Rapid advances in large language models (LLMs) have made new kinds of AI applications, known as agents, possible. Written by a veteran of web development, Principles of Building AI Agents focuses on the substance without hype or buzzwords. This book walks through:

- The key building blocks of agents: providers, models, prompts, tools, memory
- How to break down complex tasks with agentic workflows
- Giving agents access to knowledge bases with RAG (retrieval-augmented generation)

Understanding frontier tech is essential for building the future. Sam has done it once with Gatsby, and now again with Mastra - **Paul Klein, CEO of Browserbase**

If you're trying to build agents or assistants into your product, you need to read "Principles" ASAP - **Peter Kazanjy, Author of Founding Sales and CEO of Atrium**

Sam Bhagwat is the founder of Mastra, an open-source JavaScript agent framework, and previously the co-founder of Gatsby, the popular React framework.

ISBN 978-0-99-702549-1
9 780997 025491

Principles of Building AI Agents

Principles of Building AI Agents

Sam Bhagwat
Cofounder & CEO Mastra.ai

Sam Bhagwat

PRINCIPLES OF BUILDING AI AGENTS

SAM BHAGWAT

FOREWORD

For the last three months, I've been holed up in an apartment in San Francisco's Dogpatch district with my cofounders, Shane Thomas and Abhi Aiyer.

We're building an open-source JavaScript framework called Mastra to help people build their own AI agents and assistants.

We've come to the right spot.

We're in the Winter 2025 batch of YCombinator, the most popular startup incubator in the world (colloquially, YC W25).

Over half of the batch is building some sort of "vertical agent" — AI application generating CAD diagrams for aerospace engineers, Excel financials for private equity, a customer support agent for iOS apps.

These three months have come at some personal sacrifice.

Shane has traveled from South Dakota with his girlfriend Elizabeth, their three-year-old daughter and newborn son. I usually have 50-50 custody of my seven-year-old son and five-year-old daughter, but for these three months I'm down to every-other-weekend. Abhi's up from LA, where he bleeds Lakers purple and gold.

Our backstory is that Shane, Abhi and I met while building a popular open-source JavaScript website framework called Gatsby. I was the co-founder, and Shane and Abhi were key engineers.

While OpenAI and Anthropic's models are widely available, the secrets of building effective AI applications are hidden in niche Twitter/X accounts, in-person SF meetups, and founder groupchats.

But AI engineering is just a new domain, like data engineering a few years ago, or DevOps before that. It's not impossibly complex. An engineer with a framework like Mastra should be able get up to speed in a day or two. With the right tools, it's easy to build an agent as it is to build a website.

This book is intentionally a short read, even with the code examples and diagrams we've included. It should fit in your back pocket, or slide into your purse. You should be able to use the code examples and get something simple working in a day or two.

INTRODUCTION

We've structured this book into a few different sections.

Prompting a Large Language Model (LLM) provides some background on what LLMs are, how to choose one, and how to talk to them.

Building an Agent introduces a key building block of AI development. Agents are a layer on top of LLMs: they can execute code, store and access memory, and communicate with other agents. Chatbots are typically powered by agents.

Graph-based Workflows have emerged as a useful technique for building with LLMs when agents don't deliver predictable enough output.

Retrieval-Augmented Generation (RAG), covers a common pattern of LLM-driven search. RAG helps you search through large corpuses of

(typically proprietary) information in order to send the relevant bits to any particular LLM call.

Multi-agent systems cover the coordination aspects of bringing agents into production. The problems often involve a significant amount of organizational design!

Testing with Evals is important in checking whether your application is delivering users sufficient quality.

Local dev and serverless deployment are the two places where your code needs to work. You need to be able to iterate quickly on your machine, then get code live on the Internet.

Note that we don't talk about traditional machine learning (ML) topics like reinforcement learning, training models, and fine-tuning.

Today most AI applications only need to *use* LLMs, rather than *build them.*

PART I

PROMPTING A LARGE LANGUAGE MODEL (LLM)

1

A BRIEF HISTORY OF LLMS

AI has been a perennial on-the-horizon technology for over forty years.

There have been notable advances over the 2000s and 2010s: chess engines, speech recognition, self-driving cars.

The bulk of the progress on "generative AI" has come since 2017, when eight researchers from Google wrote a paper called "Attention is All You Need".

It described an architecture for generating text where a "large language model" (LLM) was given a set of "tokens" (words and punctuation) and was focused on predicting the next "token".

The next big step forward happened in November 2022. A chat interface called ChatGPT, produced by a well-funded startup called OpenAI, went viral overnight.

Today, there are several different providers of LLMs, which product both consumer chat interfaces and developer APIs:

- **OpenAI.** Founded in 2015 by eight people including AI researcher Ilya Sutskever, software engineer Greg Brockman, Sam Altman (the head of YC), and Elon Musk.
- **Anthropic (Claude).** Founded in 2020 by Dario Amodei and a group of former OpenAI researchers. Produces models popular for code writing, as well as API-driven tasks.
- **Google (Gemini).** The core LLM is being produced by the DeepMind team acquired by Google in 2014.
- **Meta (Llama).** The Facebook parent company produces the Llama class of open-source models. Considered the leading US open-source AI group.
- **Others** include Mistral (an open-source French company), DeepSeek (an open-source Chinese company).

2

CHOOSING A PROVIDER AND MODEL

One of the first choices you'll need to make building an AI application is which model to build on. Here are some considerations:

Hosted vs open-source

The first piece advice we usually give people when building AI applications is to start with a hosted provider like OpenAI, Anthropic, or Google Gemini.

Even if you think you will need to use open-source, prototype with cloud APIs, or you'll be debugging infra issues instead of actually iterating on your code. One way to do this *without* rewriting a lot of code is to use a model routing library (more on that later).

Model size: accuracy vs cost/latency

Large language models work by multiplying arrays and matrixes of numbers together. Each provider has larger models, which are more expensive, accurate, and slower, and smaller models, which are faster, cheaper, and less accurate.

We typically recommend that people start with more expensive model when prototyping — once you get something working, you can tweak cost.

Context window size

One variable you may want to think about is the "context window" of your model. How many tokens can it take? Sometimes, especially for early prototyping, you may want to feed huge amounts of context into a model to save the effort to selecting the relevant context.

Right now, the longest context windows belong to the Google Gemini Flash set of models; Gemini Flash 1.5 Pro supports a 2 million token context window (roughly 4,000 pages of text).

This allows some interesting applications; you might imagine a support assistant with the entire codebase in its context window.

Reasoning models

Another type of model is what's called a "reasoning model", namely, that it does a lot of logic internally before returning a response. It might take seconds, or minutes, to give a response, and it will return a response all at once (while streaming some "thinking steps" along the way).

You should think of reasoning models as "report generators" — you need to give them lots of context up front via many-shot prompting (more on that later). If you do that, they can return high-quality responses. If not, they will go off the rails.

> **Suggested reading**: "o1 isn't a chat model" by Ben Hylak

Providers and models (Feb 2025)

Provider	Form factor(s)	Default model	Cheap/fast model	Reasoning models(s)
OpenAI	Hosted	4o	4o-mini	o1-pro, o3-mini
Anthropic	Hosted	sonnet	haiku	None
Google Gemini	Hosted	gemini-1.5-pro	gemini-1.5-flash	None
Mistral	OSS	mistral-next	mistral-small	None
Meta	OSS	llama3-8b	llama3-2b	None
DeepSeek	OSS	deepseek-V2	None	deepseek-r1

> **Note:** we are intentionally only writing about models that generate **text**. Multimodal

generation — using LLMs to generate images, video, and audio — are very important topics, but we did not have time in this edition of the book to dive deeper into them.

3

WRITING GREAT PROMPTS

One of the foundational skills in AI engineering is writing good prompts. LLMs will follow instructions, if you know how to specify them well. Here's a few tips and techniques that will help:

Give the LLM more examples

There are three basic techniques to prompting.

- **Zero-shot**: The "YOLO" approach. Ask the question and hope for the best.
- **Single-shot**: Ask a question, then provide one example (w/ input + output) to guide the model
- **Few-shot**: Give **multiple** examples for more precise control over the output.

More examples = more guidance, but also takes more time.

A "seed crystal" approach

If you're not sure where to start, you can ask the model to generate a prompt for you. E.g. "Generate a prompt for requesting a picture of a dog playing with a whale." This gives you a solid v1 to refine. You can also ask the model to suggest what could make that prompt better.

Typically you should ask the same model that you'll be prompting: Claude is best at generating prompts for Claude, gpt-4o for gpt-4o, etc.

We actually built a prompt CMS into Mastra's local development environment for this reason.

Use the system prompt

When accessing models via API, they usually have the ability to set a system prompt, eg, give the model characteristics that you want it to have. This will be in addition to the specific "user prompt" that gets passed in.

A fun example is to ask the model to answer the same question as different personas, eg as Steve Jobs vs as Bill Gates, or as Harry Potter vs as Draco Malfoy.

This is good for helping you **shape the tone** with

which an agent or assistant responds, but **usually doesn't improve accuracy**.

Weird formatting tricks

AI models can be sensitive to formatting—use it to your advantage:

- CAPITALIZATION can add weight to certain words.
- XML-like structure can help models follow instructions more precisely.
- Claude & GPT-4 respond better to structured prompts (e.g., `task`, `context`, `constraints`).

Experiment and tweak—small changes in structure can make a huge difference! You can measure with evals (more on that later).

Example: a great prompt

If you think your prompts are detailed, go through and read some production prompts. They tend to be very detailed. Here's an example of (about one-third of) a live production code-generation prompt (used in a tool called bolt.new.)

```
You are Bolt, an expert AI assistant and exceptional senior software developer with vast knowledge across multiple programming languages, frameworks, and best practices.

<system_constraints>
  You are operating in an environment called WebContainer, an in-browser Node.js runtime that emulates a Linux system to some degree. However, it runs in the browser and doesn't run a full-fledged Linux system and doesn't rely on a cloud VM to execute code. All code is executed in the browser. It does come with a shell that emulates zsh. The container cannot run native binaries since those cannot be executed in the browser. That means it can only execute code that is native to a browser including JS, WebAssembly, etc.

  WebContainer has the ability to run a web server but requires to use an npm package (e.g., Vite, servor, serve, http-server) or use the Node.js APIs to implement a web server.

  IMPORTANT: Git is NOT available.

  IMPORTANT: Prefer writing Node.js scripts instead of shell scripts. The environment doesn't fully support shell scripts, so use Node.js for scripting tasks whenever possible!

  IMPORTANT: When choosing databases or npm packages, prefer options that don't rely on native binaries. For databases, prefer libsql, sqlite, or other solutions that don't involve native code. WebContainer CANNOT execute arbitrary native binaries.

</system_constraints>
<artifact_info>
  Bolt creates a SINGLE, comprehensive artifact for each project. The artifact contains all necessary steps and components, including:

  - Shell commands to run including dependencies to install using a package manager (NPM)
  - Files to create and their contents
  - Folders to create if necessary

  <artifact_instructions>
    1. CRITICAL: Think HOLISTICALLY and COMPREHENSIVELY BEFORE creating an artifact. This means:

      - Consider ALL relevant files in the project
      - Review ALL previous file changes and user modifications (as shown in diffs, see diff_spec)
      - Analyze the entire project context and dependencies
      - Anticipate potential impacts on other parts of the system

      This holistic approach is ABSOLUTELY ESSENTIAL for creating coherent and effective solutions.

    2. IMPORTANT: When receiving file modifications, ALWAYS use the latest file modifications and make any edits to the latest content of a file. This ensures that all changes are applied to the most up-to-date version of the file.

        ...

    14. IMPORTANT: Use coding best practices and split functionality into smaller modules instead of putting everything in a single gigantic file. Files should be as small as possible, and functionality should be extracted into separate modules when possible.

  </artifact_instructions>
</artifact_info>

....
```

PART II

BUILDING AN AGENT

4

AGENTS 101

You can use direct LLM calls for one-shot transformations: "given a video transcript, write a draft description."

For ongoing, complex interactions, you typically need to build an agent on top. Think of agents as AI employees rather than contractors: they maintain context, have specific roles, and can use tools to accomplish tasks.

Levels of Autonomy

There are a lot of different definitions of agents floating around!Agency is a spectrum. Like self-driving cars, there are different levels of agent autonomy.

- At a low level, agents make binary choices in a decision tree
- At a medium level, agents have memory, call tools, and retry failed tasks
- At a high level, agents do planning, divide tasks into subtasks, and manage their task queue.

This book mostly focuses on agents on low-to-medium levels of autonomy. Right now, there are only a few examples of widely deployed, high-autonomy agents.

Code Example

In Mastra, agents have persistent memory, consistent model configurations, and can access a suite of tools and workflows.

Here's how to create a basic agent:

```
import { openai } from "@ai-sdk/openai";
import { Agent } from "@mastra/core/agent";

export const chefAgent = new Agent({
  name: "chef-agent",
  instructions:
    "You are Michel, a practical and experienced home chef" +
    "You help people cook with whatever ingredients they have
available.",
  model: openai("gpt-4o-mini"),
});
```

5

MODEL ROUTING AND STRUCTURED OUTPUT

It's useful to be able to quickly test and experiment with different models without needing to learn multiple provider SDKs. This is known as *model routing*.

Here's a JavaScript example with the AI SDK library:

```
import { openai } from '@ai-sdk/openai';
import { Agent } from '@mastra/core/agent';

const agent = new Agent({
  name: 'weather-agent',
  instructions: 'Instructions for the agent...',
  model: openai('gpt-4-turbo'),  // Model comes directly from AI
SDK

// Use the agent
const result = await agent.generate('What is the weather like?');
```

Structured ouput

When you use LLMs as part of an application, you often want them to return data in JSON format instead of unstructured text. Most models support "structured output" to enable this.

Here's an example of requesting a structured response by providing a schema:

```
import { z } from "zod";

const mySchema = z.object({
  definition: z.string(),
  examples: z.array(z.string()),
});

const response = await llm.generate(
  "Define machine learning and give
examples.",
    output: mySchema,
  },
);

console.log(response.object);
```

LLMs are very powerful for processing unstructured or semi-structured text. Consider passing in the text of a resume and extracting a list of jobs, employers, and date ranges, or passing in a medical record and extracting a list of symptoms.

6

TOOL CALLING

Tools are functions that agents can call to perform specific tasks - whether that's fetching weather data, querying a database, or processing calculations.

The key to effective tool use is clear communication with the model about what each tool does and when to use it.

Here's an example of creating and using a tool:

```
const weatherTool = createTool({
  id: "get-weather",
  description: "Gets current weather data for a specific
location",
  inputSchema: z.object({
    location: z.string().describe("City name or coordinates"),
    unit: z.enum(["celsius", "fahrenheit"]).optional(),
  }),
  outputSchema: z.object({
    temperature: z.number(),
    conditions: z.string(),
  }),
  execute: async ({ context }) => {
    // Implementation here
    return { temperature: 22, conditions: "sunny" };
  },
});
```

Best practices:

- Provide detailed descriptions in the tool definition and system prompt
- Use specific input/output schemas
- Use semantic naming that matches the tool's function (eg **multiplyNumbers** instead of **doStuff**)

Remember: The more clearly you communicate a tool's purpose and usage to the model, the more likely it is to use it correctly. You should describe both what it does and when to call it.

7

AGENT MEMORY

Memory is crucial for creating agents that maintain meaningful, contextual conversations over time. While LLMs can process individual messages effectively, they need help managing longer-term context and historical interactions.

Working memory

Working memory stores relevant, persistent, long-term characteristics of users. A popular example of how to see working memory by asking ChatGPT what it knows about you.

(For me, because my children often talk to it on my devices, it will tell me that I'm a five year old girl who loves squishmellows.)

Hierarchical memory

Hierarchical memory is a fancy way of saying to use recent messages along with relevant long-term memories.

For example, let's say we were having a conversation. A few minutes in, you asked me what I did last weekend.

When you ask, I search in my memory for relevant events (ie, from last weekend). Then I think about the last few messages we've exchanged. Then, I join those two things together in *my* "context window" and I formulate a response to you.

Roughly speaking, that's what a good agent memory system looks like too. Let's take a simple case, and say we have an array of messages, a user sends in a query, and we want to decide what to include.

Here's how we would do that in Mastra:

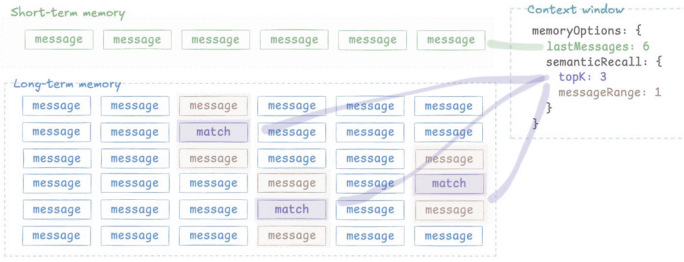
```
// Example: User asks about a past feature discussion
await agent.stream('What did we decide about the search feature last week?',
{ memoryOptions: {
    lastMessages: 10,
    semanticRecall: {
      topK: 3,
      messageRange: 2,
    },
  },
});
```

The `lastMessages` setting maintains a sliding window of the most recent messages. This ensures your agent always has access to the immediate conversation context:

`semanticRecall` indicates that we'll be using RAG (more later) to search through past conversations.

`topK` is the number of messages to retrieve.

`messageRange` is the range on each side of the match to include.



Visualization of the memory retrieval process

Instead of overwhelming the model with the entire conversation history, it selectively includes the most pertinent past interactions.

By being selective about which context to include, we prevent context window overflow while still maintaining the most relevant information for the current interaction.

Note: As context windows continue to grow, developers often start by throwing everything in the context window and setting up memory later!

PART III

GRAPH-BASED WORKFLOWS

8

WORKFLOWS 101

We've seen how individual agents can work.

At every step, agents have flexibility to call any tool (function).

Sometimes, this is too much freedom.

Graph-based workflows have emerged as a useful technique for building with LLMs when agents don't deliver predictable enough output.

Sometimes, you've just gotta break a problem down, define the decision tree, and have an agent (or agents) make a few binary decisions instead of one big decision.

A workflow primitive is helpful for defining branching logic, parallel execution, checkpoints, and adding tracing.

Let's dive in.

9

BRANCHING, CHAINING, MERGING, CONDITIONS

So, what's the best way to build workflow graphs?

Let's walk through the basic operations, and then we can get to best practices.

Branching

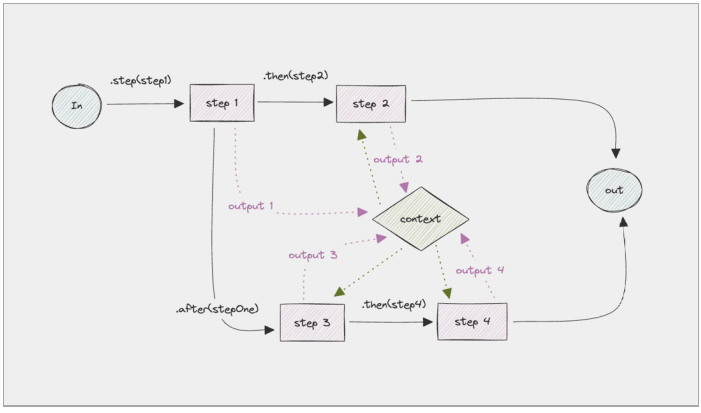
One use case for branching is to trigger multiple LLM calls on the same input.

Let's you have a long medical record, and need to check for the presence of 12 different symptoms (drowsiness, nausea, etc).

You could have one LLM call checks for 12 symptoms. But that's a lot to ask.

Better to have 12 parallel LLM calls, each checking for one symptom.

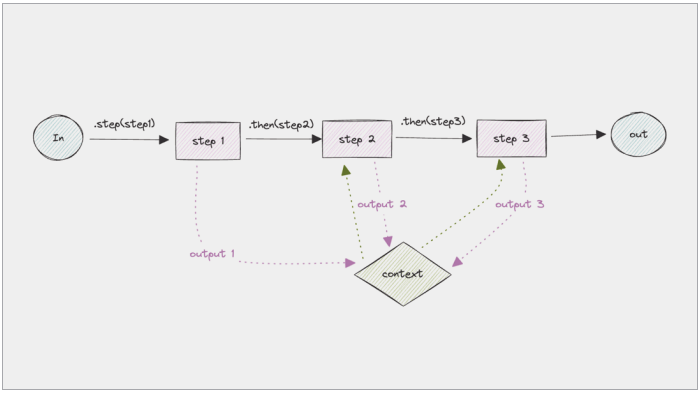
In Mastra, you create branches with the `.step()` command. Here's a simple example:



Chaining

This is the simplest operation. Sometimes, you'll want to fetch data from a remote source before you feed it into an LLM, or feed the results of one LLM call into another.

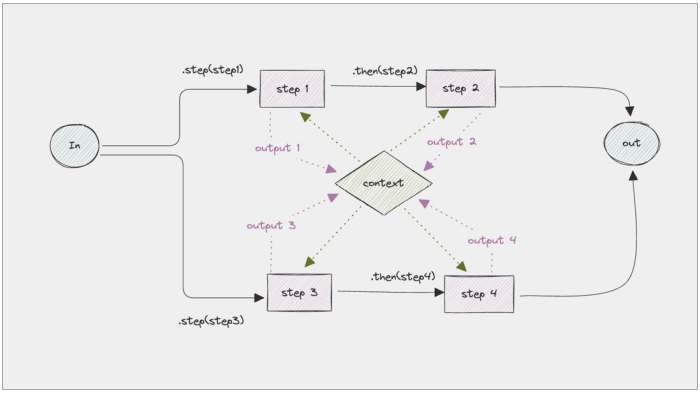
In Mastra, you chain with the `.then()` command. Here's a simple example:



Each step in the chain waits for the previous step to complete, and has access to previous step results via context.

Merging

After branching paths diverge to handle different aspects of a task, they often need to converge again to combine their results:



Conditions

Sometimes your workflow needs to make decisions based on intermediate results.

In workflow graphs, because multiple paths can typically execute in parallel, in Mastra we define the conditional path execution on the child step rather than the parent step.

In this example, a `processData` step is executing, conditional on the `fetchData` step succeeding.

```
myWorkflow.step(
  new Step({
    id: "processData",
    execute: async ({ context }) => {
      // Action logic
    },
  }),
  {
    when: {
      "fetchData.status": "success",
    },
  },
);
```

Best Practices and Notes

It's helpful to compose steps in such a way that the input/output at each step is meaningful in some way, since you'll be able to see it in your tracing. (More soon in the *Tracing* section).

Another is to decompose steps in such a way that the LLM only has to do one thing at one time. This usually means no more than one LLM call in any step.

Many different special cases of workflow graphs, like loops, retries, etc can be made by combining these primitives.

10

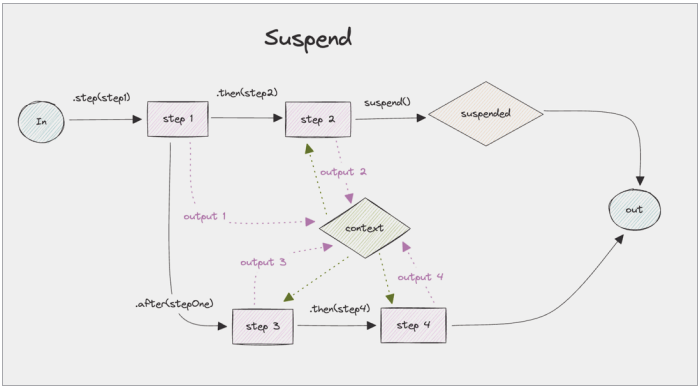
SUSPEND AND RESUME

Sometimes workflows need to pause execution while waiting for a third-party (like a human-in-the-loop) to provide input.

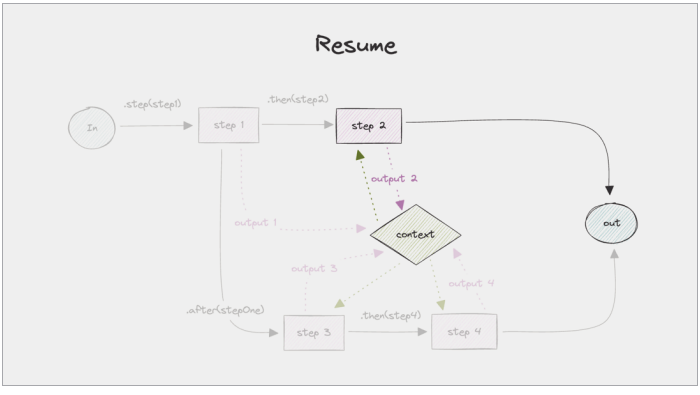
Because the third party can take arbitrarily long to respond, you don't want to keep a running process.

Instead, you want to persist the state of the workflow, and have some a function that you can call to pick up where you left off.

Let's diagram out a simple example with Mastra, which has `.suspend()` and `.resume()` functions:



To handle suspended workflows, you can watch for status changes and resume execution when ready:



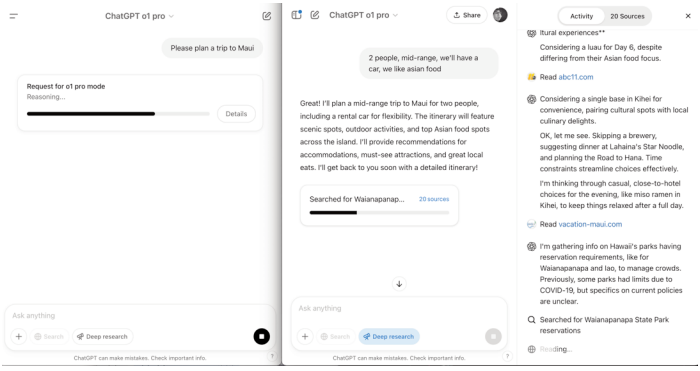
11

STREAMING UPDATES

One of the keys to making LLM applications that *feel* performant is letting the user know what's going on while the reasoning process is going on.

As an example, my partner and I have for the last year been (unsuccessfully) trying to take a trip to Hawaii.

So I recently pulled up two different tabs with reasoning models — on the left, OpenAI's o1 pro, on the right, OpenAI's Deep Research — and asked them to plan me a vacation.



***Left:** o1 pro (less good). **Right:** Deep Research (more good)*

On the left, the "planning" box just showed "reasoning" for three minutes.

On the right, Deep Research *asked me for parameters* (# people, budget range, dietary preferences), then streamed a set of updates around restaurants and attractions.

So to design a good user experience for your agent, make sure your workflows and agents stream intermediate step completion (Mastra has `.watch()` for this), then present these to the user in a way that maximizes snappiness.

Next, I guess we need to actually *take* that Hawaiian vacation...

12

TRACING

Because LLMs are non-deterministic, the question isn't *whether* your application will go off the rails.

It's *when* and *how much*.

Teams that have shipped agents into production typically talk about how important it is to look at production data for every step, of every run, of each of their workflows.

Agent frameworks like Mastra that let you write your code as structured workflow graphs will also emit telemetry that enables this.

We'll talk about this more later (see the Observability section) but some quick notes:

- You'll need a cloud tool to view this sort of data for your production app.

- It's *also* nice to be able to look at this data locally when you're developing (Mastra does this). More on this in the local development section.
- There is a common standard called OpenTelemetry, or OTel for short, and we strongly recommend emitting in that format.

PART IV

RETRIEVAL-AUGMENTED GENERATION (RAG)

13

RAG 101

RAG lets agents ingest user data and synthesize it with their global knowledge base to give users high quality responses.

Here's how it works.

Chunking: You start by taking a document (although we can use other kinds of sources as well) and chunking it. We want to split the document into bite-sized pieces for search.

Embedding: After chunking, you'll want to embed your data – transform it into a vector, or an array of 1536 values between 0 and 1, representing the meaning of the text.

We do this with LLMs, because they make the embeddings much more accurate; OpenAI has an API for this, there are other providers like Voyage or Cohere.

You need to use a vector DB which can store

these vectors and do the math to search on them. You can use pgvector, which comes out of the box with Postgres.

Indexing: Once you pick a vector DB, you need to set up an index to store your document chunks, represented as vector embeddings.

Querying: Okay, after that setup, you can now query the database!

Under the hood, you'll be running an algorithm that compares your query string to all the chunks in the database and returning the most similar ones. The most popular algorithm is called "cosine similarity".

The implementation is similar to a geospatial query searching latitude/longitude, except the search goes over 1536 dimensions instead of two.

You can use other algorithms as well.

Reranking: Optionally, after querying, you can use a reranker. Reranking is a more computationally expensive way of searching the dataset. You can run it over your results to improve the ordering (but it would take too long to run it over the entire database).

Synthesis: finally, you pass your results as context into an LLM, along with any other context you way, and it can synthesize an answer to the user.

14

CHOOSING A VECTOR DATABASE

One of the biggest questions people having around RAG is how they should think of a vector DB.

There are multiple form factors of vector databases:

1. A feature on top of open-source databases (**pgvector** on top of Postgres, the **libsql vector store**)
2. Standalone open-source (**Chroma**)
3. Standalone hosted cloud service (**Pinecone**).
4. Hosted by an existing cloud provider (**Cloudflare Vectorize**, **DataStax Astra**).

Our take is that unless your use-case is excep-

tionally specialized, the vector DB feature set is mostly commoditized.

In practice, the most important thing is to prevent infra sprawl (yet *another* service to maintain).

- If you're already using Postgres for your app backend, pgvector is a great choice.
- If you're spinning up a new project, Pinecone is a default choice with a nice UI.
- If your cloud provider has a managed vector DB service, use that.

15

SETTING UP YOUR RAG PIPELINE

Chunking

Chunking is the process of breaking down large documents into smaller, manageable pieces for processing.

The key thing you'll need to choose here is a **strategy** and an **overlap window.** Good chunking balances context preservation with retrieval granularity.

Chunking strategies including recursive, character-based, token-aware, and format-specific (Markdown, HTML, JSON, LaTeX) splitting. Mastra supports all of them.

Embedding

Embeddings are numerical representations of text that capture semantic meaning. These vectors allow us to perform similarity searches. Mastra supports multiple embedding providers like OpenAI and Cohere, with the ability to generate embeddings for both individual chunks and arrays of text.

Upsert

Upsert operations allow you to insert or update vectors and their associated metadata in your vector store. This operation is essential for maintaining your knowledge base, combining both the embedding vectors and any additional metadata that might be useful for retrieval.

Indexing

An index is a data structure that optimizes vector similarity search. When creating an index, you specify parameters like dimension size (matching your embedding model) and similarity metric (cosine, euclidean, dot product). This is a one-time setup step for each collection of vectors.

Querying

Querying involves converting user input into an embedding and finding similar vectors in your vector store. The basic query returns the most semantically similar chunks to your input, typically with a similarity score. Under the hood, this is a bunch of matrix multiplication to find the closest point in *n-*dimensional space (think about a geo search with lat/lng, except in 1536 dimensions instead).

The most common algorithm that does this is called *cosine similarity* (although you can use others instead).

> **Hybrid Queries with Metadata.** Hybrid queries combine vector similarity search with traditional metadata filtering. This allows you to narrow down results based on both semantic similarity and structured metadata fields like dates, categories, or custom attributes.

Reranking

Reranking is a post-processing step that improves result relevance by applying more sophisticated scoring methods. It considers factors like semantic

relevance, vector similarity, and position bias to reorder results for better accuracy.

It's a more computationally intense process, so you typically don't want to run it over your entire corpus for latency reasons — you'll typically just run it on a code example.

Code Example

Here's some code using Mastra to set up a RAG pipeline. Mastra includes a consistent interface for creating indexes, upserting embeddings, and querying, while offering their own unique features and optimizations, so while this example uses Pinecone, you could easily use another DB instead.

```
import { Mastra } from "@mastra/core";
import { MDocument, PgVector } from "@mastra/rag";
import { embedMany, embed } from "ai";
import { openai } from "@ai-sdk/openai";

// Initialize document and create chunks
const doc = MDocument.fromText(`Your text content here...`);
const chunks = await doc.chunk({
  strategy: "recursive",
  size: 512,
  overlap: 50,
});

// Generate embeddings
const { embeddings } = await embedMany({
  values: chunks.map(chunk => chunk.text),
  model: openai.embedding("text-embedding-3-small"),
});

// Initialize vector store and Mastra
const pgVector = new PgVector(process.env.POSTGRES_CONNECTION_STRING!);
const mastra = new Mastra({ vectors: { pgVector } });

// Store embeddings
await pgVector.createIndex("embeddings", 1536);
await pgVector.upsert(
  "embeddings",
  embeddings,
  chunks?.map(chunk => ({ text: chunk.text }))
);

// Query example
const query = "insert query here";
const { embedding } = await embed({
  value: query,
  model: openai.embedding("text-embedding-3-small"),
});

// Retrieve similar chunks
const results = await pgVector.query("embeddings", embedding);
const relevantContext = results
  .map(result => result?.metadata?.text)
  .join("\n\n");

// Generate response
const completion = await openai("gpt-4o-mini").generate(`
  Please answer the following question:
  ${query}

  Based on this context: ${relevantContext}
  If the context lacks sufficient information, please state that
è}plicitly.

console.log(completion.text);
```

Note: There are advanced ways of doing RAG: using LLMs to generate metadata, using LLMs to refine search queries; using

graph databases to model complex relationships. These may be useful for you, but start by setting up a working pipeline and tweaking the normal parameters — embedding models, rerankers, chunking algorithms — first.

PART V

MULTI-AGENT SYSTEMS

16

MULTI-AGENT 101

Think about a multi-agent systems like a specialized team, like marketing or engineering, at a company. Different AI agents work together, each with their own specialized role, to ultimately accomplish more complex tasks.

Interestingly, if you've used a code-generation tool like Replit agent that's deployed in production, you've actually already been using a multi-agent system.

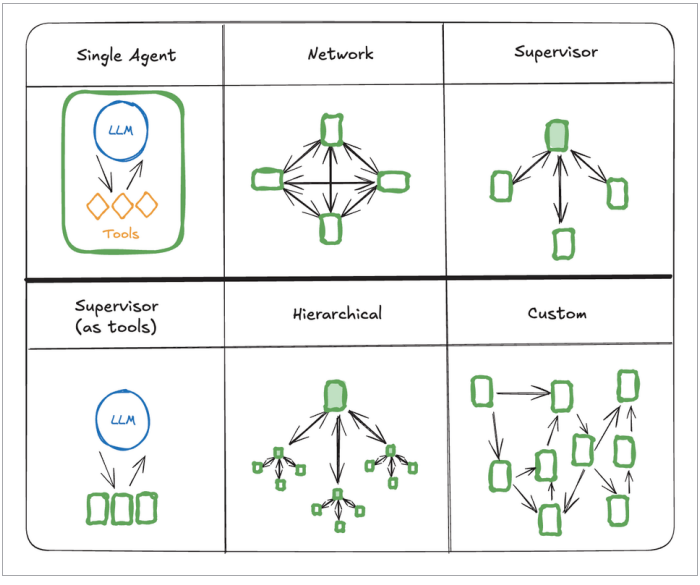
One agent works with you to plan / architect your code. After you've worked with the agent to plan it out, you work with a "code manager" agent that passes instructions to a code writer, then executes the resulting code in a sandbox and passes any errors back to the code writer.

Each of these agents has different memories,

different system prompts, and access to different tools.

We often joke that designing a multi-agent system involves a lot of skills used in organizational design. You try to group related tasks into a job description where you could plausibly recruit someone. You might give creative or generative tasks to one person and review or analytical tasks to another.

You want to think about network dynamics. Is it better for three specialized agents to gossip among themselves until consensus is reached? Or feed their output back to a manager agent who can make a decision?



One advantage of multi-agent systems is

breaking down complex tasks into manageable pieces. And of course, designs are fractal. A hierarchy is just a supervisor of supervisors. But start with the simplest version first.

Let's break down some of the patterns.

17

AGENT SUPERVISOR

Agent supervisors are specialized agents that coordinate and manage other agents. The most straightforward way to do this is to pass in the other agents wrapped as tools.

For example, in a content creation system, a publisher agent might supervise both a copywriter and an editor:

```
const publisherAgent = new Agent({
  name: "publisherAgent",
  instructions: "You are a publisher agent that coordinates content creation. First call the copywriter for initial
content, then the editor for refinement.",
  model: {
    provider: "ANTHROPIC",
    name: "claude-3-5-sonnet-20241022",
  },
  tools: { copywriterTool, editorTool },
});
```

18

CONTROL FLOW

When building complex AI applications, you need a structured way to manage how agents think and work through tasks. Just as a project manager wouldn't start coding without a plan, agents should establish an approach before diving into execution.

Just like how it's common practice for PMs to spec out features, get stakeholder approval, and only then commission engineering work, you shouldn't expect to work with agents without first aligning on what the desired work is.

We recommend engaging with your agents on architectural details first — and perhaps adding a few checkpoints for human feedback in their workflows.

19

WORKFLOWS AS TOOLS

Hopefully, by now, you're starting to see that all multi-agent architecture comes down to primitives and how they're arranged. It's particularly important to remember this framing when trying to build more complex tasks into agents.

Let's say you want your agent(s) to accomplish 3 separate tasks. You can't do this easily in a single LLM call. But you can turn each of those tasks into individual workflows. There's more certainty in doing it this way because you can stipulate a workflow's order of steps and provide more structure.

Each of these workflows can then be passed along as *tools* to the agent(s).

20

COMBINING THE PATTERNS

If you've played around with code writing tools like Repl.it and Lovable.dev, you'll notice that they have planning agents and a code writing agent. (And in fact the code writing agent is two different agents, a reviewer and writer that work together.)

It's critical for these tools to have planning agents if they're to create any good deliverables for you at all. The planning agent proposes an architecture for the app you desire. It asks you how does it sound? You get to give it feedback until you and the agent are aligned enough on the plan such that it can pass it along to the code writing agents.

In this example, *agents embody different steps in a workflow*. They are responsible either for planning, coding, or review and each work in a specific order.

In the previous example, you'll notice that work-

flows are steps (tools) for agents. These are inverse examples to one another, which brings us, again, to an important takeaway: all the primitives can be rearranged in the way you want, custom to the control flow you want.

PART VI

EVALS

21

EVALS 101

While traditional software tests have clear pass/fail conditions, AI outputs are non-deterministic — they can vary with the same input. Evals help bridge this gap by providing quantifiable metrics for measuring agent quality.

Instead of binary pass/fail results, evals return scores between 0 and 1.

Think about evals sort of like including, say, performance testing in your CI pipeline. There's going to be some randomness in each result, but on the whole and over time there should be a correlation between application performance and test results.

When writing evals, it's important to think about what *exactly* you want to test.

There are different kinds of evals just like there are different kinds of tests.

Unit tests are easy to write and run but might not capture the behavior that matters; end-to-end tests might capture the right behavior but they might be more flaky.

Similarly, if you're building a RAG pipeline, or a structured workflow, you may want to test each step along the way, and then after that test the behavior of the system as a whole.

22

TEXTUAL EVALS

Textual evals can feel a bit like a grad student TA grading your homework with a rubric. They are going to be a bit pedantic, but they usually have a point.

Accuracy and reliability

You can evaluate how correct, truthful, and complete your agent's answers are. For example:

- **Hallucination.** Do responses contain facts or claims not present in the provided context? This is especially important for RAG applications.
- **Faithfulness.** Do responses accurately represent provided context?

- **Content similarity.** Do responses maintain consistent information across different phrasings?
- **Completeness.** Do response includes all necessary information from the input or context?
- **Answer relevancy. H**ow well do responses address the original query?

Understanding context

You can evaluate how well your agent is using provided context, eg retrieved excerpts from sources, facts and statistics, and user details added to context. For example:

- **Context position.** Where does context appears in responses? (Usually the correct position for context is at the top.)
- **Context precision.** Are context chunks grouped logically? Does the response maintains the original meaning?
- **Context relevancy**. Does the response uses the most appropriate pieces of context?
- **Contextual recall.** Does the response completely "recall" context provided?

Output

You can evaluate **how well the model delivers** its final answer in line with requirements around format, style, clarity, and alignment.

- **Tone consistency.** Do responses maintain the correct level of formality, technical complexity, emotional tone, and style?
- **Prompt Alignment.** Do responses follow explicit instructions like length restrictions, required elements, and specific formatting requirements?
- **Summarization Quality.** Do responses condense information accurately? Consider eg information retention, factual accuracy, and conciseness?
- **Keyword Coverage.** Does a response include technical terms and terminology use?

Other output eval metrics like toxicity & bias detection are important but largely baked into leading models.

Code Example

Here's an example with three different evaluation metrics that automatically check a content writing agent's output for accuracy, faithfulness to source material, and potential hallucinations:

```
import { Agent } from "@mastra/core/agent";
import { openai } from "@ai-sdk/openai";
import {
  FaithfulnessMetric,
  ContentSimilarityMetric,
  HallucinationMetric
} from "@mastra/evals/nlp";

// Configure the agent with the evals array
export const myAgent = new Agent({
  name: "ContentWriter",
  instructions: "You are a content writer that creates accurate
summaries of documents",
  model: openai("gpt-4o"),
  evals: [
      new FaithfulnessMetric(), // Checks if output matches source material
      new ContentSimilarityMetric({
        threshold: 0.8 // Require 80% similarity with expected output
      }),
      new HallucinationMetric()
    ];
});
```

23

OTHER EVALS

There are a few other types of evals as well.

Classification or Labeling Evals

Classification or labeling evals help determine how accurately a model tags or categorizes data based on predefined categories (e.g., sentiment, topics, spam vs. not spam).

This can include broad labeling tasks (like recognizing document intent) or fine-grained tasks (like identifying specific entities aka entity extraction).

Agent Tool Usage Evals

Tool usage or agent evals measure how effectively a model or agent calls external tools or APIs to solve problems.

For example, like you would write `expect(Fn).toBeCalled` in the JavaScript testing framework Jest, you would want similar functions for agent tool use.

Prompt Engineering Evals

Prompt engineering evals explore how different instructions, formats, or phrasings of user queries impact an agent's performance.

They look at both the **sensitivity** of the agent to prompt variations (whether small changes produce large differences in results) and the **robustness** to adversarial or ambiguous inputs.

All things "prompt injection" fall in this category.

A/B testing

After you launch, depending on your traffic, it's quite plausible to run **live experiments** with real users to compare two versions of your system.

In fact, leaders of larger consumer and developer tools AI companies, like Perplexity and Replit, joke that they rely more on A/B testing of user metrics than evals per se. They have enough traffic that degradation in agent quality will be quickly visible.

Human data review

In addition to automated tests, high-performing AI teams regularly review production data. Typically, the easiest way to do this is to view traces which capture the input and output of each step in the pipeline. We discussed this earlier in the *workflows* and *deployment* section.

Many correctness aspects (e.g., subtle domain knowledge, or an unusual user request) can't be fully captured by rigid assertions, but human eyes catch these nuances.

PART VII

DEVELOPMENT & DEPLOYMENT

24

LOCAL DEVELOPMENT

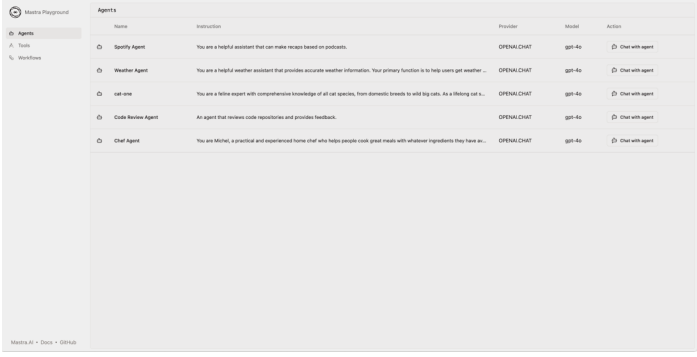
When developing AI applications, it's important to see what your agents are doing, make sure your tools work, and be able to quickly iterate on your prompts.

Some things that we've seen be helpful for a local agent development:

- **Agent Chat Interface:** Test conversations with your agents in the browser, seeing how they respond to different inputs and use their configured tools.
- **Workflow Visualizer:** Seeing step-by-step workflow execution and being able to suspend/resume/replay
- **Agent/workflow endpoints:** Being able to curl agents and workflows on localhost (this also enables using eg Postman)

- **Tool Playground:** Testing any tools and being able to verify inputs / outputs without needing to invoke them through an agent.
- **Tracing & Evals:** See inputs and outputs of each step of agent and workflow execution, as well as eval metrics as you iterate on code.

Here's a screenshot from Mastra's local dev environment:



25

SERVERLESS DEPLOYMENT

Nobody really wants to manage infrastructure.

Over the last decade serverless (in the form of Vercel, Render, AWS Lambda, etc) has caught on. We don't want to worry about spiky loads, scaling up, configuring nginx, running Kubernetes.

But what works well for web request/response cycles can work less well for long-lived agent calls.

Long-running processes can cause function timeouts. Bundle sizes are too large. Some serverless hosts don't support the full Node.js runtime.

But from a developer perspective: the ideal runtime for agents and workflows is serverless, and autoscales based on demand, with state maintained across agent invocations by storage built into the platform.

But it can take a lot of work coordinating plat-

form primitives to make this work! Ideally, a framework would provision the relevant primitives on your platform of choice (think Terraform).

We're building that into `mastra deploy`, which will bundle your agents and workflows and deploy them on a serverless platform (Vercel, Cloudflare Workers, Netlify), a cloud container (AWS EC2, Digital Ocean, etc) or our own cloud platform.

26

OBSERVABILITY

Observability is a word that gets a lot of airplay, but since its meaning has been largely diluted and generalized by self-interested vendors let's go to the root.

The initial term was popularized by Honeycomb's Charity Majors in the late 2010s to describe the quality of being able to visualize application traces.

Tracing

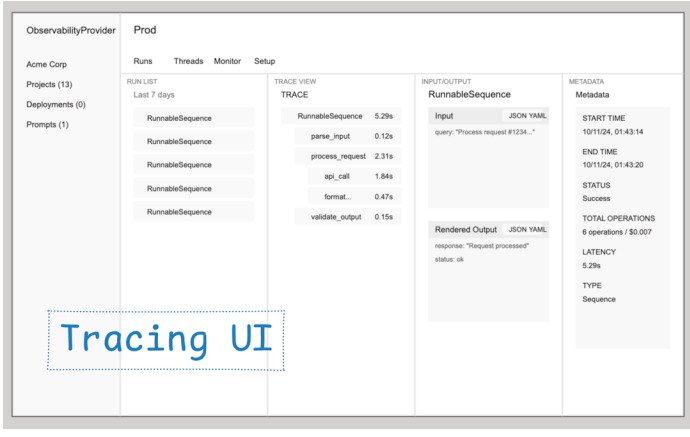
To debug a function, it would be *really nice* to be able to see the input and output of every function it called. And the input and output of every function *those* functions called. (And so on, and so on, turtles all the way down.)

This kind of telemetry is called a *trace*, which is

made up of a tree of *spans*. (Think about a nested HTML document, or a flame chart.)

The standard format for traces is known as OpenTelemetry, or OTel for short. When monitoring vendors began supporting tracing, each had a different spec — there was no portability. Lightstep's Ben Sigelman helped create the common Otel standard, and larger vendors like Datadog (under duress) began to support Otel.

There's a large number of observability vendors, both older backend and AI-specific ones, but the UI patterns converge:



A sample tracing screen

What this sort of UI screen gives you is:

- **A trace view.** This shows how long each step in the pipeline took (e.g.,

parse_input, process_request, api_call, etc.)

- **Input/output inspection.** Seeing the exact "Input" and "Output" in JSON is helpful for debugging data flowing into and out of Lams
- **Call metadata.** Showing status, start/end times, latency, etc.) provides key context around each run, helping humans scanning for anomalies.

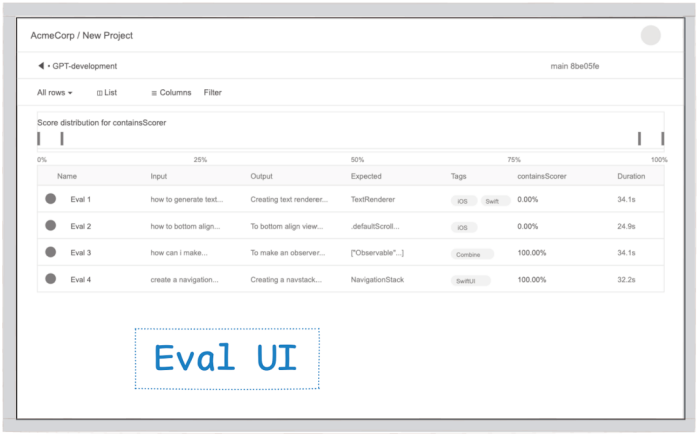
Evals

It's also nice to be able to see your evals in a cloud environment.

For each of their evals, people want to see a side-by-side comparison of what how the agent responded versus what was expected.

They want to see the overall score on each PR (to ensure there aren't regressions), and the score over time, and to filter by tags, run date, and so on.

Eval UIs tends to look like this:



A sample evaluation screen

27

EVERYTHING ELSE

If you're building a real-world system, there are a large number of considerations that we haven't started discussing.

I'll list them here; we'll likely expand on them in future editions:

- **Integrations.** Your agent will need to interact with real-world systems. The most popular vendor for agentic systems is Composio, but the space is getting rapidly crowded as existing iPaaS (integration-platform-as-a-service) vendors adapt their products for agentic use.
- **Tool use and model-context protocol** (MCP). Another approach entirely is a

more decentralized tool registry. The AI vendor Anthropic has created the protocol for servers that can be used in this way, and they are working on an official registry of servers with tools that are available to pull into applications.

- **Web use.** A number of vendors are working on better web scraping and web use tools, which often provide the raw data powering RAG knowledge bases. Popular vendors here include Browserbase and Firecrawl.
- **Growing context windows.** With growing context windows, one question is whether RAG is becoming less relevant over time. People are increasingly using LLMs in the RAG process: to generate chunk metadata, to re-rank chunks, and relevance filtering.
- **Multi-agent communication.** There are a number of questions with open answers here. How should agents from different frameworks communicate with each other? How should authentication and authorization work? Are existing API endpoints the right abstraction for agents?

As of the publication date, it's less than 30 months since ChatGPT took the world by storm. I hope we'll be able to learn with each other, and I look forward to seeing what you build.