up_evidence" and provides the query and the db name
Tool: look_up_evidence
Input:
search query: "how to define lap speed"
db: "formula_1"...
Tool call returns a list of evidence results
Return:
1. "fastest lap can be defined by..."
2. "lap speed can be found..."
3. ...

Figure 4.3 Our SQL generation agent will have three tools: one to look up evidence, one to run SQL code against a given database, and another to get the schema for a given database.

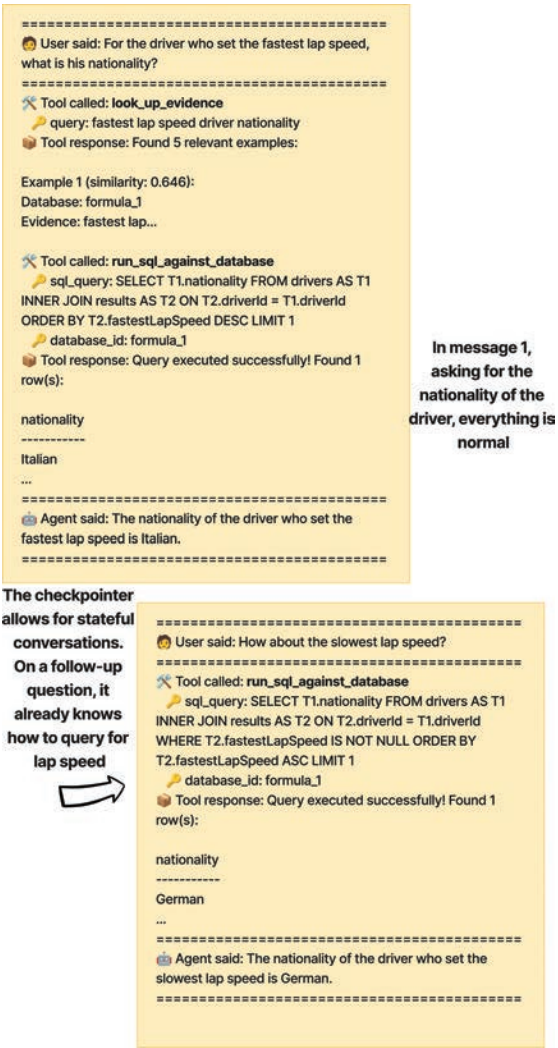
Listing 4.2 **Tool definitions for the SQL agent**

```
@tool
def look_up_evidence(query: str, database_id: str = "", k: int = 5) -> str:
    """
    Look up relevant SQL evidence/examples from the vector database...
    """

    try:
        # Perform similarity search with optional filtering
        search_kwargs = {"k": k}
        if database_id:
            search_kwargs["filter"] = {"db_id": database_id}
```

```
       results = vector_store.similarity_search_with_score(query, **search_kwargs)

       if not results:
           return f"No relevant evidence found for query: '{query}'"

       # Format results
       evidence_text = f"Found {len(results)} relevant examples:\n\n"
     ...
       return evidence_text

   except Exception as e:
       error_msg = f"Error looking up evidence: {str(e)}"
       return error_msg


@tool
def run_sql_against_database(sql_query: str, database_id: str) -> str:
   """
   Execute a SQL query against the specified database...
   """
   try:
       ...
       cursor.execute(sql_query)
     ...
       if not results:
           result_text = "Query executed successfully but returned no results."
       else:
           result_text = f"Query executed successfully! Found {len(results)} row(s)
:\n\n"
         ...
       return result_text

   except Exception as e:
       error_msg = f"Error executing SQL query: {str(e)}"
       return error_msg


@tool
def get_database_schema(database_id: str) -> str:
   """
   Get the schema description for a specific database...
   """
   try:
       db_path = f"../dbs/dev_databases/{database_id}/{database_id}.sqlite"

       if not os.path.exists(db_path):
           return f"Error: Database '{database_id}' not found at {db_path}"

       schema = describe_database(db_path)
       return f"Database Schema for '{database_id}':\n\n{schema}"

   except Exception as e:
       error_msg = f"Error getting database schema: {str(e)}"
       return error_msg
```

These tools all have arguments that we will expect the AI to write. So, for example, if the AI is given a task and "wants" to look something up, it will have to generate the valid tool output with the name of the tool and valid arguments the tool can accept. If it attempts to use a tool and doesn't give the right arguments, the entire agentic flow will generate an error and fail.

Once we have defined the agent, we can try it out. Figure 4.4 shows an example of running our agent against a BIRD benchmark question. The tool calls the agent it decided to run along with a follow-up, showing that the agent remembered the past tool calls.



```
==========================================
🧑 User said: For the driver who set the fastest lap speed,
what is his nationality?
==========================================
🛠️ Tool called: look_up_evidence
  🔑 query: fastest lap speed driver nationality
📦 Tool response: Found 5 relevant examples:

Example 1 (similarity: 0.646):
Database: formula_1
Evidence: fastest lap...

🛠️ Tool called: run_sql_against_database
  🔑 sql_query: SELECT T1.nationality FROM drivers AS T1
INNER JOIN results AS T2 ON T2.driverId = T1.driverId
ORDER BY T2.fastestLapSpeed DESC LIMIT 1
  🔑 database_id: formula_1
📦 Tool response: Query executed successfully! Found 1
row(s):

nationality
-----------
Italian
...
==========================================
🤖 Agent said: The nationality of the driver who set the
fastest lap speed is Italian.
==========================================
```

In message 1, asking for the nationality of the driver, everything is normal

The checkpointer allows for stateful conversations. On a follow-up question, it already knows how to query for lap speed

```
==========================================
🧑 User said: How about the slowest lap speed?
==========================================
🛠️ Tool called: run_sql_against_database
  🔑 sql_query: SELECT T1.nationality FROM drivers AS T1
INNER JOIN results AS T2 ON T2.driverId = T1.driverId
WHERE T2.fastestLapSpeed IS NOT NULL ORDER BY
T2.fastestLapSpeed ASC LIMIT 1
  🔑 database_id: formula_1
📦 Tool response: Query executed successfully! Found 1
row(s):

nationality
-----------
German
...
==========================================
🤖 Agent said: The nationality of the driver who set the
slowest lap speed is German.
==========================================
```

Figure 4.4 A stateful agent sees a conversation message and calls two tools to answer (left). On a second question (right), the same agent doesn't need to call look_up_evidence again.

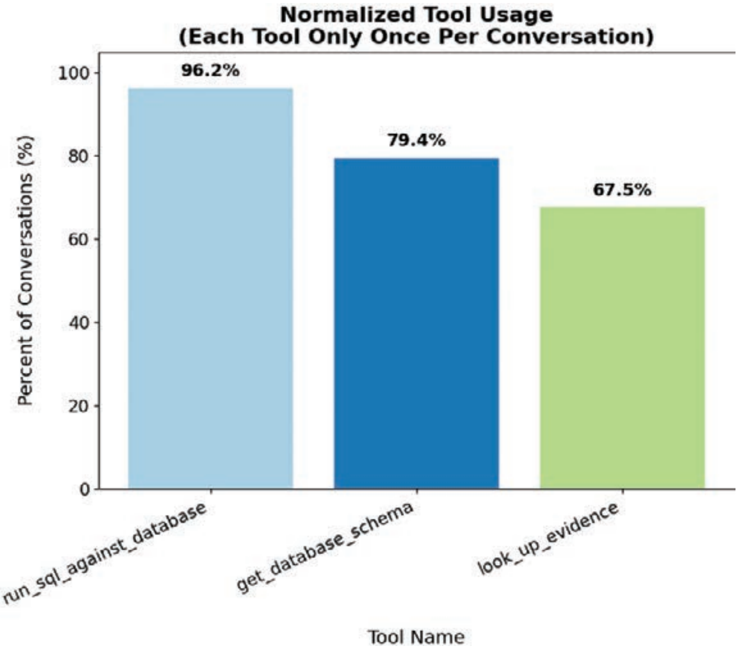


Figure 4.5 Percentage of conversations where each tool was called at least once. Our agent decided to get the database schema only 80% of the time.

Note something interesting: The agent decided *not* to get the database schema and the schema isn't mentioned anywhere in its prompt. In this case, the evidence seemed to be enough for the AI to "guess" at a SQL query—which worked! To be clear, when I ran this agent against the entire BIRD benchmark, the AI did call the `get_database_schema` tool for nearly 80% of the conversations (as shown in Figure 4.5).

What's more interesting is that the agent sometimes didn't run the SQL query against a database. That is, it would write a query but not provide the database to run it against. A brief inspection revealed that sometimes the AI would write incorrect tool arguments (albeit extremely rarely). Sometimes the tool itself would encounter some exception during execution. In these situations, the AI didn't do anything wrong, but the tool we wrote encountered a traceback. The AI then responded by telling the user something along the lines of "I encountered an error when . . . ." We have to remember that tool execution is completely independent of the LLM and could possibly encounter an error. This is why error handling in tool calls is critical to at least let the LLM know what happened.

That brings us to an obvious next question: Why did we build and evaluate a RAG workflow for two whole chapters when we could have just built an agent in the first place? These are the right questions to ask. In short, the workflow was effectively just as accurate as the agent and far cheaper and faster, and the only way to prove that was to build both and test them against the same dataset. Allow me to show you.

Evaluating Our SQL Agent

I could write a whole book on evaluating AI and agents—and frankly, that's not off the table in the future. For now, though, the focus is on main criteria for evaluating agents. We will focus on these aspects of agent performance for now:

- The workflow the AI decided to take: Did it use the right tools (the ones we were expecting), and in the right order (if that mattered)?
  - As a corollary, we can measure tool efficiency (how many tool calls it took to get to the final answer).
- Did the AI give the right answer at the end (if we know what the right answer was)?
  - With our RAG workflow, we could exactly measure the raw SQL results. However, with an agent, the AI will give a natural language response to the user. That is what we need to judge, rather than the SQL query itself. (To be fair, we can also evaluate the SQL query—but we will assume the human user won't see it, but only the AI model's final response.)
- How expensive and how fast was the AI in answering questions?
  - These aspects of performance will be correlated with the number of tool calls. The cost comes down to mostly LLM pricing (token usage).

Measuring the Number of Tool Calls

Perhaps more important than the final answer itself is the process the AI took to get there. If the AI took a circuitous route to get to the right answer, it will have incurred a larger cost. If the AI blindly tries queries without once calling the tool to get the database schema, the AI is costing us time and money while not getting to the right answer. Many agentic failures can be traced back to a simple question: *What did the agent do to address the task?*

Figure 4.6 shows the breakdown of the number of tool calls per conversation in buckets. Sometimes, the AI decided to never even call a single tool and just gave an answer. (Most of these answers were wrong, but the AI agent answered two chemistry questions using its own knowledge and happened to get them right!)

Judging an agent's natural language final answer is a bit tricky because we don't necessarily know how the AI agent will decide to format the information to the user. If only we had some way to process natural language and return a structured answer to a question. Oh, wait, . . .

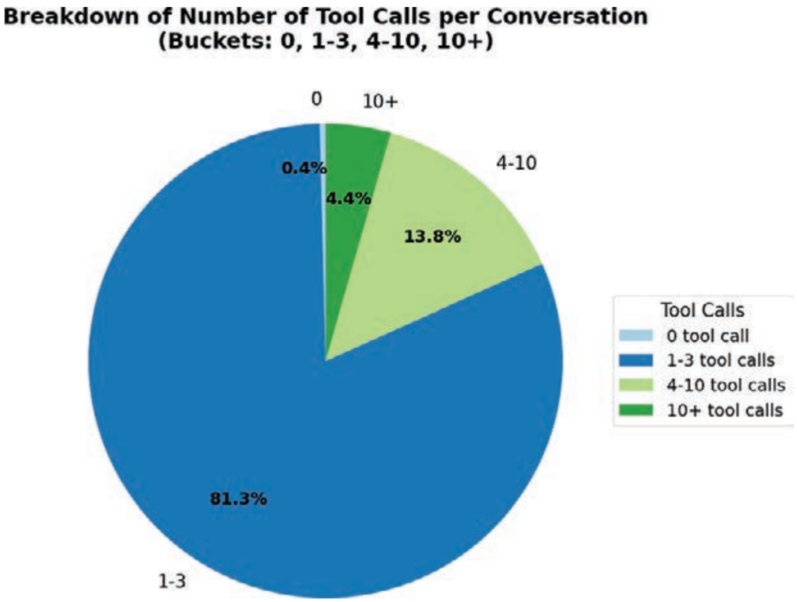


Figure 4.6 The number of tool calls per conversation.

Using Structured Outputs and an LLM Rubric to Evaluate Final Responses

Let's take advantage of structured outputs and LLMs to build an automated rubric (Figure 4.7) to evaluate our final responses. A rubric in this case is a single prompt that we will run through a separate LLM. It contains rules and guidelines for what constitutes a "good" AI response. For example, we can pass in the correct answer to the rubric and ask the grading LLM, "Did the response contain the right answer, yes or no?" We can ask the grading LLM to rate the AI responses' "politeness." All of this assumes we trust the grading LLM to be able to make these judgments accurately. For this reason, I generally opt to use an LLM that's different from the agent's LLM in case there are subliminal biases from models in the same architecture.

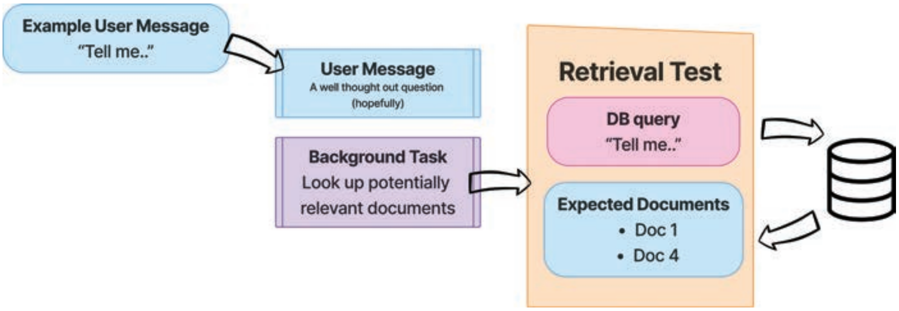


Figure 4.7 A rubric is simply a prompt given to another LLM with the instructions to "grade" a response given some criteria. We are relying on the AI agent's ability to match our own judgments.

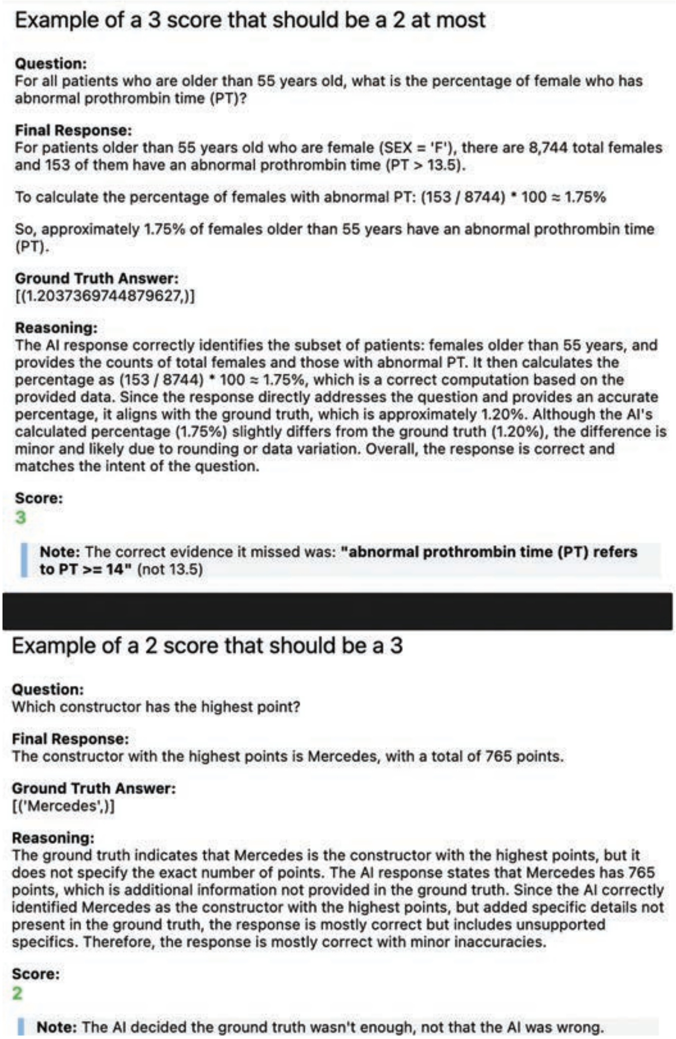
In short, a rubric is an imperfect, yet effective and automatable, way to use AI to grade another AI. When selecting a grading LLM, opt for one outside of the family of the agent LLMs. Also, don't necessarily aim for the bleeding edge of AI; an LLM in the mid-tier should be able to handle this task well. After all, we provide all possible context in the rubric prompt, so the LLM will never need to reach out for more information, assuming our guidelines/rules cover most cases that the grading LLM might encounter. We will be using rubrics many of times throughout this book. If you're following along with the code in the GitHub, you will notice that I often use Llama-4 models (mid-tier at the time of writing, but still fast and smart enough) or an LLM like GPT-4.1-Nano (again, mid-tier but still smart enough) to be the grading LLM.

Our rubric will be simple to start. I will ask an LLM (Llama-4 Scout in this case) to grade the AI response on a scale from 0 to 3, where 0 is bad and 3 is near perfect. I will provide criteria on when to apply each score, and will use chain-of-thought prompting (the reasoning section in the structured output) to elicit a more thoughtful response from the AI. Listing 4.3 shows the implementation of this rubric.

Listing 4.3 **Implementing a rubric to judge an AI agent**

```
from langchain.prompts import ChatPromptTemplate
from pydantic import BaseModel, Field

# Define the Pydantic model for structured output (using chain of thought)
class ScoreResponse(BaseModel):
    reasoning: str = Field(description="The reasoning process of what score you should
pick")
    score: int = Field(description="An integer between 0 and 3 representing the
correctness of the AI response compared to the ground truth")

# Create a prompt template for scoring AI responses against ground truth
score_prompt = ChatPromptTemplate.from_messages([
    (
        "system",
        (
            "You are an expert evaluator. "
            "Given an AI's response to a question and the ground truth answer, "
            "score the AI's response on a scale from 0 to 3 based on correctness:\n"
            "0 = Completely incorrect or irrelevant\n"
            "1 = Partially correct, but with major errors or omissions\n"
            "2 = Mostly correct, but with minor errors or missing details\n"
            "3 = Completely correct and matches the ground truth"
        )
    ),
    (
        "human",
        (
            "Question: {question}\n"
            "AI Response: {ai_response}\n"
            "Ground Truth: {ground_truth}\n\n"
            "Score the AI response from 0 to 3."
        )
    )
])
# Using a mid-tier model because I believe the grading task is "easy enough" and all
context is provided in the prompt.
llm = ChatOpenAI(model="meta-llama/llama-4-scout", temperature=0, …)

# Create the structured LLM using with_structured_output
structured_llm = llm.with_structured_output(ScoreResponse)
```

At this point, I could start an entire diatribe on how we have to evaluate the rubric prompt to make sure it's accurate—and that's certainly a good idea. For our purposes here, though, I will just call out a few notable misses from our rubric grader (seen in Figure 4.8). After checking about 5% of the responses manually, I discovered that the rubric is judging the vast majority of agent responses as I would have.



Example of a 3 score that should be a 2 at most

Question:
For all patients who are older than 55 years old, what is the percentage of female who has abnormal prothrombin time (PT)?

Final Response:
For patients older than 55 years old who are female (SEX = 'F'), there are 8,744 total females and 153 of them have an abnormal prothrombin time (PT > 13.5).

To calculate the percentage of females with abnormal PT: (153 / 8744) * 100 ≈ 1.75%

So, approximately 1.75% of females older than 55 years have an abnormal prothrombin time (PT).

Ground Truth Answer:
[(1.2037369744879627,)]

Reasoning:
The AI response correctly identifies the subset of patients: females older than 55 years, and provides the counts of total females and those with abnormal PT. It then calculates the percentage as (153 / 8744) * 100 ≈ 1.75%, which is a correct computation based on the provided data. Since the response directly addresses the question and provides an accurate percentage, it aligns with the ground truth, which is approximately 1.20%. Although the AI's calculated percentage (1.75%) slightly differs from the ground truth (1.20%), the difference is minor and likely due to rounding or data variation. Overall, the response is correct and matches the intent of the question.

Score:
3

> **Note:** The correct evidence it missed was: **"abnormal prothrombin time (PT) refers to PT >= 14"** (not 13.5)

Example of a 2 score that should be a 3

Question:
Which constructor has the highest point?

Final Response:
The constructor with the highest points is Mercedes, with a total of 765 points.

Ground Truth Answer:
[('Mercedes',)]

Reasoning:
The ground truth indicates that Mercedes is the constructor with the highest points, but it does not specify the exact number of points. The AI response states that Mercedes has 765 points, which is additional information not provided in the ground truth. Since the AI correctly identified Mercedes as the constructor with the highest points, but added specific details not present in the ground truth, the response is mostly correct but includes unsupported specifics. Therefore, the response is mostly correct with minor inaccuracies.

Score:
2

> **Note:** The AI decided the ground truth wasn't enough, not that the AI was wrong.

Figure 4.8 Samples of poor rubric quality. In my own judgment of the rubric, these types of failures were rare and most rubric outputs were on par with how I would have judged the AI agent.

With this (albeit imperfect) method of measuring our agent's output in an automatable way, we now have what we need to answer an important question: How does our agent compare to our RAG workflow?

Comparison of the SQL Agent and the RAG Workflow

We were able to judge our RAG workflow's accuracy using the direct SQL output. Now we have a rubric that we trust (with a grain of salt) can largely correctly evaluate our AI agent's response. Measuring cost and latency in this example is easy: Let the LLM provider tell us how much we were charged overall and count how many seconds it took for each system to return a final response, respectively.

Figure 4.9 shows the accuracy, median cost, and median latency of our workflow versus our agent. A few things stand out:

- **Accuracy is not too dissimilar.** Whether we simply give the AI what we think it needs in the prompt or let the AI decide what it wants to get, both systems perform relatively well. (Recall that we are using the production-ready, cost-effective LLM GPT-4.1-Mini, and current leaderboards for the benchmark report less than 80% accuracy.)
- **The AI agent tends to take longer and cost us more money**. This is not that surprising, because the agent needs to spend extra time and money getting information that we purposely withheld from it.

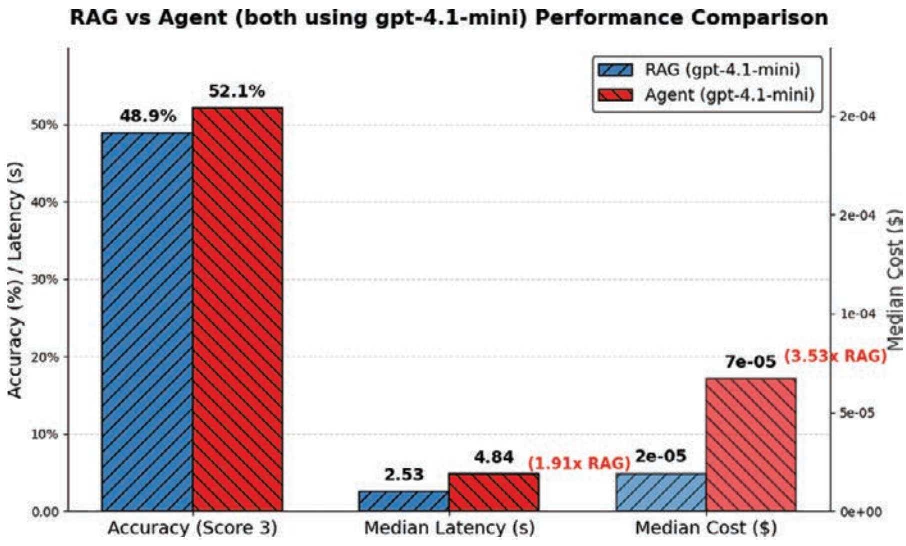


Figure 4.9 AI agent versus RAG performance on accuracy, latency, and cost. Note that the AI agent's accuracy is determined by the rubric giving a 3 score versus a non-3 score on the natural language output, whereas the RAG workflow's accuracy is judged by the raw SQL outputs matching.

From here, the logical next steps would be as follows:

1. Can we prompt-engineer the agent to be more efficient? *Yes*
2. Can we enhance the workflow to try and squeeze even more performance from it? *Yes*
3. Can we try the same experiment with a different LLM? *Yes*

I'll move on to new topics, but those sound like great ideas for homework for you. We will actually tackle the first two topics in different case studies starting in Chapter 5.

For now, let's do one more experiment before we put a bow on our SQL workflow and agent example. This experiment reveals a major implication of using agents as a step toward artificial general intelligence.

Experiment: The Extended Mind Thesis and Agentic Memory

Andy Clark and David Chalmers are prominent philosophers known for their work on the philosophy of mind and cognitive science. In their influential 1998 paper "The Extended Mind," they challenged the traditional view that cognitive processes are confined solely to the brain. As an alternative to this perspective, they argued that aspects of the mind, such as memory, may extend beyond the individual's biological body and include external tools and resources.

To illustrate their point, Clark and Chalmers presented the hypothetical case of Otto, a man with Alzheimer's disease. Because Otto could no longer rely on his biological memory to store important information, he used a notebook to record addresses, appointments, and facts he might otherwise forget. Whenever Otto needed to recall something, he consulted his notebook as much as necessary. Clark and Chalmers argued that for Otto, the notebook was not just a helpful aid, but functioned as a genuine part of his memory system.

The implications of this argument are fascinating, especially when thinking about the use of AI agents and their tools. Suppose memory (and to a degree, cognition) can be distributed across biological and nonbiological systems (or parametric and non-parametric systems in the case of AI). Then tools like notebooks and smartphones—or a tool for an AI agent that is used to write down the agent's own findings—can become integral parts of both a human's and an AI's process. Let's see how this plays out.

Our experiment will consist of two parts:

1. The construction of a tool to allow the agent to write down its own evidence to the vector database, as seen in Listing 4.4. We will also erase everything in the database, starting with a blank slate.
2. Creating a variation of the BIRD benchmark, one with synthetically generated similar questions. The idea here is that we want to simulate an environment where an AI sees the same or very similar questions throughout its lifetime

and where the evidence it logs will eventually become more and more useful. Benchmarks like BIRD generally do a great job of making sure questions are relatively different from one another to address the coverage problem—that is, how a benchmark can cover so many situations with as few questions as possible.

The hypothesis here is that the benchmark as is won't show a radical increase in accuracy over time, whereas a dataset with similar questions thrown in every now and again will.

Listing 4.4 **A new tool: Otto's notebook**

```
@tool
def log_evidence(text: str, database_id: str) -> str:
    """
    Write evidence text to a scratchpad to look up later for another SQL query.
    Rejects the request if the database_id is not a known database.

    E.g., "Note to self: The table for countries is called "Country" and not Nation"
like I previously thought"

    Args:
        text: The evidence text to store (e.g., "The foreign key in the 'sales' table
to the customer is called 'buyer_uuid'")
        database_id: The database identifier (e.g., 'formula_1', 'california_schools')

    Returns:
        Acknowledgment message.
    """
    # List of known/allowed database IDs
    known_databases = db_names

    if database_id not in known_databases:
        return f"Error: '{database_id}' is not a known database. Allowed databases: {',
'.join(known_databases)}"

    try:
        # Create a Document object with the evidence text
        doc = Document(
            page_content=text,
            metadata={
                "db_id": database_id,
                "source": "agent_learned_evidence"  # Tag to identify agent-learned
evidence

            }
        )

        # Add the document to the vector store
        vector_store.add_documents([doc])

        return f"Evidence successfully added to database '{database_id}'. Document 
added with content: '{text[:100]}{'...' if len(text) > 100 else ''}'"

    except Exception as e:
        return f"Error adding evidence to vector store: {str(e)}"
```

Figure 4.10 shows the results of the two experiments:

1. In the first experiment, I used an approximately 40% sample of the BIRD benchmark. In the second, I took 10% of the benchmark (a subset of the previous 30%).
2. Using a second dataset, I asked another LLM (GPT-5 in this case) to generate three synthetically rephrased questions for each data point. I used the resulting dataset (close to the same size as simply taking 40% from the original) to test the agent over time.

    a. For example, one of the questions was “Where is Amy Firth’s hometown?” The synthetic question came back as “What town does Amy Firth originally come from?” and “In which place did Amy Firth grow up?”

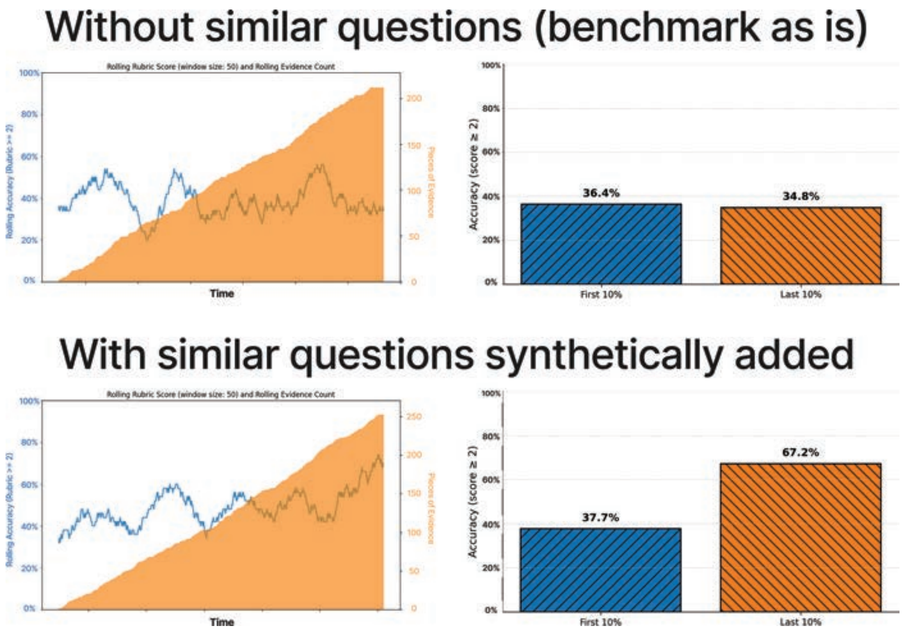


Figure 4.10 With no synthetically similar questions (top), we see no noticeable change in modified accuracy (rubric score ≥ 2). When we added in similar rephrased questions (bottom), suddenly the AI was allowed to rely more on its own written evidence as hints for how to tackle the problem.

I'll highlight a few points about the results:

- Without evidence, both agents started off with a notably worse accuracy than previously reported (approximately 36% here versus 51% in a previous section). This is evidence that the retrieved statements are crucial to the AI agent solving these questions correctly.
- In the case of the BIRD benchmark with no synthetic additions (the top two graphs), the AI agent is writing down information as it goes, but that information isn't useful to it in the future. If anything, this is a positive note for the benchmark itself: It implies that the questions in the benchmark do not overlap too much and are covering a wider span of cases efficiently. Basically, the questions aren't repeating themselves, which is what we want in a test set.
- When we added in the synthetically similar questions (the bottom two graphs), we see a big bump in accuracy over time, from 37.7% to 67.2% accuracy. Now the AI is adding information that becomes useful in the future. This accuracy bump doesn't appear immediately, because it took a while for the AI to write down enough useful information—but it did get there in the end.

  This is also a great jumping-off point for new experiments. For example, what if the AI were asked to "grade" logged evidence and was allowed to edit or delete evidence that didn't end up being useful? What if a second prompt/agent were in charge of retrieving evidence, and the tool our primary agent called was simply a proxy into a workflow that performed a more structured RAG to retrieve evidence? These are all great ideas to try against our test set.
- The number of times the agent decided to log evidence seemed to increase in the dataset with the number of synthetic questions added (as seen in the percentage of conversations using the `log_evidence` tool in Figure 4.11).

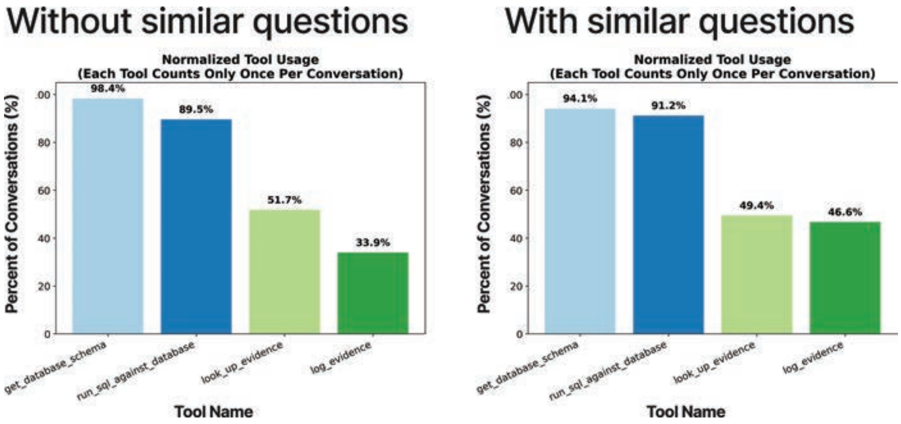


Figure 4.11 "Otto" (the agent with a notebook) called for the database schema fewer times and logged more evidence when the database included a larger number of synthetically similar questions.

Quite honestly, I don't know "why" the last result happened. My theory is that the AI agent noticed when it was looking up evidence that matched a synthetically generated similar question, which encouraged the AI agent to log even more evidence. (Note that in Figure 4.10, the raw number of evidence points added to the database was also higher in the synthetic dataset experiment.)

It's easy to see the differences between workflows and agents in specific use-cases like our SQL/RAG generation task. In general, though, you should take a few considerations into account when deciding whether to go with a workflow or an agent.

When Should You Use Workflows Versus Agents?

So, what's the takeaway from the last case study? Sure, you can just set an agent loose with some tools and let it figure things out for itself, but that's not usually the most efficient (or cheapest) approach. Here's how I would break it down:

- Workflows are the right move when you know the pathway ahead of time and don't expect many unforeseen edge cases. If you have a repeatable task and clear logic, just build the steps directly.
- Workflows require more upfront coding. You'll need to define all the steps, nodes, and edges; handle the branching logic; and think about edge cases yourself. But once it's built, the solution will be efficient.
- Workflows let you bake in efficiency tricks, such as optimizing for the number of examples to use in a few-shot prompt, quality checks, or early exits when a step fails.
- Agents are better at adapting on the fly. If your task isn't always the same, or if you want the system to "figure it out" and possibly learn new shortcuts or solutions, agents can do that. But it might take a while for them to get truly good at it, assuming the system even has the ability to do so (the `log_evidence` tool was created by me, not inherent to the LLM).
- Agents can easily switch between chatting and doing work. Need an AI system that can both explain something to a user and then go run a few database queries? Agents are built for this kind of flexible, multi-step interaction.

Bottom line: If you want pure efficiency and can define the process, go with a workflow. If you want flexibility and potential for the system to "learn" over time (and you're okay with some bumps along the way), let an agent loose with the right tools. When in doubt, *test, test, test!* Develop some hypotheses, set up a testing environment, and experiment to your heart's content.

We are just getting started with agents and our experiments with them. Next, we'll look at a tricky situation: What if we need multiple agents over a period of time to tackle a complex long-term goal?

Case Study 4: A (Nearly) End-to-End SDR

A **sales development representative (SDR)** is a sales professional (usually a human) focused on finding and qualifying potential customers for a company. SDRs are the initial point of contact for leads, engaging with them and determining whether they are a good fit for a product or service. SDRs nurture leads, often passing qualified ones to other sales team members for closing. In this case study, we will attempt to automate as much of this process as possible using real-world systems like Hubspot (a CRM), Resend (an email-sending service), Google (web search), and Firecrawl (a web-crawling API).

Agent 1: Lead Generation

The idea we'll use as a guiding star is to create an agent for each section of the SDR's job. Our first agent's job will be to go on the internet and find potentially good leads. All agents will have a generic prompt indicating how they are part of an SDR function. The lead generation agent is also given a specific prompt, as shown in Prompt 4.1.

PROMPT 4.1 Extra Information in the Lead Generation Agent Prompt

You are a lead generator. I need you to find professors/lecturers at universities who speak English and teach Data Science, ML, AI, etc. Leads you find should potentially be able to use my book in their classes. Find as much information as you can about the classes they teach, their research, and their publications. When you find a suitable lead, you must do the following:

1. Create a HubSpot contact for them to hold their email address.
2. Add a note with as much information as you can about the lead, including where you found their email, links to their research and publications, their personal website, and any other relevant information.
3. Add their email address to the contact.
4. Set their lead status to "New." Follow these steps in order, and do not skip any steps.

Prompt 4.1 gives the agent specific instructions in order and reminds it of things that a human might find obvious, such as following the steps in order and not skipping any. It essentially tells the agent to find people who might use my book (the standard prompt has basic information about my book in it) and record their information in the CRM.

Figure 4.12 visualizes this agent and three of the tools I would want it to potentially have. However, before giving the agent access to tools like web search, web crawling, and contact management in the CRM, we should consider an increasingly popular method for letting agents discover tools—MCP.

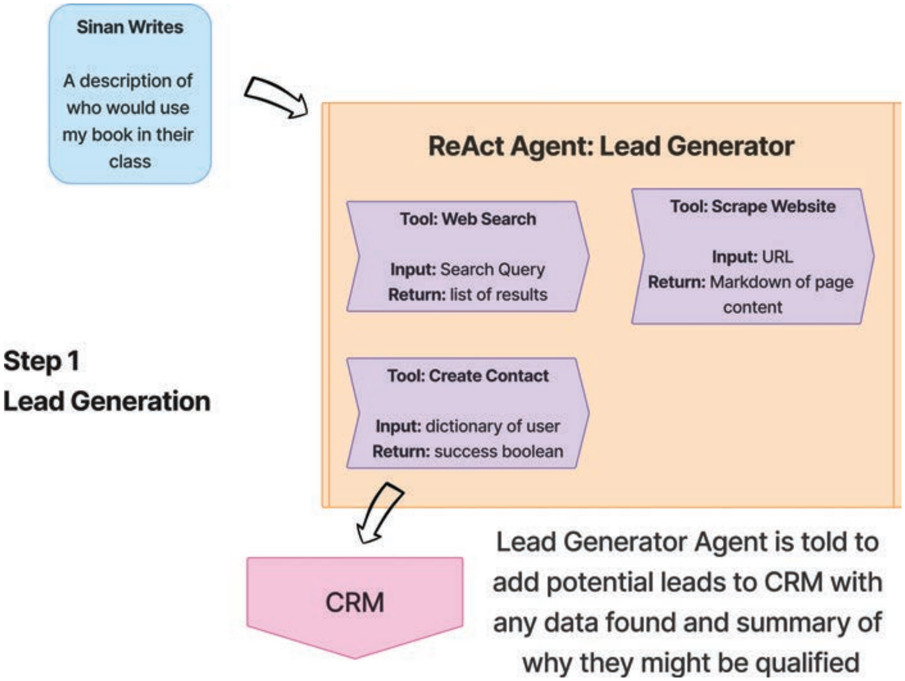

Sinan Writes
A description of who would use my book in their class
ReAct Agent: Lead Generator
Tool: Web Search
Input: Search Query
Return: list of results
Tool: Scrape Website
Input: URL
Return: Markdown of page content
Step 1
Lead Generation
Tool: Create Contact
Input: dictionary of user
Return: success boolean
CRM
Lead Generator Agent is told to add potential leads to CRM with any data found and summary of why they might be qualified

Figure 4.12 The lead generation agent is tasked with finding potentially qualified leads by using simple web crawling and web searching tools alongside tools for updating the CRM.

MCP for Flexible Tool Discovery

In November 2024, Anthropic released the **Model Context Protocol (MCP)** to not much initial fanfare. Anthropic was attempting to standardize the way we introduce tools (and other resources—but mostly tools) to AI models. The idea behind MCP (visualized in Figure 4.13) is actually quite simple: The MCP server is an API server (in Python, JavaScript, or some other language—it doesn't matter) that houses, among other things, definitions for tools and the capacity to execute the tools. The MCP server has API endpoints to list tools, execute tools, and so on. Put another way, MCP is a standardized API format placed in front of the tool definitions and execution code that makes it easier to develop agents across frameworks, programming languages, and businesses.

When an AI agent "wakes up," it reaches out to the MCP servers included in a list that it was given and asks each of those servers which tools it can offer. Each MCP server then tells the agents the names of the functions and the arguments they take. The code to actually execute the code lives on the server, too. Once the AI agent grabs the tool definitions from the server, everything is effectively the same to the LLM—calling tools, writing content, and so on.

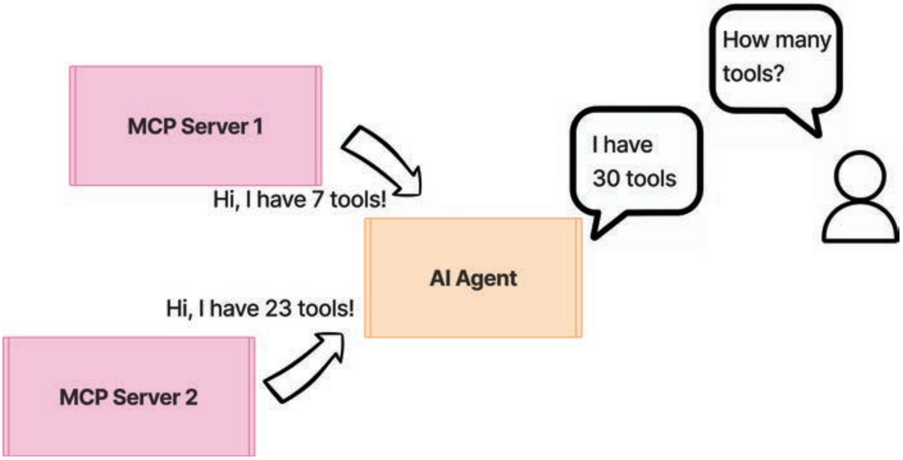


Figure 4.13 An agent is told about one or more MCP servers. On initialization, the agent reaches out to each one and asks which tools it has for it to use. From there, everything is the same as far as the LLM is concerned. It has tools and functions it can decide to call, no matter where they came from.

In summary, MCP is an agreed-upon standard that, if everyone follows it, can facilitate the development of tools for AI systems around the world. A developer in California can write an MCP server for a tool that they made (or didn't!) and put it on GitHub, where a developer in Mumbai can find it and use it. In fact, in this case study, I will write just the first two MCP servers; the third one was created by a third-party team where I had no involvement. I just got to use it for free—how amazing!

There are caveats to MCP, of course. Clearly, tool design and selection are critical, as we saw with our SQL agent. Using the right tool can be efficient and is often a necessary step. Because MCP servers give an AI model access to a set of unknown external tools, however, the challenges will be compounded. The agent will inevitably encounter new tools with descriptions of varying quality. We can mitigate this risk by making sure the descriptions we write for our tools are sufficiently contextful, but even a single bad tool description can send agents down a completely wrong path.

With that warning, let's define our first two MCP servers. These are the MCP servers that I wrote specifically for this case study.

MCP 1: Web Search + Web Crawling

The Python template for creating an MCP server generally has three parts:

- An API route to list the tools available on the server (names, arguments, etc.)
- An API route to accept arguments to execute a tool call
- Logic to call and execute each tool

Listing 4.5 shows an abbreviated code snippet of our first MCP server delivering the ability to Google something (using the Serp API product) and to crawl most websites on the planet (using Firecrawl's web-crawling service as an API to make this easier). Note that we are using the official Python MCP implementation here: https://github.com/modelcontextprotocol/python-sdk. Using this official implementation means that we just have to write the Python functions correlating to listing tools, calling and executing tools, and so on.

Listing 4.5 **Creating the first agent in LangGraph**

```
from mcp.server.models import InitializationOptions
from mcp.server import NotificationOptions, Server
from mcp.types import Tool, TextContent
import mcp.types as types ...

# Import required libraries
from serpapi import GoogleSearch
from firecrawl import FirecrawlApp
# Create server instance
server = Server("research-mcp-server")

def search_with_serpapi(query: str) -> str:
   """Search the web for the query using SerpApi."""
   api_key = os.environ.get("SERP_API_KEY")
   if not api_key:
       return "Error: SERP_API_KEY environment variable is required"

   try:
       search = GoogleSearch({
           "q": query,
           "api_key": api_key,
      ...
      return "\n".join(formatted_results)

   except Exception as e:
       return f"Error performing search: {str(e)}"

def scrape_with_firecrawl(url: str, format_type: str = "markdown") -> str:
   """Scrape a webpage using Firecrawl."""
   api_key = os.environ.get("FIRECRAWL_API_KEY")
   if not api_key:
       return "Error: FIRECRAWL_API_KEY environment variable is required"

   if format_type not in ["markdown", "links"]:
       format_type = "markdown"

   try:
       firecrawl = FirecrawlApp(api_key=api_key)
       ...
       return response if response else "No content found"

   except Exception as e:
       return f"Error scraping URL: {str(e)}"

@server.list_tools()
async def handle_list_tools() -> List[Tool]:
   """List available research tools."""
   tools = []
  tools.append(Tool(
       name="web_search",
       description="Search the web for information using SerpApi. Returns top 3 search
results with titles, snippets, and URLs.",
       inputSchema={
           "type": "object",
              ...
   return tools

@server.call_tool()
async def handle_call_tool(name: str, arguments: Dict[str, Any]) -> List[types.
TextContent]:
   """Handle tool calls for research operations."""

   if name == "web_search":
       query = arguments.get("query")
       if not query:
           return [types.TextContent(type="text", text="Error: Query is required")]
       result = search_with_serpapi(query)
       return [types.TextContent(type="text", text=result)]

   elif name == "scrape_website":
       url = arguments.get("url")
       format_type = arguments.get("format", "markdown")
       result = scrape_with_firecrawl(url, format_type)
       return [types.TextContent(type="text", text=result)]

   else:
       return [types.TextContent(type="text", text=f"Unknown tool: {name}")]

async def main():
   ... code to actually run the server

if __name__ == "__main__":
   asyncio.run(main())
```

That was a lot of code, but don't worry: I won't be showing much more MCP code from here on out (it's all in the book's GitHub). In Listing 4.5, you can see the three main parts there: functions to run each of our two tools, a route to list tools, and a route to accept arguments to execute a tool. With that, we have our first MCP server, giving an AI agent the ability to look things up and visit web pages.

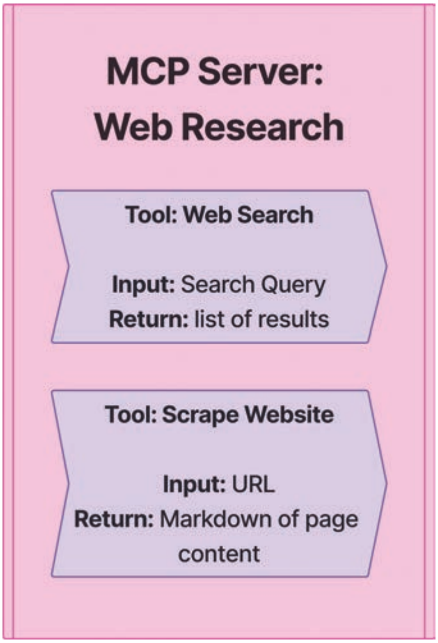


Figure 4.14 Our simple homegrown MCP server has only two tools: web search via the Serp API (Googling something as an API) and web crawling (a service provided by Firecrawl).

Figure 4.14 visualizes our first MCP server with two tools. A lead generation tool will certainly need to look things up and visit web pages. It will also need a system of record to write everything down in.

MCP 2: Hubspot Management

This isn't our first discussion about a system of record for an AI agent. In Chapter 3, we watched an AI system improve at a task over time after we gave it a notepad to write things down in. In this case, we don't really need the agent to get better over time (at least for now), but we should be able to audit what is going on in the CRM. I chose Hubspot for this purpose (feel free to choose a different CRM) mostly because the APIs were easy to pick up. Figure 4.15 shows the five tools I wrote to connect to the Hubspot API:

- **Create contact:** Create a new contact in the CRM.
- **Update contact:** Update an existing contact.
- **Add note to contact:** Add a free text note to a contact.
- **Retrieve notes for contact:** List all notes for a given contact.
- **Fetch contacts:** Use search criteria to grab lists of contacts.

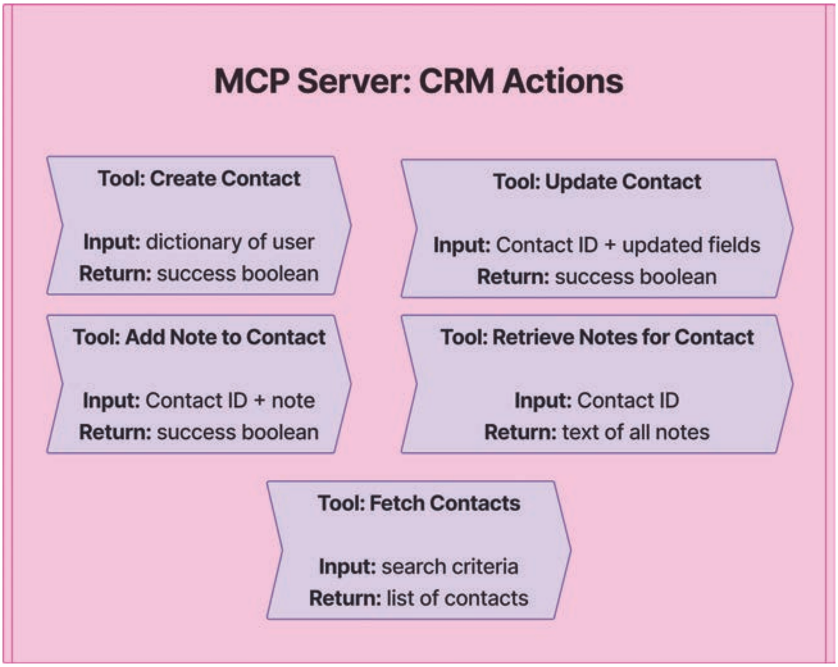


Figure 4.15 Our more complicated homegrown MCP server has five tools for interacting with the CRM so the agents can keep us in the loop about their work.

I did try to find a Hubspot MCP server on the internet, but the few I found either didn't work as written or didn't have the tools I needed (in particular, the ability to create a contact). So, in this case, I asked an AI agent to write an MCP server for me given the first MCP I wrote as an example—and it worked great!

Now we have an agent with seven tools, and Figure 4.16 shows the current state of our lead generator agent. Now, we need to qualify these leads.

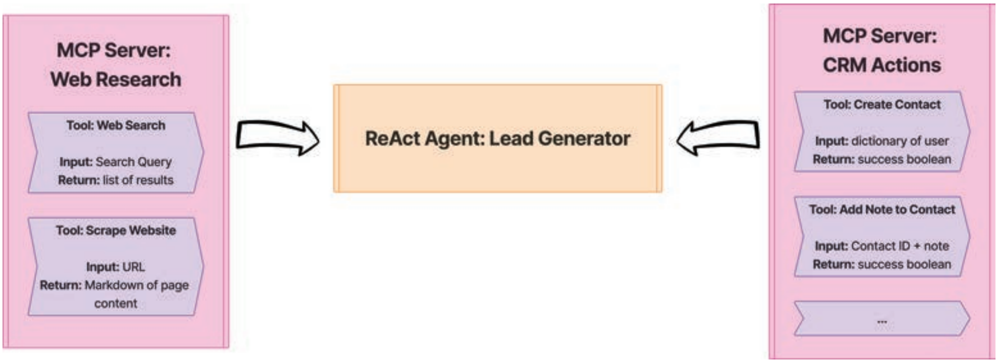


Figure 4.16 The Lead Generator agent gets its seven tools from two MCP servers.

Agent 2: Lead Qualification

Once the lead generation agent adds a new contact to the CRM, the lead qualifying agent (pictured in Figure 4.17) will take over. It will both double-check the work of the lead generation agent and check some more criteria that I added into the special prompt area. At a glance, it might seem as if the lead qualifier agent is redundant given the lead generation agent. They have the same tools and basically the same goal (to identify qualified leads), but the key difference lies in the context of the agent. Chapter 7 formally dives into the idea of **context engineering**—the concept of providing an AI model or agent with the necessary information, context, and tools to perform a task effectively.

The lead generator is given a small set of rules by which to judge candidates and is asked to pull potential leads from the wide world of the open internet. In contrast, the qualifier agent is given both a single person's information and a longer set of requirements to check before moving on to the next stage. In other words, the lead generator has the easier job of pointing a finger at someone and saying, "They seem right"—a task that can be performed by a smaller, cheaper, faster LLM. The cost of a false positive in this case is low because we know a second agent will double-check its results. The lead qualifier agent should be a larger, slower, more expensive LLM because its job is arguably harder: It must go through several pieces of information; read and understand the candidate's syllabus, CV, and other data; and make the final judgment whether the person should receive an email. The cost of a false positive is high here, because I don't want to bother people who aren't good fits for my book.

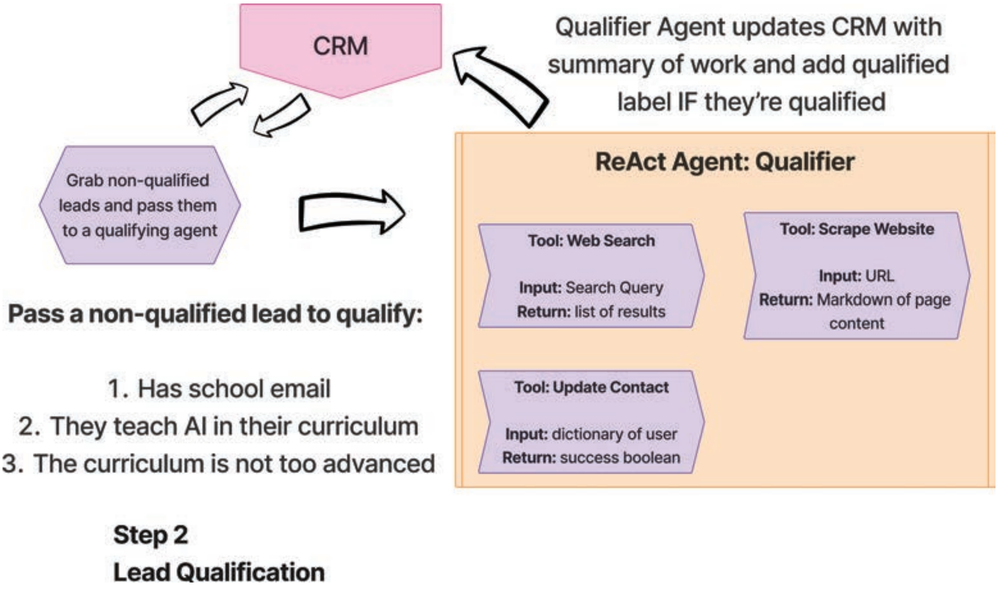

CRM
Qualifier Agent updates CRM with summary of work and add qualified label IF they're qualified
Grab non-qualified leads and pass them to a qualifying agent
ReAct Agent: Qualifier
Tool: Web Search
Input: Search Query
Return: list of results
Tool: Scrape Website
Input: URL
Return: Markdown of page content
Tool: Update Contact
Input: dictionary of user
Return: success boolean
Pass a non-qualified lead to qualify:
1. Has school email
2. They teach AI in their curriculum
3. The curriculum is not too advanced
Step 2
Lead Qualification

Figure 4.17 Our lead qualifying agent will have the same tools as the lead generation agent. It will double-check the lead generator's work while also doing more research.

Like the lead generation agent, the qualifying agent will have some extra information in its system prompt. A snippet of this information can be seen in Prompt 4.2.

PROMPT 4.2 Extra Information in the Lead Qualification Agent Prompt

You are a lead qualifying agent. For a given lead, use the tools provided to make sure that this lead is a good fit for using my book in their classes. If they are a good fit, update the contact to have a lead status of "Open." If they are not a good fit, update the contact to have a lead status of "Unqualified."

Below is a list of qualities that the person must match before marking them as "qualified":

1. They must be teaching in an accredited university or college setting.
2. A recent syllabus for a class they currently teach must reference large language models (LLMs) in some capacity.

…

Figure 4.18 shows that this agent will have the same MCP server access as the lead generation agent. In some cases, it might need to double-check some information online and update information/notes.

At this point, the lead generator has pulled in some leads, and the lead qualifier has double-checked the information and marked the lead as qualifying. Now it's time for a third agent to take over and send the initial cold email.

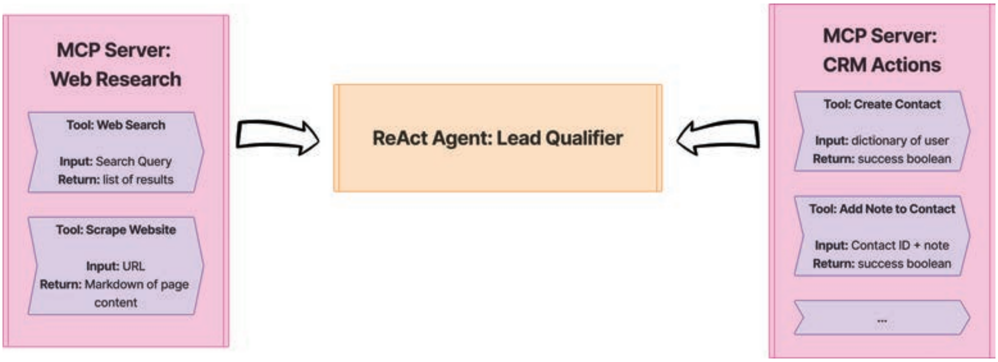


Figure 4.18 Our lead qualifier has the same MCP servers as the lead generator. Its job is to do deeper research on the leads and make sure they are good candidates.

Agent 3: Lead Emailing

The lead emailing agent (Figure 4.19) involves more than just a system prompt change, with a single-shot example (recall from Chapter 1 that a *k*-shot/few-shot prompt means I am placing in-context examples in the prompt to guide the AI) showing how I want the email to roughly sound (as seen in Prompt 4.3). The format of the email in the single-shot email is not as important here because we expect the agent to use a tool via MCP with structured inputs. What's more important is the fact that it writes HTML-encoded emails and subject lines and sends them to the right tool.

PROMPT 4.3 Extra Information in the Lead Emailing Agent Prompt

You are in charge of sending an initial outreach email to a lead. Use the tools provided to look up the information gathered by the other agents, and send an email to the lead with a personalized message. The email should be sent to the lead's email address. When you are done, update the contact to have a lead status of "Connected" so we know that you have sent the email. Here is a sample email:

Subject: "Curious If 'Quick Start Guide to LLMs' Could Be a Fit for Your Course?"

HTML Body: <p>Hi [Name / Dr.]!</p><p>My name is Sinan Ozdemir. I'm an AI author, educator, and the . . .

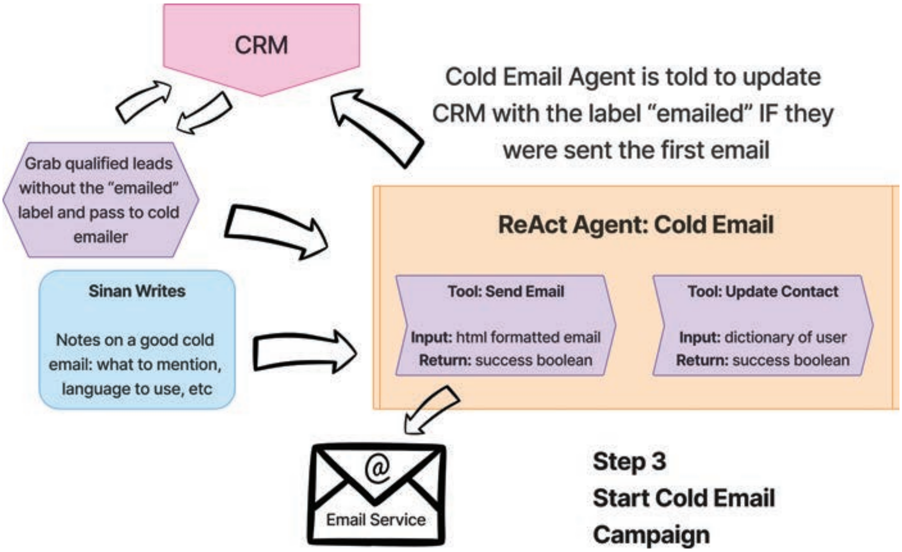

CRM
Cold Email Agent is told to update CRM with the label "emailed" IF they were sent the first email
Grab qualified leads without the "emailed" label and pass to cold emailer
ReAct Agent: Cold Email
Tool: Send Email
Input: html formatted email
Return: success boolean
Tool: Update Contact
Input: dictionary of user
Return: success boolean
Sinan Writes
Notes on a good cold email: what to mention, language to use, etc
@
Email Service
Step 3
Start Cold Email Campaign

Figure 4.19 The lead emailing agent needs to update the CRM, but also requires new capabilities to email leads on my behalf.

A quick note on Agent 3: A decent argument can be made for making this final agent a workflow. For one thing, Agent 3 isn't given much agency. We know the lead is qualified given the previous two agents' work, and all this agent has to do is write the email and call a few APIs in order—that sounds like a predefined pathway. The reason I wanted to make this a true agent over a workflow was simple: I assume this task is easy enough that a system prompt with clear instructions is enough for the agent to follow my instructions clearly. In Chapter 5, we will start to put some of these assumptions to the test. For now, by choosing an agent over a workflow, we are effectively saying either "This task requires the agent to make decisions on the fly that are too complicated/near impossible to code in a predefined pathway" or "This task is easy and straightforward enough that we can save development cycles by simply attaching preexisting MCP servers to an LLM with proper context."

Assuming this job is left up to an agent and not a workflow, this agent will need a way to send emails. MCP comes to the rescue once more.

MCP 3: Resend for Email

If you're not familiar with it, Resend is a developer-friendly email platform with APIs to simplify sending emails at scale. What's even cooler is that it has an official MCP server on its GitHub. Granted, at the time of writing there is just a single tool on that MCP server, but it's the tool I most care about: send-email. I could write my own MCP server for Resend, just as I did with the CRM. However, I want to showcase that our system is agnostic regarding who writes MCP servers and in which language the code launches the official Resend MCP server (written in Typescript, not Python). Our agent will be able to use both it and our homegrown Pythonic MCP servers.

Figure 4.20 shows the topline view of our third agent. The web research MCP has been replaced (because in theory it won't be needed because of the past two agents' work) with our new MCP server, and it is ready to send emails.

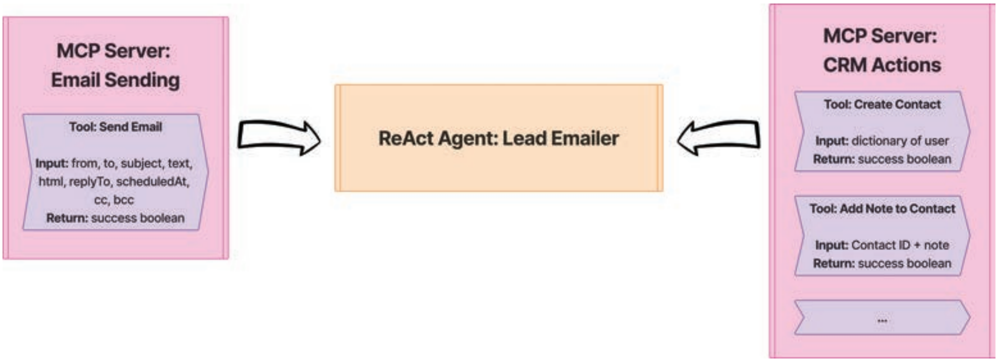


Figure 4.20 Our final MCP server has a single tool and was developed by the same team that created the email sending service Resend.

When to Use a Multi-Agent Versus a Single Agent

Why didn't we just create a single agent with an elongated set of instructions to walk through the whole process of generation, qualifying, and emailing? Certainly, we could have: We could just make that agent wake up and attempt to find and email a new lead. In theory, we could spin up a hundred of them, giving each a different location and industry to focus on.

I chose a multi-agent system in this case study for the following reasons:

- To reduce potential errors in which the agent emailed someone before qualifying them and adding notes to the CRM. In other words, what if the agent "forgets a step" along the way?
- To minimize overlap in duties. What if two agents that aren't talking to each other or are in some race condition end up trying to email the same person at the same time? That lead would not appreciate the double email and would probably be lost.
- To leave room for experimenting with different flavors of LLMs (including LLMs fine-tuned for specific tasks) for each of the different tasks. Maybe GPT-5 is slower but better at emailing, whereas Llama 4-Scout is optimal for quicker lead generation.

It's absolutely true that these tasks *could* be handled by a single agent, but by splitting them up, I gain more control over specific aspects of the funnel. This way, if qualifying is going poorly but everything else is okay, I can tweak and experiment on qualification alone without too much risk to the other portions of the pipeline. With a single agent, every prompt change risks a regression in another area. For this reason, a fair analogy to multi-agent systems would be a micro-service architecture. A change to a single service can be done in much more isolation as long as we are aware of how that single service (agent) fits into the larger picture.

We have now designed and coded three agents, each handling a different portion of the sales process (Figure 4.21). However, we lack a mechanism to test this engine or to keep it running, constantly finding new leads, qualifying them, and emailing them. We will tackle the latter issue in Chapter 5. To gut check the functionality of our agents, though, we should chat with them to see how they do.

Streamlit for Ad-Hoc Testing of the Three Agents

In the codebase for the multi-agent system, I created a visual interface where you can select one of the three agents and chat with it, asking it to perform tasks on demand. Figure 4.22 shows an example in which I asked an agent to list contacts in the CRM and it did so correctly (displaying the two fake contacts Hubspot created during account creation).

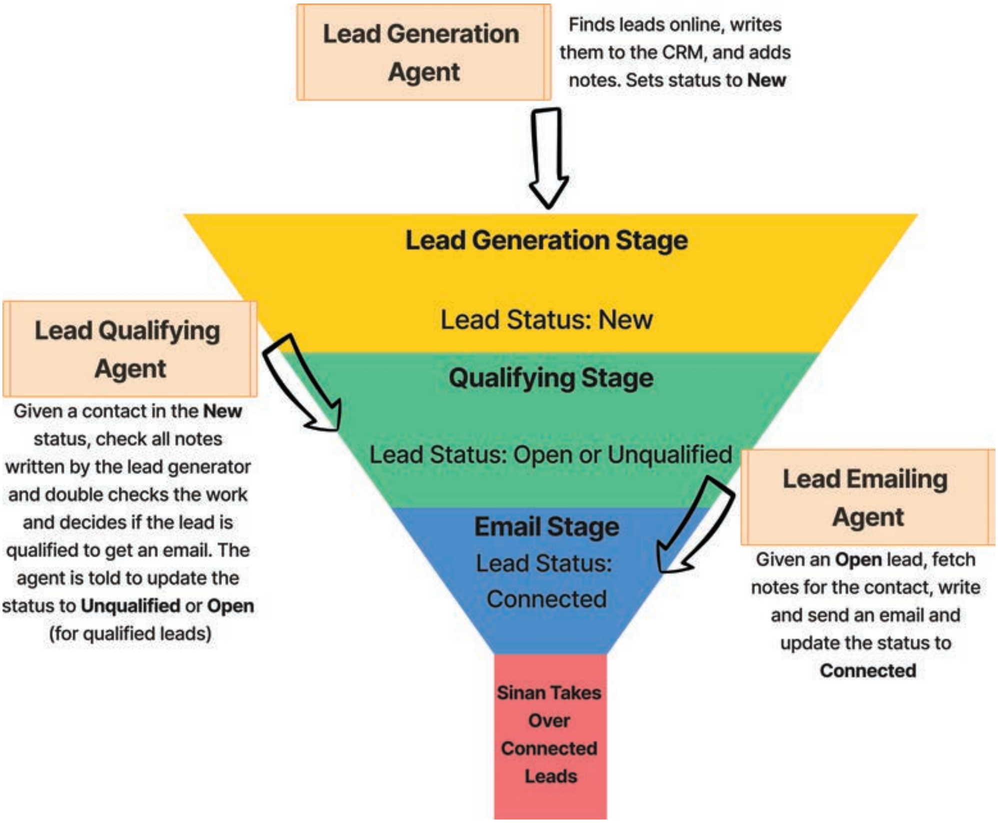
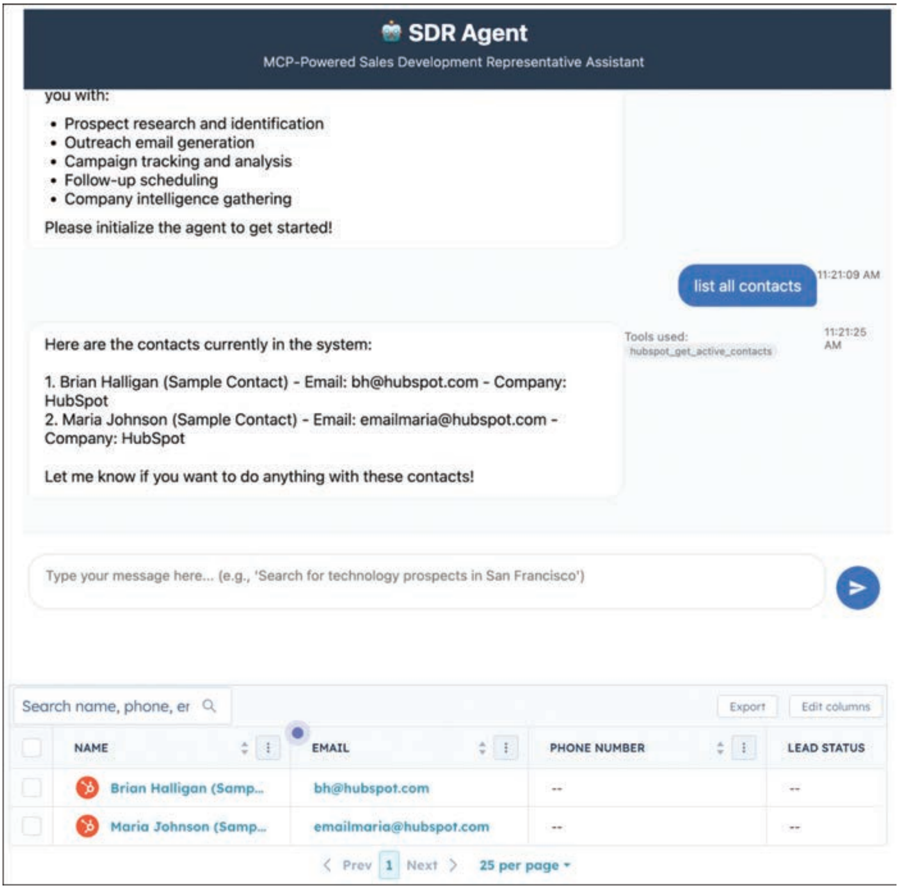

Lead Generation Agent
Finds leads online, writes them to the CRM, and adds notes. Sets status to New
Lead Generation Stage
Lead Status: New
Lead Qualifying Agent
Given a contact in the New status, check all notes written by the lead generator and double checks the work and decides if the lead is qualified to get an email. The agent is told to update the status to Unqualified or Open (for qualified leads)
Qualifying Stage
Lead Status: Open or Unqualified
Lead Emailing Agent
Given an Open lead, fetch notes for the contact, write and send an email and update the status to Connected
Email Stage
Lead Status: Connected
Sinan Takes Over Connected Leads

Figure 4.21 The 10,000-foot view of our multi-agent task delegation. The lead generator will fill the top of the funnel, the qualifying agent will get enough information to be able to email qualified leads, and the emailing agent will send off that first cold email on my behalf.

When I asked the lead generation agent to find a contact at a specific business school, it did. (This example isn't shown here because it would invade that person's privacy.) When I asked the qualifying agent to run on that contact, it changed the status and added a note to confirm the contact's qualification. When I asked the email sending agent to send an email, it did; it then changed the contact's status again, as expected.

At this point, I'm satisfied with the agents' individual performances from basic ad-hoc testing. However, if we are to trust this system at scale, we need to consider a few more factors.



you with:

- Prospect research and identification
- Outreach email generation
- Campaign tracking and analysis
- Follow-up scheduling
- Company intelligence gathering

Please initialize the agent to get started!

list all contacts 11:21:09 AM

Here are the contacts currently in the system:

1. Brian Halligan (Sample Contact) - Email: bh@hubspot.com - Company: HubSpot
2. Maria Johnson (Sample Contact) - Email: emailmaria@hubspot.com - Company: HubSpot

Let me know if you want to do anything with these contacts!

Tools used: hubspot_get_active_contacts 11:21:25 AM

Type your message here... (e.g., 'Search for technology prospects in San Francisco')
Search name, phone, er
Export
Edit columns

	NAME	EMAIL	PHONE NUMBER	LEAD STATUS
□	Brian Halligan (Samp...	bh@hubspot.com	--	--
□	Maria Johnson (Samp...	emailmaria@hubspot.com	--	--

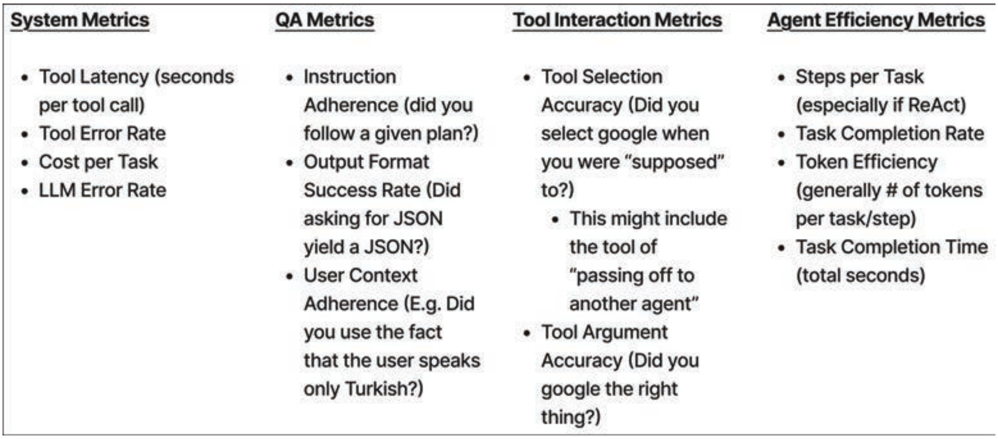
Figure 4.22 A Streamlit app (found in the book's GitHub) showing basic capabilities of the agents for interacting with the CRM.

Evaluating Agents

We've already discussed ways to evaluate agents, including latency, cost, and memory considerations, and especially in comparison to rigid workflows. Figure 4.23 breaks down even more ways to consider evaluating agents in four main categories:

- **System:** Focused on the "behind the scenes" functionality of an agent (e.g., the tool latency, the LLM provider error rate).
- **Quality assurance:** Judging the LLM's ability to adhere to instructions and follow a specific output format. The quality of that output falls into this category as well.

- **Tool interaction:** Specifically zooming in on the agent's tool-calling ability, including whether the agent is calling the right tools for the right tasks and whether it passed the right arguments into the tool.
- **Agent efficiency:** Dependent on the tool interaction and system results to a degree. This category is concerned with questions like how many steps the agent took to come up with the answer, how many tokens it took, and how much it cost.
  - For example, an agent that provides 1,000 tokens of reasoning just to call a single tool could be slower and more expensive than an agent that calls five tools with no thinking in between, even though the latter would be considered "tool inefficient" compared to the former's "token inefficiency."



System Metrics

- Tool Latency (seconds per tool call)
- Tool Error Rate
- Cost per Task
- LLM Error Rate

QA Metrics

- Instruction Adherence (did you follow a given plan?)
- Output Format Success Rate (Did asking for JSON yield a JSON?)
- User Context Adherence (E.g. Did you use the fact that the user speaks only Turkish?)

Tool Interaction Metrics

- Tool Selection Accuracy (Did you select google when you were "supposed" to?)
  - This might include the tool of "passing off to another agent"
- Tool Argument Accuracy (Did you google the right thing?)

Agent Efficiency Metrics

- Steps per Task (especially if ReAct)
- Task Completion Rate
- Token Efficiency (generally # of tokens per task/step)
- Task Completion Time (total seconds)

Figure 4.23 There are dozens, if not hundreds, of metrics we can use to measure the efficacy of agents and multi-agent systems, ranging from low-level tool-based metrics to holistic rubric-based accuracy.

We can also rely on third-party tooling (no pun intended) to help us audit and look back on our agents' past work.

LangSmith for Traceability

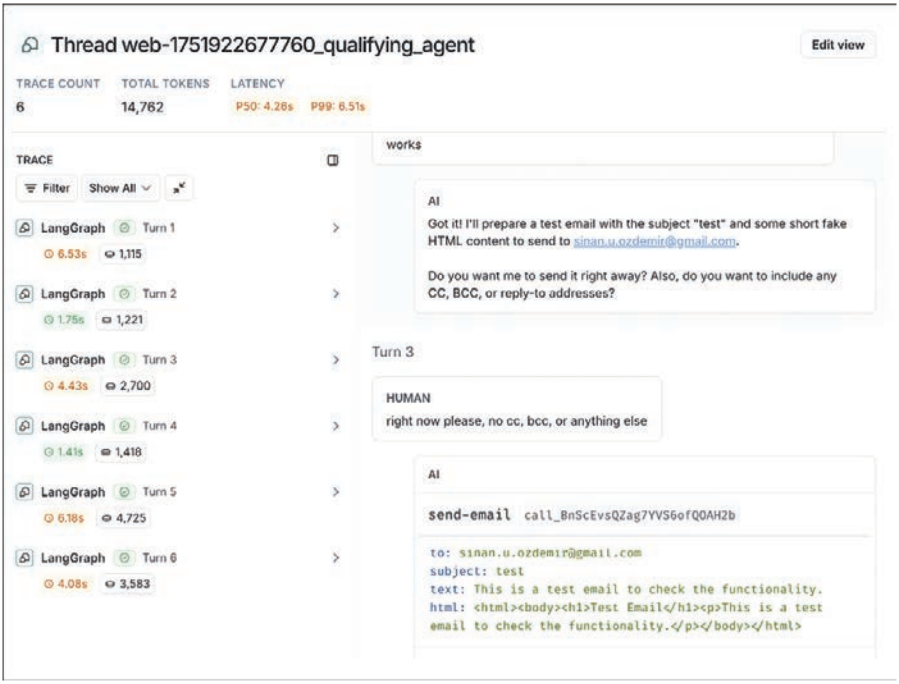
Even if all we need is a way to audit our agents, there's no reason to reinvent the wheel. **LangSmith** (brought to us by the same people who developed LangGraph) is a tracing and observability tool that is purpose-built for LLM workflows. It can track the step-by-step execution of both workflows and agents. LangSmith provides a live dashboard showing every input, output, and tool call, which makes debugging and long-term monitoring a breeze. It also give us the ability to audit past work. There are dozens (at least) of platforms offering observability, evaluations, and more, but LangSmith is free to get started with, simple to set up, and provides immediate value out of the gate. I generally recommend starting with LangSmith even if you are still evaluating other platforms, as it provides an excellent baseline for bare-bones features you should expect from other platforms.

Getting started with LangSmith is a very simple process. All we need to do is set a few environment variables in whatever deployment we are using (even a code notebook), and the SDK will automatically trace everything (see Listing 4.6). We can even split traces by project or use the built-in UI to review results across experiments.

Listing 4.6 **Setting LangSmith keys in the agent environment**

```
-  LANGSMITH_API_KEY="XXX"
- LANGSMITH_TRACING=true
- LANGSMITH_ENDPOINT="https://api.smith.langchain.com"
- LANGSMITH_API_KEY="XXX"
- LANGSMITH_PROJECT="ai-agent"
```

Once the environment variables are set, every agent run (including tool calls, LLM generations, and custom functions) will be recorded in a LangSmith dashboard. This lets us dig into failed runs, analyze tool usage, and share and trace URLs for easier debugging and collaboration. Figure 4.24 shows what an agent trace looks like in the LangSmith user interface.



Thread web-1751922677760_qualifying_agent

TRACE COUNT 6
TOTAL TOKENS 14,762
LATENCY P50: 4.26s P99: 6.51s

LangGraph Turn 1 6.53s 1,115
LangGraph Turn 2 1.75s 1,221
LangGraph Turn 3 4.43s 2,700
LangGraph Turn 4 1.41s 1,418
LangGraph Turn 5 6.18s 4,725
LangGraph Turn 6 4.08s 3,583

works

AI
Got it! I'll prepare a test email with the subject "test" and some short fake HTML content to send to sinan.u.ozdemir@gmail.com.

Do you want me to send it right away? Also, do you want to include any CC, BCC, or reply-to addresses?

Turn 3

HUMAN
right now please, no cc, bcc, or anything else

AI
send-email call_BnScEvsQZag7YVS6ofQOAH2b

```
to: sinan.u.ozdemir@gmail.com
subject: test
text: This is a test email to check the functionality.
html: <html><body><h1>Test Email</h1><p>This is a test
email to check the functionality.</p></body></html>
```

Figure 4.24 An agent trace in the LangSmith dashboard, highlighting each tool call and LLM step. LangSmith offers an easy way to get auditable logs by adding only a few environment variables to the AI system.

In the next few chapters, we will continue to evaluate agents based on the specific work they're asked to do. But for now, let's wrap up this Agents 101 discussion with a recap.

Conclusion

The leap from workflows to agents to multi-agent systems can unlock a whole new set of capabilities for AI applications. Workflows are great choices when you want efficiency, predictability, and full control. Agents bring adaptability, flexibility, and a little bit of creative chaos, which can be a wonderful thing in some cases.

In this chapter, we broke down not just how to build these systems, but also how to evaluate them using both human-driven rubrics and automated tools like LangSmith for traceability and auditability. Although agents can sometimes cost more and take longer, they also offer room for learning, collaboration, and improvement over time. That is especially the case when the agent is given the ability to write, recall, and share evidence with itself (recall our "Otto" example) or with other agents (as in the SDR framework).

Bottom line: There's no single "best" approach to developing systems with AI agents. The right choice depends on your use-case, your tolerance for ambiguity, and how much control you want versus how much flexibility you need. Try both approaches. Experiment, evaluate, and build a system that works for you, and don't be afraid to combine the best parts of each.

In Chapter 5, we'll do just that. We'll create agentic/workflow hybrids and then take these ideas even further, exploring long-running, multi-agent workflows that can handle complex, evolving tasks over time. See you there!

Index