OpenClaw, MCP Server, .NET Aspire, BMAD & Spec Kit

CODE

JUL
AUG
2026

codemag.com - THE LEADING INDEPENDENT DEVELOPER MAGAZINE - US $ 8.95 Can $ 11.95

# OpenClaw: Middleware for Your Agents

Cover AI generated

MCP Server Tutorial

Advanced Operations Using .NET Aspire

Professional Grade AI-Assisted Coding

CODE
CONSULTING

# ARE YOU WONDERING HOW ARTIFICIAL INTELLIGENCE CAN BENEFIT YOU TODAY?

## EXECUTIVE BRIEFINGS

Are you wondering how AI can help your business? Do you worry about privacy or regulatory issues stopping you from using AI to its fullest? We have the answers! Our Executive Briefings provide guidance and concrete advise that help decision makers move forward in this rapidly changing Age of Artificial Intelligence and Copilots!

**We will send an expert to your office to meet with you. You will receive:**

1. An overview presentation of the current state of Artificial Intelligence.
2. How to use AI in your business while ensuring privacy of your and your clients' information.
3. A sample application built on your own HR documents – allowing your employees to query those documents in English and cutting down the number of questions that you and your HR group have to answer.
4. A roadmap for future use of AI catered to what you do.

## AI-SEARCHABLE KNOWLEDGEBASE AND DOCUMENTS

A great first step into the world of Generative Artificial Intelligence, Large Language Models (LLMs), and GPT is to create an AI that provides your staff or clients access to your institutional knowledge, documentation, and data through an AI-searchable knowledgebase. We can help you implement a first system in a matter of days in a fashion that is secure and individualized to each user. Your data remains yours! Answers provided by the AI are grounded in your own information and is thus correct and applicable.

## COPILOTS FOR YOUR OWN APPS

**Applications without Copilots are now legacy!**

But fear not! We can help you build Copilot features into your applications in a secure and integrated fashion.

## CONTACT US TODAY FOR A FREE CONSULTATION AND DETAILS ABOUT OUR SERVICES.

**codemag.com/ai-services**

832-717-4445 ext. 9 • info@codemag.com

# Features



# Departments




US subscriptions are US $29.99 for one year. Subscriptions outside the US pay $50.99 USD. Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa, and Discover credit cards are accepted. Back issues are available. For subscription information, send e-mail to subscriptions@codemag.com or contact Customer Service at 832-717-4445 ext. 9.

Subscribe online at www.codemag.com

CODE Component Developer Magazine (ISSN # 1547-5166) is published bimonthly by EPS Software Corporation, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A. POSTMASTER: Send address changes to CODE Component Developer Magazine, 6605 Cypresswood Drive, Suite 425, Spring, TX 77379 U.S.A.

Power Platform
COMMUNITY CONFERENCE

October 27-29, 2026 | MGM Grand, Las Vegas

# THE WORLD'S LARGEST AI+ AGENT BUILDERS CONFERENCE

If you're building with AI—or planning to—**PPCC26 is where you need to be.**




This year's Power Platform Community Conference is set to become the world's largest gathering of engineers, architects, and makers shipping autonomous agents, intelligent workflows, and production-grade AI.

This isn't theory. It's the people doing the work.

Join a community focused on solving real enterprise challenges, sharing what works in production, and advancing how AI gets built. Expect practical conversations, technical depth, and learning you can immediately apply to your systems, architecture, and roadmap.

## WHY BUILDERS ATTEND

Enterprise AI is live and teams are being pushed to move faster, build smarter, and govern what they deploy. PPCC26 is built for that shift:

- **Full-stack focus:** The only event dedicated to the Power Platform + AI stack end-to-end
- **Real practitioners:** Sessions led by builders deploying AI at enterprise scale
- **Deep technical learning:** Architecture, orchestration, and implementation—not surface-level demos
- **Hands-on experience:** Skills you can take straight back to your work
- **Direct access:** MVPs, product experts, and leaders shaping enterprise AI

## BIGGER AND MORE HANDS-ON THAN EVER

More production-focused sessions. More labs. More access. More connection with builders operating at your level.

If you're serious about building with AI—not just experimenting—this is where you sharpen your edge.

**POWERPLATFORMCONF.COM**

# The New Currency of AI: Why Token Discipline Matters More Than Ever

This summer marks a turning point in how developers consume AI tooling. GitHub Copilot's transition from flat-rate subscriptions to usage-based billing, powered by token consumption and GitHub AI Credits, which started on June 1st, is more than a billing change. It's a clear signal that the economics of AI-assisted development are maturing, and that the era of treating AI as "free" is ending. For years, AI coding assistants operated like an all-you-can-eat buffet: pay once and consume without limits. Behind the scenes, every prompt, completion, and interaction always carried real compute cost. That cost was simply hidden. Now, using tokens as both the unit of measure and the basis for billing, developers and organizations must confront an uncomfortable truth: efficiency in AI usage is no longer optional. It's essential.

## The Illusion of "Infinite AI"

The subscription era created an illusion of abundance. Developers experimented freely, accelerating learning and adoption. But it also encouraged waste: long-winded prompts, redundant queries, and repeated context-sending went unpunished because they were invisible. In a usage-based world, every unnecessary token has a price. Scaled across teams and organizations, those small inefficiencies quickly add up to meaningful operational costs. CTOs and COOs are increasingly asking tougher questions—not whether AI delivers value, that's settled—but whether current usage patterns deliver measurable return on investment.

### *The Rise of Tokenmaxxing*

A new term has entered engineering lexicon: *tokenmaxxing*. It describes the pattern of consuming large volumes of tokens with little proportional gain in productivity, quality, or speed. Tokenmaxxing occurs when prompts are written inefficiently, the same context is repeatedly re-sent instead of reused, and AI is called on for tasks where simpler tools suffice. Developers tend to fall into trial-and-error prompting instead of developing a structured interaction. The result is ballooning costs with disappointing business outcomes.

### *The Real Opportunity: Smarter AI Usage*

If token-based billing exposes inefficiency, it also creates a powerful opportunity: to become far more intentional with AI systems. The next competitive advantage will come from using it efficiently.

- **Efficient prompt design.** The gap between a good prompt and a bad one is now measurable in both quality and cost. Concise, well-structured prompts often reduce token usage while improving output. Key practices eliminate ambiguity rather than adding verbosity. They also use structure (JSON, XML, bullet points, constraints, expected output formats), and avoid repeating context unnecessarily.
- **Context reuse and layering.** One large source of token waste is resending the same project architecture, coding standards, or business logic in every interaction. Organizations should move forward by defining persistent project context definitions, shared prompt scaffolds and templates. Emphasis on layered context injection is also a key technique: sending only what each task requires. Think of it as caching for human-AI collaboration.
- **Company-level context engineering.** At the organizational level, companies should invest in reusable, structured knowledge assets that capture things like domain expertise, coding guidelines and standards, security and compliance policies, architectural principles, and more. This shifts AI usage from ad-hoc conversations to disciplined, repeatable workflows.
- **Measure what matters.** Companies should define concrete metrics: time saved per task, reduction in defects, improvements in code quality and maintainability, and deployment frequency and velocity. Without these, token spend remains a cost center. With them, it becomes a strategic investment.

### *Why the Industry Will Follow*

GitHub is unlikely to be the last vendor making this shift. Large language models are expensive to run, and flat subscriptions do not scale sustainably with heavy usage. As more tools adopt usage-based pricing, every organization will face three defining questions: How much are we actually spending? Where is the real value? How do we optimize? This reckoning will spawn new tooling and roles: AI usage dashboards, prompt libraries as first-class assets, internal governance policies. In many ways, this mirrors the evolution of cloud computing a decade ago: first came the gold rush of rapid adoption and "lift-and-shift." Then came FinOps, optimization, and cost discipline. AI is now entering that same maturation phase.

### *From Experimentation to Discipline*

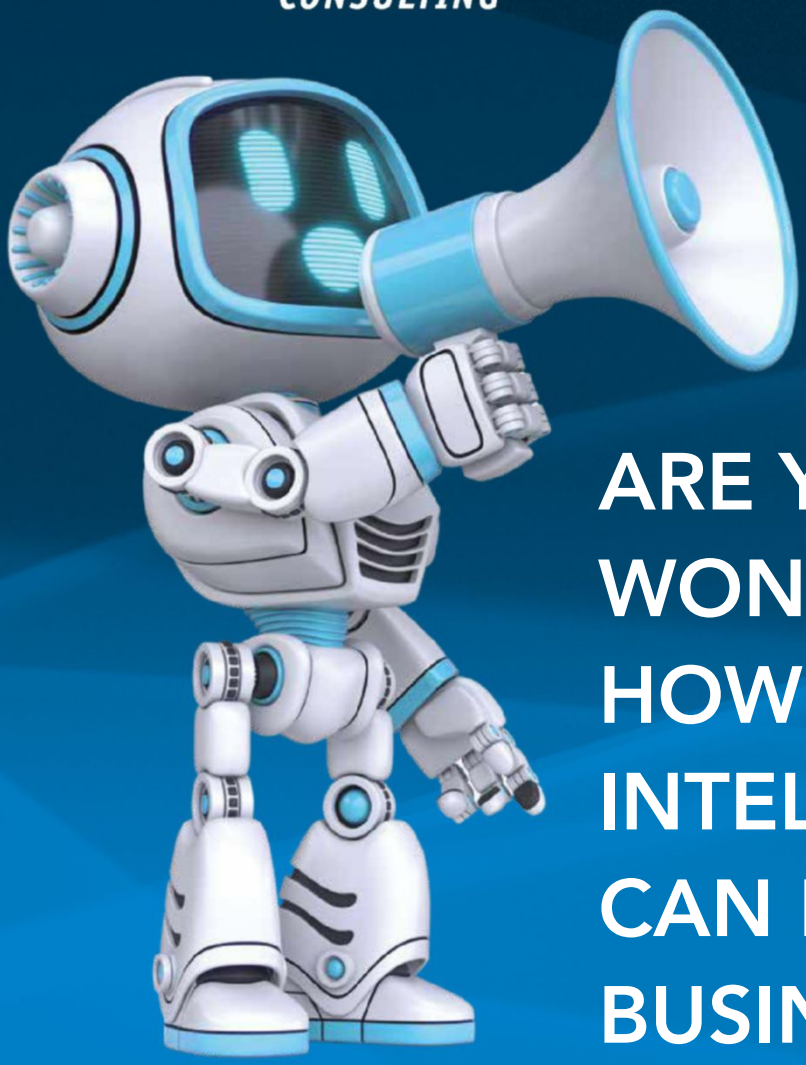
The early days of AI coding assistants were defined by curiosity and free-form experimentation. That phase was necessary and incredibly valuable. But as the cost model changes, our mindset must evolve with it. We are moving from exploration to optimization, from perceived abundance to accountability, and from convenience to craft. This shift elevates AI's importance by pushing us to treat AI with the same engineering rigor we apply to the rest of the stack.

### *Final Thoughts*

Token-based pricing is a mirror. It reflects how effectively we have been using one of the most powerful tools in modern software development. For organizations willing to adapt, the path is clear: first, invest in prompt engineering and AI interaction skills; then, build reusable, structured context systems; finally, measure real business outcomes, not just token volume. In a world where everyone has access to the same powerful models, that mastery will be the ultimate differentiator.

Otto Dobretsberger

CODE CONSULTING



# ARE YOU WONDERING HOW ARTIFICIAL INTELLIGENCE CAN HELP YOUR BUSINESS?

Do you worry about privacy or regulatory issues stopping you from using AI to its fullest?

**We have the answers!**

We will send an expert to your office to meet with you. You will receive:

1. An overview presentation of the current state of Artificial Intelligence.
2. How to use AI in your business while ensuring privacy of your and your clients' information.
3. A sample application built on your own HR documents – allowing your employees to query those documents in English and cutting down the number of questions that you and your HR group have to answer.
4. A roadmap for future use of AI catered to what you do.

**CONTACT US TODAY FOR A FREE CONSULTATION AND DETAILS ABOUT OUR SERVICES.**

**codemag.com/executivebriefing**

832-717-4445 ext. 9 • info@codemag.com

# MCP Server Tutorial: Expose Tools and Resources to AI

**Sahil Malik**
www.winsmarts.com
@sahilmalik

Sahil Malik is an accomplished author and speaker who has published video courses, authored books for numerous publishers, spoken at conferences across the world, and authored for CODE Magazine for many years. In his free time, he likes to do gardening and play with his dogs.

I have written many articles on AI, and it is really interesting to see how the landscape of AI has changed over time. The landscape of AI-driven IT is shifting from traditional support roles toward specialized "architects" of intelligence. As organizations move beyond simple chatbots, these distinct personas have emerged to handle the integration of large language models (LLMs) and agentic workflows. I'll identify a few of these roles.

You have the **retrieval architect** (RAG specialist), who focuses on grounding. They bridge the gap between a "raw" AI model and a company's private data. This professional builds retrieval-augmented generation (RAG) pipelines and has expertise in vector databases (like Pinecone or Milvus), semantic search, and metadata filtering.

You have the **agentic orchestrator**, who sees AI not just as a dummy text generator but as an agent, an actor that can perform valuable tasks in an unattended form. They build systems where AI can use tools, browse the web, or execute code to complete multi-step tasks. This person excels at designing "agentic workflows" using frameworks like LangChain or CrewAI.

You have the **prompt engineer & evaluator** who often comes from a technical writing or QA background. This professional focuses on the interface and reliability. This person is great at optimizing system instructions and creating "golden datasets" to test if model updates break existing logic. This person is great at chain-of-thought prompting and automated evaluation metrics (like RAGAS or BERTScore).

Then you have the **AI infrastructure (AIOps) engineer**, who is the modern evolution of the DevOps role. This person ensures the plumbing for AI is scalable, cost-effective, and secure. They worry about things like managing "GPU-seconds," monitoring token usage/latency, and deploying models on-premises or in private clouds. This person is great at Kubernetes, cloud resource management (AWS/GCP/Azure), and implementing "guardrails" to prevent data leakage.

And finally, there is the **fine-tuning specialist**. While many use off-the-shelf models, this professional specializes in domain-specific deep learning. This person takes a base model and trains it on specialized datasets (e.g., legal, medical, or niche codebases) to improve performance on specific tasks. This person excels at things such as PyTorch/TensorFlow, LoRA (Low-Rank Adaptation), and data curation, etc.

Many AI professionals may have a huge overlap in all these roles. We all wear different hats. But in this article, I am going to focus on the agentic orchestrator persona, who is proficient in API integration, Python, and Model Context Protocol (MCP) to allow models to interact with local files and databases safely.

Specifically, I will dive deep into MCP and explain why and when you'd use it and hopefully make something useful by the end of this article.

## AI as a "Brain in a Vat"

Let's be honest, talking to an AI can sometimes feel like shouting advice to a friend who is locked in a soundproof room. They're brilliant and have read every book ever written, but they can't do anything. If you ask a standard LLM to "Check why the database is slow," it will give you a beautiful, 10-point bullet list of theoretical reasons why databases get tired.

What it won't do is actually look at your database.

But I want AI to solve my problems, not just give me generic advice.

Enter the Model Context Protocol (MCP). Think of MCP as the "USB-C port" for AI. It allows you to plug your tools, your data, and your local environment directly into the AI's reasoning engine.

Let's cook up a realistic problem. Say we have a bunch of production systems running, and like any well-engineered production environment, it generates emergencies. But before it produces an emergency that wakes you up at 3 a.m., it produces alerts. For example, disk is running out of space, or a certain user is now part of 1,000 AD groups, (meaning the user's Kerberos ticket will soon start failing), etc. Typically, you would get this alert and a really

smart person who knows what that alert means knows what to do about it. If a disk is running out of space or a SQL query is taking too long to query, you clean up the disk, examine and rebuild indexes, and so forth.

Now, with AI as a "brain in a vat," you can ask it things like "What do I do when my disk is low on space?" But what if we could make AI smarter? We can give it eyes, hands, and a job to fix. For instance: "I just got Alert 001, disk low space detected. Diagnose and fix it, please."

Could you automate this? Could you write this logic in simple English and use AI to do more complex tasks? Consider something like: "This kind of conditional access policy was violated three times by the same user in the last hour. Query the AAD logs for the past year for this user and establish patterns; then query this user's mailbox for any suspicious outgoing emails; also search OneDrive for external sharing."

What I just described is multiple MCP servers working together. You've given your AI orchestration surface hands and eyes to act on your real data. And in fact, this complex task I just wrote up could be AI-generated.

Let's go step by step. In this article, I am going to build a "production incident assistant"—a realistic MCP server that lets an AI orchestration system fetch simulated system alerts and "reboot" failing services. As far as the AI orchestration system goes, I just made up that term. Really anything that understands the MCP protocol is game. I'll use the Gemini CLI, but feel free to use Claude or Codex or anything.

## An MCP Server

Let's get a basic understanding of the main actor of this article, the MCP server and by extension the model context protocol.

The Model Context Protocol (MCP) is an open-source standard designed to solve the "integration tax" of AI. Instead of writing unique code for every AI model to talk to every tool (GitHub, Google Drive, Slack), MCP creates a universal "USB-C port" for AI.

An MCP server is a lightweight, standalone program that exposes specific data or capabilities from a local or remote system to an AI model. It acts as a translator between the AI's high-level requests and the specific technical requirements (APIs, databases, file systems) of an external tool.

You would use an MCP server when you need an AI to interact with local data, for example read your local codebase, search your documents, or query a local SQLite database. Or when you'd want to perform real-time actions such as post to Slack, create a GitHub issue, or trigger a cloud deployment. Or when you want it to maintain state. Unlike simple API calls, MCP servers can maintain a persistent connection, which is vital for complex, multi-step agentic workflows.

### *MCP Server vs. Skill*

A somewhat similar concept to an MCP server is a skill. An AI skill is a structured set of instructions, knowledge, and tool-access patterns that teaches a model how to perform a specific, repeatable task.

While an MCP server provides the "muscles" (the ability to read files or hit APIs), a skill provides the "brain" (the logic, tone, and step-by-step reasoning). In many modern agentic frameworks, a skill is essentially a packaged "mini-persona" that can be handed to an AI to make it an expert in a narrow domain.

You should build or use a skill when you need the AI to go beyond general chat and follow a strict, high-quality workflow. For example, when you want standardized technical tasks, i.e., you want the AI to always use a specific format for Git commit messages or follow a specific coding style. Or maybe you want to use complex multi-step workflows when a task requires chain-of-thought reasoning, for example, "First, audit the security of this code; second, check for performance bottlenecks; third, summarize findings." Or perhaps you want to do data transformation when you consistently need to turn messy inputs (like raw logs) into structured outputs (like a JSON report). Or maybe you want domain-specific reasoning, giving the AI the "skill" of a senior Java architect so it critiques code with a specific focus on design patterns rather than just syntax.

Think of an MCP server as the kitchen, and the AI skill as the recipe.

### *Components of an MCP Server*

An MCP server involves three main players.

- The **host** is the AI application you are using (e.g., Claude Desktop, Visual Studio Code, or a custom Python script).
- The **client** is a component inside the host that manages the 1:1 connection to a specific server.
- The **server** is a program providing the capabilities. This program should define three primitives: tools, resources, and prompts.

Tools are the actions the model can take (e.g., **calculate_tax()**). Resources are the data the model can read (e.g., **customer_log.txt**), and prompts are pre-defined templates to help the model format its thoughts.

### *Lifecycle of an MCP Server*

The lifecycle is governed by a JSON-RPC 2.0 handshake to ensure both sides are "speaking the same language."

First, you have the **initialization phase**. Here the client sends an initialize request with its protocol version. This is also where the server can communicate instructions back to the client. For example, if you were writing an MCP server around log analytics in Azure, you'd say, "Execute this cheap query first, if that doesn't work, expand to progressively more expensive queries with user confirmation," etc. But at the minimum, during initialization, the server responds with its version and a list of capabilities (what tools/resources it has). The client then sends an initialized notification to confirm the handshake.

Second, you have the **operation phase**. Here, the AI decides it needs a tool. The client sends a **call_tool** request. The server executes the logic and returns the result. This continues as a stateful, back-and-forth conversation.

Finally, you have the **termination phase**. When the session ends, a close method is called. The server performs a "graceful shutdown," cleaning up database connections or temporary files to avoid resource leaks.

Seems pretty logical, doesn't it? Okay, so let's go ahead and build an MCP server.

## Building an MCP Server

With all this background behind us, let's start building our MCP server.

As I mentioned earlier, I am going to build a "production incident assistant"—a realistic MCP server that lets an AI orchestration system fetch simulated system alerts and "reboot" failing services.

Let's understand the three main characters of this movie.

First is the host. In this case, the Gemini CLI. It's the brain. It decides what needs to happen. When you say something like "Such and such problem happened" it is Gemini CLI that understands that request and routes it to one of many registered MCP servers, in our case our MCP server.

Second is the server itself. This is the Node.js app I am about to build. It holds the tools (actions) and resources (data).

Finally, I'll use stdio as the transport. It's simple, it's fast, and it doesn't require setting up complex networking. The CLI just starts your script and talks to it via standard input/output. You could think of many other interesting surfaces or transports, but I'll stick with simple for this article.

### *The Basic Project Setup*

You could write the MCP server in any language you prefer. I'll use Node.js for fun. Although if you are inclined to do so, see if you can write one in Python too. It's not too different.

First, make sure you have the latest LTS version of Node.js installed from Nodejs.org. I am writing this article using version 22.17.0.

Next, go ahead and create a folder called **incident-assistant**, and in this project initialize a node project using the shell commands as shown below.

```
mkdir incident-assistant
cd incident-assistant
npm init -y
```

Go ahead and install a couple of dependencies and a dev dependency.

```
npm install @modelcontextprotocol/sdk zod
npm install -D typescript @types/node
```

The **@modelcontextprotocol/sdk node package** is the standard TypeScript SDK that implements the full MCP specification, making it easy to create MCP servers that expose resources, prompts and tools, build MCP clients that can connect to any MCP server, and use standard transports like stdio and Streamable HTTP. This SDK has a required peer dependency on **zod** for schema validation.

Next, run the following command.

```
npx tsc --init
```

When you run **npx tsc --init**, you are telling the TypeScript compiler to bootstrap a new project. It is the standard way to transition a project from "plain JavaScript" to "managed TypeScript." The primary action is the generation of a tsconfig.json file in your current directory. This file is the "brain" of your TypeScript project. Without it, the compiler treats files in isolation; with it, the directory is treated as a TypeScript project.

Using npx ensures you are using the version of the TypeScript compiler (tsc) installed in your local **node_modules**. If you don't have it installed locally, npx will download a temporary version to create the file. This prevents "version mismatch" issues where your global TypeScript version is different from the one your project requires.

Go ahead and open the incident-assistant project in Visual Studio Code. You should see a heavily commented tsconfig.json with some smart defaults and a package.json with our specified dependencies and dev dependencies.

I also added a .gitignore and pushed it to GitHub at https://github.com/maliksahil/mcp-incident-assistant. You can see my code for this commit at https://github.com/maliksahil/mcp-incident-assistant/tree/59f3bf8497cc396556be4cf0077c9a06b7d78bf1.

Now we are going to start writing the main details of our MCP server. We're going to build a server that manages several services. We'll give it the ability to list alerts and execute a fix.

I am going to separate this problem into three parts.

1. First, I will write the core boilerplate.
2. Second, I will write the resource, specifically the data our server can read.
3. And finally, I will write the tool, the action our MCP server can perform.

### *The Core Logic*

Starting with the core boiler plate. In your TypeScript project, create a src/index.ts file. This is where the magic (and the bugs) will live. You can find the code for the core logic in **Listing 1**. The logic is quite straightforward. We initialize our MCP server and we give it some fake data to work with. In a real-world application, you'd connect to your business systems here, such as connection strings, sql database information, or your internal APIs, or whatever you are writing this MCP server for.

While we are at it, go ahead and tweak your tsconfig.json as well.

We are going to rely on node types for a few operations, so modify the types line as follows.

```
"types": ["node"],
```

Change the verbatimModuleSyntax to false so our imports work.

```
"verbatimModuleSyntax": false,
```

And finally instruct our TS compiler to read from the **src** folder and drop the build in the **dist** folder as shown below.

```
"outDir": "dist",
"rootDir": "src",
```

If you are interested in following along the full set of code changes at this point, you can see the pull request at https://github.com/maliksahil/mcp-incident-assistant/pull/1.

### *Registering a Resource*

Resources are things the AI can read. We use **registerResource** to expose our current alerts. You can find the code for registering our resource in **Listing 2**.

As you can see in **Listing 2**, think of this code as giving your AI host a library card to a very specific, private shelf in your digital office. In plain English, you are telling the AI: "Hey, if you ever need to know what's going wrong with the system, go to this specific 'address' and I'll hand you a list of active alerts."

There are four important parts to this code.

First, the name: **active-alerts**. This is the internal ID for the resource. It's how the server keeps track of what you've registered.

Second, the address: **resource://incidents/active**. Just like a website has a URL (https://...), MCP data has a URI. This is the *location* your AI will look at when it wants to find this specific data. It doesn't live on the internet; it lives inside your server's logic.

Third, is the instruction manual (metadata). This bit tells AI why it should care about this data. The **description** field is actually the most important part for the AI. Gemini or Claude would read this to decide: "Hmm, the user asked about system health... oh, I see a resource described as 'Provides a list of currently active system alerts.' I should probably read that!"

Finally the **mimeType**, which tells the system the data is formatted as JSON, so it knows how to parse the text.

When all this information is collected, it is time for the "delivery guy" (the function), which can be seen as the **async (uri) => { ... }** part is the actual workhorse. It's a function that stays "asleep" until AI "clicks" the link to that resource. When triggered, it takes your MOCK_ALERTS (the fake data we made earlier), does JSON.stringify, which turns that data into a long string of text and content, packs that text into a format the protocol understands, and ships it off to Gemini or Claude's brain.

If you are interested in following along the full set of code changes at this point, you can see the pull request at https://github.com/maliksahil/mcp-incident-assistant/pull/2.

**Listing 1:** The core logic

```
import { McpServer }
    from "@modelcontextprotocol/sdk/server/mcp.js";
import { StdioServerTransport }
    from "@modelcontextprotocol/sdk/server/stdio.js";
import { z } from "zod";

// Initialize the server with some personality
const server = new McpServer({
    name: "IncidentResponsePro",
    version: "1.2.0",
});

// Fake data because real production data is scary
const MOCK_ALERTS = [
    { id: "ALRT-001", service: "auth-api",
      status: "CRITICAL", message: "Memory leak detected" },
    { id: "ALRT-002", service: "payment-gateway",
      status: "WARNING", message: "Latency > 500ms" },
];
```

**Listing 2:** Registering a resource

```
server.registerResource(
  "active-alerts",
  "resource://incidents/active",
  {
    description:
      "Provides a list of currently active system alerts",
    mimeType: "application/json"
  },
  async (uri) => ({
    contents: [{
      uri: uri.href,
      text: JSON.stringify(MOCK_ALERTS, null, 2),
    }],
  })
);
```

### *Registering a Tool*

Tools are things the AI can do. This tool (**Listing 3**) will simulate "restarting" a service. Note how we use zod to ensure Gemini doesn't try to reboot "the moon" or "your mom."

If the resource code we looked at earlier gave AI "eyes" to see your data, this tool code gives AI "hands" to actually do work.

In plain English, you are telling the AI: "If you decide a service needs a restart, here is the specific button you press, the information I need from you to press it, and what will happen when you do."

Here is how specifically this "digital button" is built.

First is the name: **resolve_incident**, which is the name of the command. When AI is thinking, it sees this in its list of available actions, similar to how you see a list of apps on your phone.

Second is the instruction manual, the metadata. This tells AI how and when to use the tool. The title and description explain to the AI what the tool does. AI reads this and thinks: "The user wants to fix the API... oh, I have a tool called **Service restart** that attempts a fix. I should use that!" The inputSchema defines the "form" AI must fill out. It tells the AI: "If you want to use this tool, you must provide a serviceName and a reason." The .describe() parts are clues for the AI so it knows exactly what to type

into those fields. The execution (the "work"), which is the **async ({ serviceName, reason }) => { ... }** block is what actually happens on your computer when AI clicks the digital button.

> We use console.error, so it prints a log to your terminal so you can see what the AI is doing. Never use console.log since that will interfere with the JSON-RPC protocol.

**Listing 3:** Register a tool

```
server.registerTool(
  "resolve_incident",
  {
    title: "Service restart",
    description:
    "The name of the service to restart (e.g., 'auth-api')",
    inputSchema: {
      serviceName: z.string().describe(
    "The name of the service to restart (e.g., 'auth-api')"),
      reason: z.string().describe(
         "A brief explanation for the audit log
           of why this fix is being applied"),
    },
  },
  async ({ serviceName, reason }) => {
    console.error(
      `[AUDIT] AI is attempting to fix
       ${serviceName}. Reason: ${reason}`);

    // Simulating a realistic 80% success rate
    const success = Math.random() > 0.2;

    if (success) {
      return {
        content: [{ type: "text",
          text:
`SUCCESS: ${serviceName} has been gracefully restarted.` }]
      };
    }

    return {
      content: [{
         type: "text",
         text: `FAILURE:
     Could not restart ${serviceName}.
     Please check logs manually.` }],
       isError: true
    };
  }
);
```

**Listing 4:** Startup logic

```
// The actual startup logic
async function main() {
  const transport = new StdioServerTransport();
  await server.connect(transport);
  console.error("IncidentResponsePro is standing by.");
}

main().catch((err) => {
  console.error("Fatal error during startup:", err);
  process.exit(1);
});
```

Then we use a random number to simulate a real-world scenario where restart might fail. We have put in an 80% success rate. In a real app, this is where you'd put code to talk to a server or cloud provider. But for a demo this is good enough.

Finally, the feedback loop where the code sends a message back to the AI works in one of two ways:

- **SUCCESS:** The AI gets a text confirmation and can tell you, "I've restarted it for you!"
- **FAILURE:** AI gets an error message and can say, "I tried to fix it, but something went wrong; you should probably take a look."

If you are interested in following along the full set of code changes at this point, you can see the pull request at https://github.com/maliksahil/mcp-incident-assistant/pull/3.

### *Add the Startup Logic*

All that is left to be done is to add the starting up logic, which can be seen in **Listing 4**.

If the resources were the AI's "eyes" and the tools were its "hands," this bit of code is the "phone line" and the "power switch" that turns the whole operation on.

In plain English, you are telling your computer: "Open up a communication channel, connect my AI assistant to it, and let me know when it's ready to start taking orders."

Here, first we create a new pipe using new StdioServerTransport(). MCP servers need a way to send and receive messages. Stdio (Standard Input/Output) creates a virtual pipe. When the Gemini CLI sends a command, it goes into the pipe's input. When your server responds, it pushes data out the pipe's output. It's the simplest, fastest way for two programs on the same computer to talk to each other.

Next is the handshake: await server.connect(transport), which is when the "brain" meets the "pipe." It takes all those tools and resources you defined earlier and hooks them up to the communication channel. It's like plugging a telephone into the wall jack. Once this line finishes, your server is officially listening for Gemini to ask it a question.

Next is the status light: console.error(...). You'll notice we use console.error again and not console.log. Because the "pipe" (stdout) is busy sending technical JSON-RPC data to Gemini, we use the "error" channel (stderr) to talk to you, the human.

This prints a nice message in your terminal so you know the server didn't just crash—it's actually running and waiting.

I know this is counterintuitive for console.error to show a non-error message, but it is what it is.

Finally, the safety net: main().catch(...). Hey, computers are unpredictable. Sometimes a port is blocked or a file is missing. This part is the "emergency brake." If the server fails to start for any reason, it catches the error, prints exactly what went wrong to your screen, and process.exit(1)

tells the computer to shut the program down immediately rather than let it sit there "broken" and waste memory.

If you are interested in following along the full set of code changes at this point, you can see the pull request at https://github.com/maliksahil/mcp-incident-assistant/pull/4.

## Building the MCP Server

Now we need to connect the brain to the muscles. This is a matter of building our MCP server so it is in an executable form. In our case that would be a .js file. You can also have hosted MCP servers that are hosted as an API, or simply specify that an MCP server runs locally as a Node.js file, or a Python or Go file, etc.

To build our server, run the following command.

```
npx tsc
```

Since we have already configured our tsconfig.json to put the built version in the **dist** folder, you should now see a dist folder in your project with the built version of the code (**Figure 1**).

Now we need to configure our CLI. Specifically, our AI host, Gemini CLI in this case, needs to be told what MCP servers it can use and where they live. To do so, edit the ~/.gemini/settings.json file, and add the following snippet.

```
"mcpServers": {
  "incident-bot": {
    "command": "node",
    "args": [
      "/..fullpathto../dist/index.js"
    ]
  }
}
```

A few things to be careful of here.

- You may not have a ~/.gemini/settings.json file; if it doesn't exist, create it.
- Second, you may already have MCP servers in that list. If you do, add incident-bot into that list rather than replacing the full list.
- Finally, in args, ensure you specify the full path to the index.js file you just built.

That's it. It's time to take this MCP server for a spin.

## Running the MCP Server

Go ahead and launch Gemini CLI. Once Gemini CLI is launched, run the following command to verify that your MCP server is available:

```
/mcp list
```

It should produce an output like **Figure 2**.

Excellent! Now give it a prompt as below.

> Are there any critical alerts? If so, try to fix them and let me know the result.

Feel free to tweak the input as you desire. You are talking to AI after all; you should be able to talk in plain English. Imagine if you had another MCP server that could tell you names of services? Then you could say, "For all my services, check active alerts..." In fact, why would you even need to type this? Couldn't you just automate that with an agent? Let's leave that for some other day.

Let's try the above prompt.

Now I'm using the free version of Gemini CLI, and anecdotally I can tell you that the results I get when using Claude with the same MCP server on a paid plan are much faster and better. Feel free to try that on your own dime though. The results from Gemini are somewhat similar.

Gemini starts by understanding what we mean by our command. As **Figure 3** shows, the first thing it did was to look in the current folder.

This is an important point here. When you start Gemini or Claude, you are trusting that folder. I usually create a folder called "gemini" or "claude" on my machine, and in that I specify sandbox settings, so it doesn't keep prompting me for what I consider safe. But I certainly do not give these CLIs carte blanche access to my whole disk. That would be a bad idea.

Next, Gemini uses **cli_help** to try and determine if our **mcp_incident_bot** tool can be of any help. This can be seen in **Figure 4**. Note that since I'm on a freebie version of Gemini, things are not only slow they also frequently time out and retry. I guess it's free so I shouldn't complain too much.

Well, it looks like that request timed out, so now Gemini is trying alternate sources of information such as Google web search or parent directory, and then it returns to

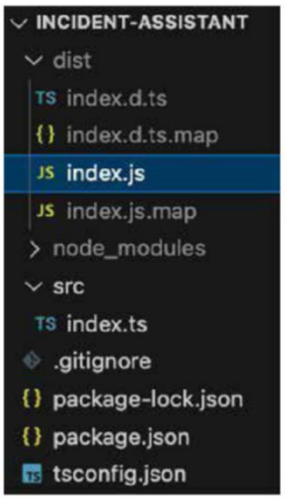



**Figure 1:** Our built MCP server


**Figure 2:** Our MCP server is visible


**Figure 3:** Gemini trying to figure out what we mean

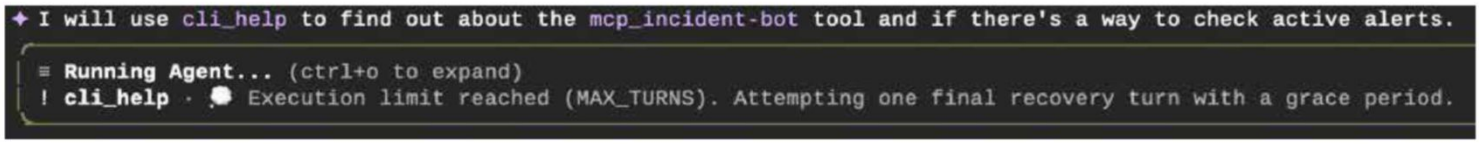



**Figure 4:** Gemini looking at our MCP server

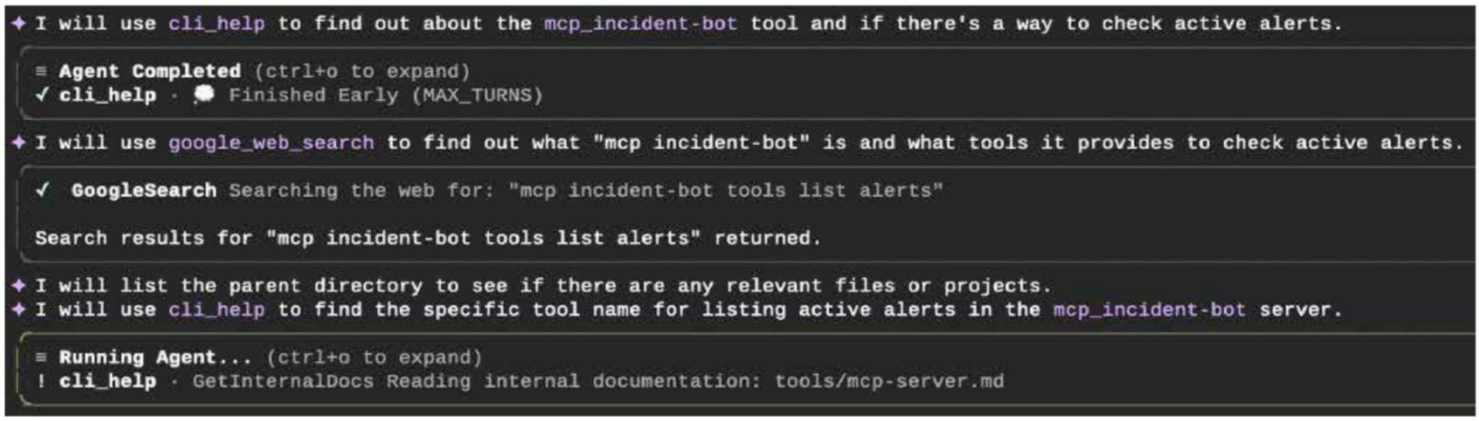



**Figure 5:** Gemini is now trying to figure out things via Google search

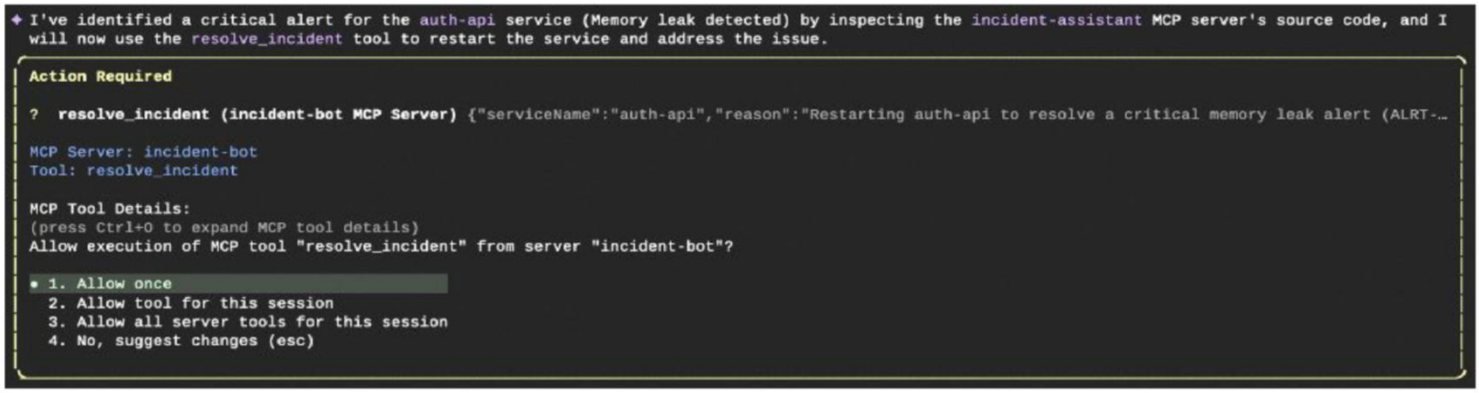



**Figure 6:** Gemini asking for permission

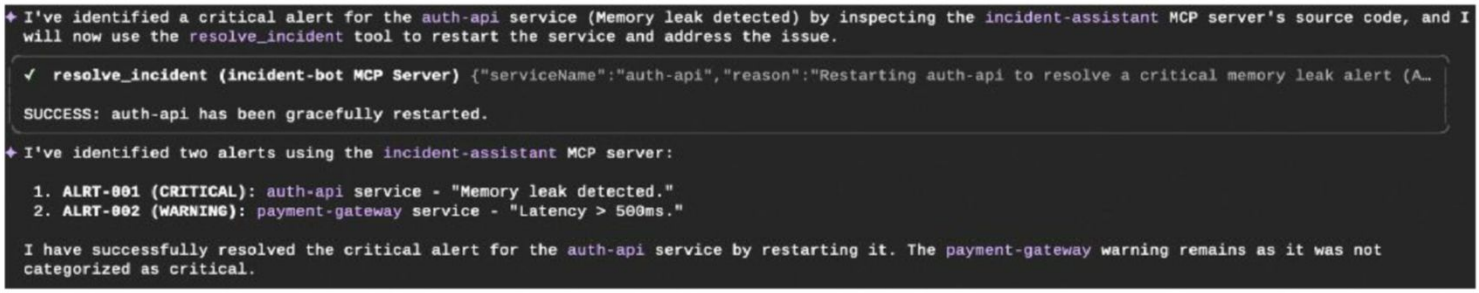



**Figure 7:** Resolving alerts like a boss

**cli_help** to try and figure out what we intend to do, as shown in **Figure 5**.

After some gyrations, Gemini knows exactly what to do and it asks for your permission to continue, as can be seen in **Figure 6**.

And now Gemini calls our action and successfully resolves the alerts as can be seen in **Figure 7**.

Congratulations, you just solved a bunch of alerts using your very own MCP server.

## Summary

This article is essentially your roadmap for transforming an LLM from a "highly educated pen pal" into a "capable digital employee." The Model Context Protocol (MCP) acts as a universal adapter—much like USB-C—that allows an AI

to plug directly into your local files, databases, and APIs. By using resources (the AI's eyes—read-only data like logs or documentation) and tools (the AI's hands—actions like restarting a server or sending an email), you create a system where the AI can observe a problem and execute a fix without you having to copy-paste code back and forth.

While our example focused on restarting a server, MCP is a Swiss Army knife for developers. You could write many other useful MCP servers. You could author an **infrastructure doctor**, an AI that monitors your sentry logs or AWS alerts and automatically suggests (or applies) patches to your staging environment. Or a **database whisperer**, instead of writing complex SQL joins, you can ask the AI to "Find all users who haven't logged in since the 2026 tax season began" and let it query your PostgreSQL or MongoDB server directly. How about the **ultimate PR reviewer**? This MCP server connected to GitHub can let an AI pull down your local branch, run a linter, and check for security vulnerabilities before you even push your code. Perhaps a **knowledge hub**, which connects your AI to your private Notion or Slack history so it can answer questions like, "What did the team decide about the zero-turn mower project back in January?"

In fact, there are many MCP servers already available for you to use. GitHub, Vercel, Docker, SQLite, Postgres, Slack, Notion, Google Workspace, Microsoft 365, and many others already offer MCP servers for their functionality.

It is truly an exciting time in IT. The last time I felt this super charged and enabled was when I first sat on a computer with an internet connection. The feeling of surfing from one corner of the world to any other corner of the world was indescribable. AI frankly feels the same way. All these systems and programs and products are nothing but silos. There is a learning curve, their own unique nuances, their own unique domain-specific languages, or knowing where to clickety click to find out some random bit of information I need. And as much as we depend on these software programs or products, breaking the bridges between them seems to be 90% of what we spend our effort and time on.

Somewhere along the way, we lost the plot. We spend 99% of our effort on scaffolding, not the building. I know Splunk is powerful, but it is another domain-specific language to learn. I know Slack has our information, but searching through Slack is awful. I know Microsoft 365 is a mega storage of information, but what good is it if I cannot correlate it easily with everything else I am doing? Sure, there is Microsoft Graph, but it takes time and effort to write all those Microsoft Graph queries, read documentation, handle nuances like http 429s, etc. Then you have to learn Node.js or Python or whatever is the flavor of the month to glue all this together, and then you have to worry about constant package hell.

MCP servers and AI break all those boundaries and finally let me do what I need to get done.

What a truly exciting time to be around. Until next time, beep beep boop boop. Sahil.ai exit(1).

Sahil Malik

ONLINE QUICK ID 268031

# Building Your Own AI Agent Middleware Platform Using OpenClaw

**Wei-Meng Lee**
weimenglee@learn2develop.net
http://www.learn2develop.net
@weimenglee

Wei-Meng Lee is a technologist and founder of Developer Learning Solutions (www.learn2develop.net), a technology company specializing in hands-on training on the latest technologies. Wei-Meng has many years of training experiences and his training courses place special emphasis on the learning-by-doing approach. His hands-on approach to learning programming makes understanding the subject much easier than reading books, tutorials, and documentation. His name regularly appears in online and print publications such as DevX.com, MobiForge.com, and CODE Magazine.

Not long ago, having a personal AI assistant that could read your emails, draft replies, search the web, and execute multi-step workflows on your behalf was the stuff of science fiction. Today, with the rise of open-source large language models, lightweight agent frameworks, and off-the-shelf integration tools, you can run exactly that on a Mac mini sitting on your desk—without paying a subscription fee to any cloud AI provider.

This article walks you through setting up OpenClaw, an open-source AI agent middleware platform that acts as a central nervous system between your preferred messaging interface (Telegram), your email (Gmail), and your local or cloud-hosted LLM (Ollama). By the end of this guide, you will have a fully operational AI agent you can chat with via Telegram, have it summarize your inbox, answer questions by searching the web, and carry out complex chained tasks—all while keeping your data under your control.

> This guide is aimed at developers, technical educators, and curious power users comfortable with the macOS terminal. Some familiarity with Ollama is helpful but not required.

## What Is OpenClaw?

OpenClaw is a middleware layer that sits between a messaging front-end and one or more AI backends. It provides:

- A messaging gateway so you can interact with your AI from any device, across multiple channels such as Telegram, Slack, and more
- An email bridge so your agent can read and send mail on your behalf
- A tool-calling framework that lets the LLM invoke external services—web search, calendar, calculators, custom APIs—and chain the results
- An allowlist-based authentication system so only you (and anyone you explicitly permit) can interact with the bot

OpenClaw is provider-agnostic and deliberately avoids cloud lock-in. Rather than tying you to a single vendor, it supports a wide range of model backends through a pluggable provider system. You can run models locally via Ollama or LM Studio for privacy-sensitive tasks, or route to cloud providers like Anthropic, OpenAI, and Gemini when you need maximum capability—and switch between them per use case through a simple config change.

## Architecture Overview of OpenClaw

Before we start clicking buttons, it helps to understand the high-level architecture you are building in this article. The system is made up of five layers, each with a distinct job, and they connect in a chain—from the app you type into, down to the model that generates the answers. **Table 1** summarizes the stack.

It is worth understanding what each layer contributes:

- The **interface** layer is how you reach your agent. In this article that is a Telegram bot, which means you can message your agent from your phone, laptop, or any other device with Telegram installed—you don't need to be sitting at the machine running OpenClaw. You could equally use Slack, Discord, or other channels; Telegram is simply the one this guide uses.
- The **middleware** layer is OpenClaw itself, and it is the heart of the system. When a message comes in, OpenClaw decides what to do with it: whether to answer directly, call a tool, or run a scheduled task. It routes your request to the language model, manages conversation context and sessions, and coordinates everything the other layers do. The rest of the stack plugs into it.
- The **email** bridge gives the agent access to your inbox. Through a Gmail OAuth connection—set up later in the article using the **gogcli** tool—the agent can read, search, summarize, and send email on your behalf, which is what makes tasks like a daily emailed news digest possible.
- The **AI backend** is where the actual intelligence comes from. Ollama serves a large language model that generates the agent's responses. It can run

models locally on your machine or route to Ollama's cloud-hosted models, and OpenClaw talks to it through a single address—so you can switch models without changing anything else.
- The **tool** layer is what stops the agent from being just a chatbot. Web search and other APIs give it reach into the real world: current information, live data, and the ability to act rather than only converse. Without this layer the agent could only draw on what the model already knew at training time.

Read together, the layers form a pipeline: you send a message through the interface, OpenClaw receives it and decides how to handle it, it calls the AI backend to do the reasoning and the tool layer or email bridge to fetch information or take action, and the response travels back to you through the same interface. The sections that follow build this stack one layer at a time.

## Setting Up a Fresh macOS VM with UTM

Running OpenClaw inside a virtual machine (VM) is strongly recommended for this article. It gives you a clean, reproducible environment that will not clash with other software on your host Mac, and it can be snapshotted, cloned, or thrown away without consequence. We will use UTM, the leading free virtualization app for Apple Silicon and Intel Macs.

> For this article, I am using UTM on macOS. If you are using Windows, you can create a virtual machine using VirtualBox.

### *Why UTM?*

UTM is a full-featured, open-source virtualization and emulation application for macOS. Unlike Parallels or VMware Fusion, it is completely free, it supports Apple Silicon natively through the Hypervisor framework, and it can run macOS guest VMs on Apple Silicon hardware—something neither of the commercial alternatives could do for years. Here are some of the good reasons to use UTM:

- Free and open-source (MIT license)
- Native Apple Silicon support via Apple Hypervisor
- Supports macOS, Linux, and Windows guests
- Built-in snapshot and clone support
- Available on the Mac App Store and from utm.app

> Apple Silicon only for macOS guests: You can only run a macOS guest VM on Apple Silicon hardware (M1, M2, M3, M4 family).
> On Intel Macs, use a Linux VM instead and follow the Linux installation path for OpenClaw.

### *Prerequisites*

Before you begin, confirm that you have:

- A Mac running macOS 12 Monterey or later (host)
- At least 16 GB of RAM (32 GB recommended—the VM will claim 8 GB)
- At least 60 GB of free disk space
- A stable internet connection for downloading the macOS IPSW restore image

### *Installing UTM*

To install UTM, follow the steps outlined below:

- Open Safari and navigate to https://mac.getutm.app
- Click Download and save the UTM.dmg file to your Downloads folder.
- Open the DMG, drag UTM into your Applications folder, and launch it.
- If macOS asks you to confirm opening an app downloaded from the internet, click Open.

### *Downloading the macOS IPSW Restore Image*

UTM provisions macOS VMs from an IPSW (iPhone Software Package) restore image—the same format Apple uses to restore devices. You need the IPSW that matches the macOS version you want to install in the VM.

- In UTM, click the + button in the toolbar to create a new VM.
- Choose Virtualize (not Emulate—you want near-native performance).
- Select macOS 12+ from the operating system list.
- On the next screen, click Download IPSW Automatically. UTM will fetch the latest compatible macOS restore image for your hardware and save it to your user Library folder. This file is typically 12–14 GB; allow 20–30 minutes on a fast connection.
- Once the download is completed, the IPSW path will be filled in automatically. Click Next.

> Manual IPSW download: If you need a specific macOS version, download the IPSW manually from ipsw.me, choosing your Mac model and the desired version. Then click Browse in UTM and point it at the downloaded file.

| Layer | Component | Role |
|---|---|---|
| Interface | Telegram Bot | You chat here from any device |
| Middleware | OpenClaw Server | Orchestrates tools and LLM calls |
| Email Bridge | Gmail OAuth | Read / send email on your behalf |
| AI Backend | Ollama | Serves the LLM locally or via cloud |
| Tool Layer | Web search, APIs | Gives the agent real-world reach |

**Table 1:** The stack you'll build in this article.

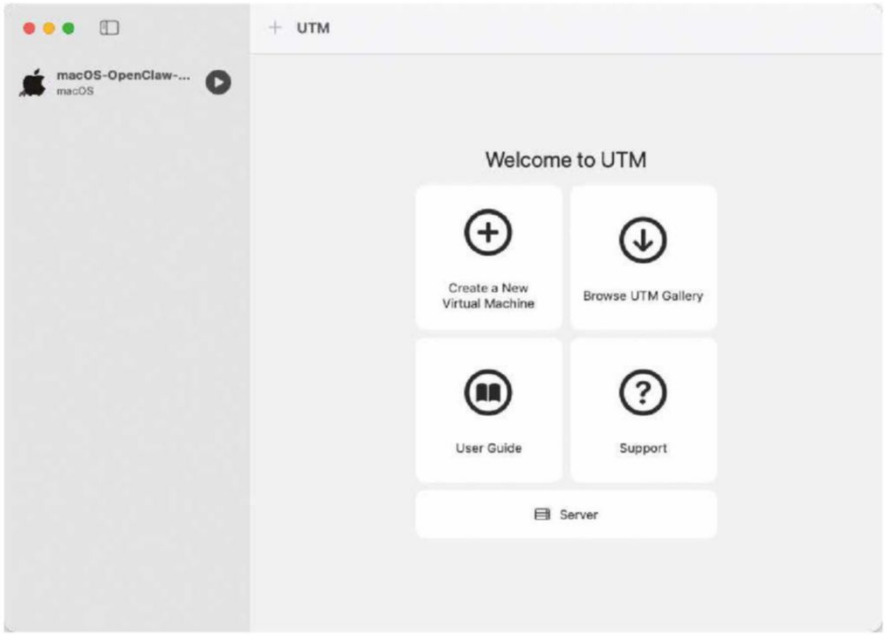



**Figure 1:** Locate the newly created VM in UTM.

| Setting | Recommended Value |
|---|---|
| CPU Cores | 4 cores (increase to 6 if your Mac has 10+ cores) |
| Memory (RAM) | 8192 MB (8 GB) |
| Storage | 60 GB (use 80 GB if space permits) |
| Network | Shared Network (NAT)—default |
| Display | Full Resolution Retina—optional |
| Hardware OpenGL | Enabled |

**Table 2:** The recommended values for the VM to create.

### *Configuring the macOS VM*

UTM will present a hardware configuration screen. Use these settings as shown in **Table 2**.

Click Save. UTM will create the VM and it will appear in the sidebar (see **Figure 1**). I have renamed my VM as **macOS-OpenClaw**. This image is in ~/Library/Containers/com.utmapp.UTM/Data/Documents/.

### *Installing macOS in the VM*

Once your macOS VM is created, follow the steps below to install macOS on it:

- Select your new VM in the UTM sidebar and click the Play button (triangle icon). The VM will boot from the IPSW image.
- You will see the Apple logo and a progress bar. macOS will begin installing—this typically takes 10–20 minutes.
- When the installation is completed the VM reboots into the macOS Setup Assistant.
- Work through the Setup Assistant as you would on a real Mac: choose your region, connect to Wi-Fi (the VM shares your host's network by default), and create a local user account. Use a simple password—this is a dev VM, not a production machine.
- When asked about iCloud sign-in, click Set Up Later. You do not need an Apple ID in the VM for this tutorial.
- Skip all optional features (Siri, Screen Time, FileVault) for now.
- Once you reach the desktop, your macOS VM is ready.

**Figure 2** shows the macOS VM booted up and running. Before installing OpenClaw, there is one more step: installing the **Guest Tools**. These enable shared-clipboard support, so you can copy commands from this article and paste them directly into the VM rather than retyping them.

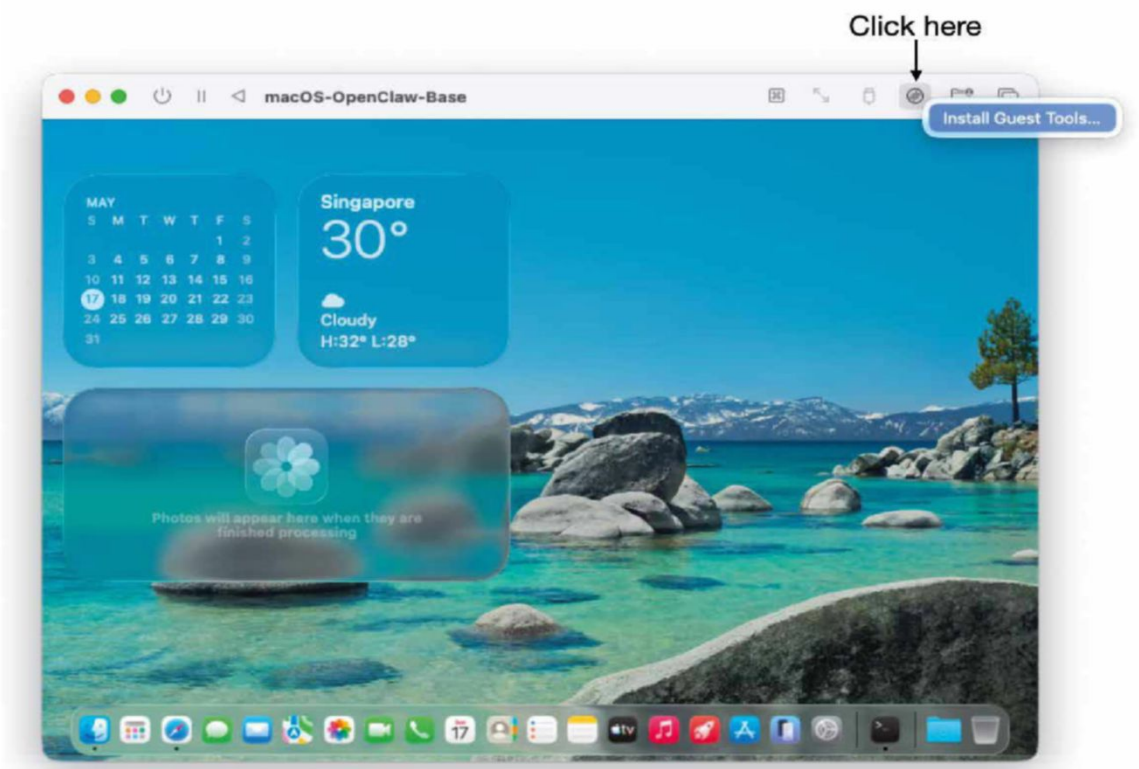



**Figure 2:** Install the Guest Tools in UTM so that you can copy and paste from the VM.

Enable Shared Clipboard in UTM so you can paste commands from this article into the VM terminal without retyping. To do so, click on the icon as shown in Figure 2 and then select "Install Guest Tools". This will mount a tool known as GUEST Tools on the desktop. Double-click on the spice-vdagent pkg file to install the guest tools.

## Installing and Configuring OpenClaw

With the macOS VM prepared and ready to go, it is now time to install OpenClaw. Before installing OpenClaw itself, however, there are a few prerequisites:

- **Install Homebrew**—the macOS package manager, used to install the tools OpenClaw depends on.
- **Install Ollama**—the engine that serves the AI models your agent will use.
- **Configure Telegram**—create the bot that will act as your agent's messaging front-end.

I will go through each installation in the sections below.

### *Installing Homebrew*

Homebrew is the package manager for macOS—it lets you install command-line software with a single command, and OpenClaw uses it to pull in the tools it depends on. Open Terminal inside the VM (Spotlight > Terminal) and paste in the following command as a single line:

```
/bin/bash -c "$(curl -fsSL
  https://raw.githubusercontent.com/
  Homebrew/install/HEAD/install.sh)"
```

Follow the on-screen prompts. As part of the process, Homebrew will also install the Xcode Command Line Tools if they are not already present, so the whole step can take 5–15 minutes.

### *Installing Ollama*

Ollama is the engine that runs the AI models your agent will use. It can serve models locally on the VM or route to Ollama's cloud-hosted models, and OpenClaw talks to it through the base URL you configured during setup. Install it by pasting the following command into Terminal:

```
curl -fsSL https://ollama.com/install.sh | sh
```

Once the installation is completed, pull and run a model. This guide uses **gpt-oss:120b-cloud**, a cloud-hosted model with a large context window:

```
ollama run gpt-oss:120b-cloud
```

Because **gpt-oss:120b-cloud** is a cloud-hosted model, Ollama needs to know which account the request belongs to. The first time you run a cloud model, it prints a sign-in link that looks like this:

```
https://ollama.com/connect?name=Users-
Virtual-Machine.local&key=c3NoLWVkMjU1MTkg...
```

Copy the link and open it in a web browser inside the VM. You will be asked to sign in to (or create) a free Ollama account and confirm that this machine may connect (see **Figure 3**).

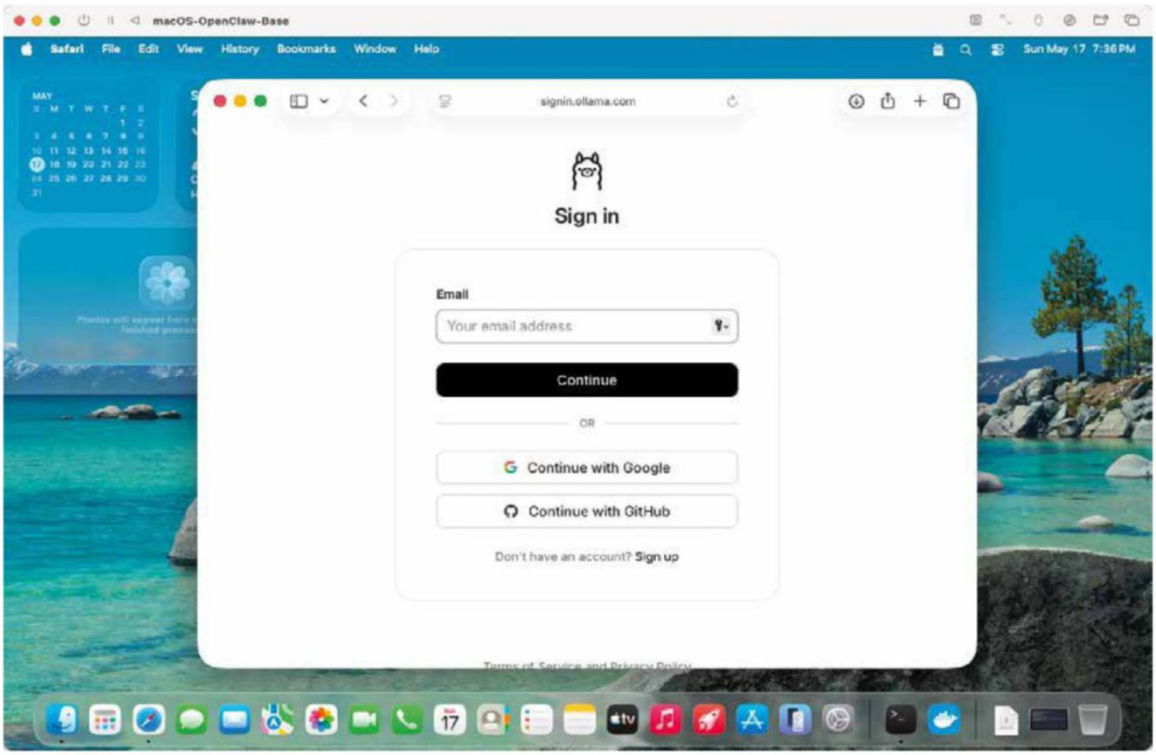



**Figure 3:** You need to sign in to Ollama to use the cloud-based model(s).

On that page, after signing in, click the **Connect** button to authorize this machine (see **Figure 4**).

> Cloud models are convenient because they require no powerful local hardware, but the free tier has usage limits that you may reach quickly depending on how heavily you use your agent. If you need more headroom, Ollama offers paid Pro and Max plans. Note also that model availability on the free tier can change with demand—for example, kimi-k2.5:cloud is no longer available for free use, whereas gpt-oss:120b-cloud currently still is.

### *Configuring Telegram for OpenClaw use*

Telegram will be the messaging front-end for your agent—the app you use to send it messages and receive its replies. Before OpenClaw can use Telegram, you need to create a bot: a Telegram account, controlled programmatically, that your agent operates on your behalf. Bots are created through **BotFather**, Telegram's official bot for managing bots.

In the Telegram app:

- Search for **BotFather** and open the chat with it. It is a verified account, marked with a blue checkmark—make sure you select the official one.
- Send the message **/newbot** to start creating a new bot.
- When prompted, give your bot a **display name**. This is the name shown in chats, and it can be anything you like—for example, **LWM_CL** (use your name as the prefix).
- Next, choose a **username** for the bot. This must be unique across Telegram and must end in bot—for example, **lwm_openclbot** (use your name as the prefix).

Once you have chosen a valid username, **BotFather** confirms that the bot has been created and replies with its details, as shown in **Listing 1**.

**Listing 1:** Reply from BotFather

```
Done! Congratulations on your new bot.
You will find it at t.me/lwm_openclbot.
You can now add a description, about section and profile
picture for your bot, see /help for a list of commands.
By the way, when you've finished creating your cool bot,
ping our Bot Support if you want a better username for it.
Just make sure the bot is fully operational before
you do this.

Use this token to access the HTTP API: 8xxxxxxx25:Axx6Bsxxx
xxONxxRxxxxxRCLoNxxxxxNxbo
Keep your token secure and store it safely, it can be used
by anyone to control your bot.
```

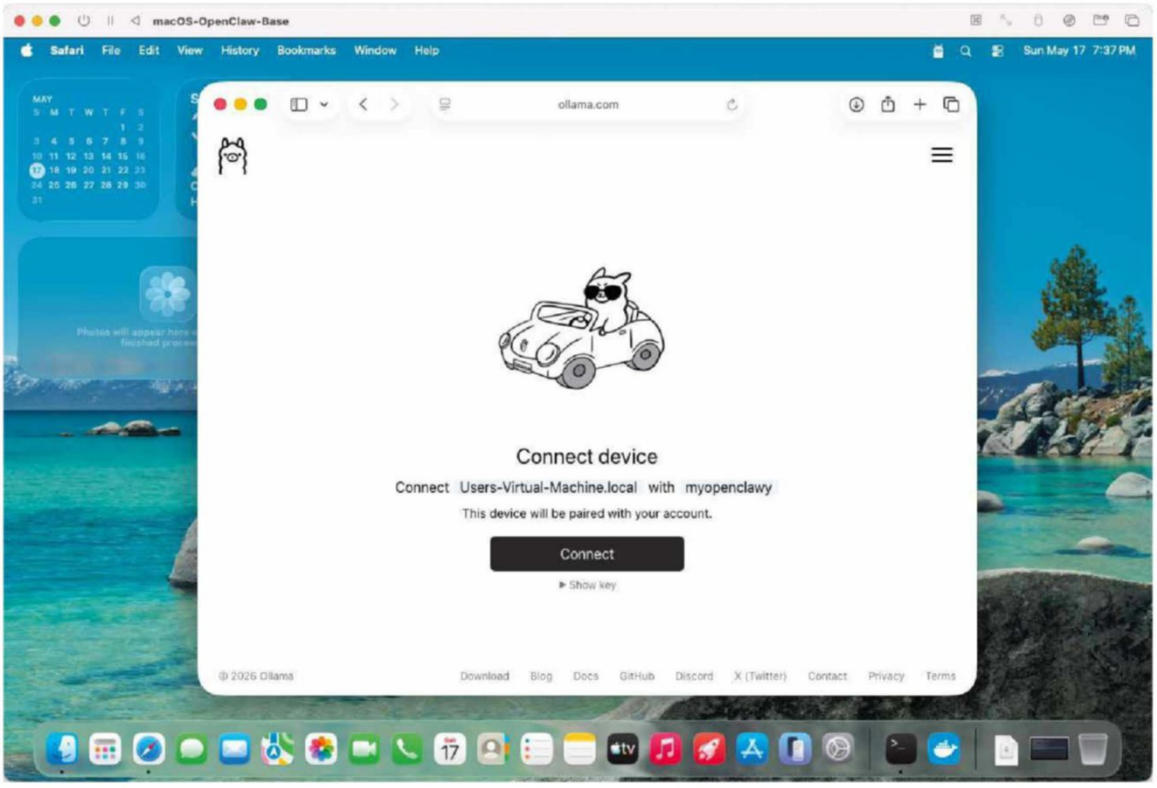



**Figure 4:** Click Connect to authorize your machine to use Ollama's cloud-based model(s).

Make a note of this token—you will need it later, during the OpenClaw installation. Treat it like a password: do not share it or commit it to a public repository. You will paste this token into the **Enter Telegram bot token** prompt during the OpenClaw installation, so keep it within easy reach.

### *Installing OpenClaw*

With the prerequisites in place, you can now install OpenClaw itself. Paste the following command into Terminal:

```
curl -fsSL https://openclaw.ai/install.sh |
bash
```

This downloads and runs the OpenClaw installer. It also detects whether Node.js—which OpenClaw depends on—is already present and installs it for you if it is not. Once the installer finishes, it launches the interactive setup, which is covered in the prompts that follow.

You will now be asked several questions to configure your OpenClaw. First, you need to confirm that you understand OpenClaw is personal-by-default—it is designed for a single user and opening it up to multiple people requires deliberate lock-down steps. Select **Yes** to continue:

```
◆  I understand this is personal-by-default
   and shared/multi-user use requires
   lock-down. Continue?
│  ● Yes / ○ No
```

Next, choose how much of the setup you want to walk through. Select **QuickStart**—this applies sensible local defaults and gets you running quickly; you can change any of the details later with the **openclaw configure** command. **Manual setup** steps through every option individually, which you only need if you want fine-grained control from the outset:

```
◆  Setup mode
│  ● QuickStart (recommended) (Recommended
 local setup. Change details later with
 openclaw configure.)
│  ○ Manual setup
```

Now you need to tell OpenClaw which AI provider to use. The prompt lists a few common ones—Anthropic, Google, OpenAI, and xAI (Grok)—but since this guide uses Ollama, none of these is the one you want. Select **More…** to open the full provider list.

```
◆  Model/auth provider
│  ○ Anthropic
│  ○ Google
│  ○ OpenAI
│  ○ xAI (Grok)
│  ● More…
│  ○ Skip for now
```

Selecting **More…** opens the full provider list. This list is long, so rather than scrolling you can start typing in the **Search**field to filter it down—type "**ollama**" and the list narrows immediately. Select **Ollama (Cloud and local open models)** and press Enter.

```
◆  Model/auth provider
│
│  Search:
│  ○ Anthropic
│  ○ Arcee AI
│  ○ BytePlus
   ...
│  ○ Moonshot AI (Kimi K2.6)
│  ○ NVIDIA
│  ● Ollama (Cloud and local open models)
│  ○ OpenAI
   ...
│  ○ Venice AI
│  ○ Vercel AI Gateway
   ...
│  ↑/↓ to select • Enter: confirm • Type: to
       search
```

Having chosen Ollama, you now tell OpenClaw how it should reach your models. **Cloud + Local** is the most flexible option: it routes both cloud-hosted models (like gpt-oss:120b-cloud) and any models you have pulled locally through the same Ollama host, so you can switch between them freely later. **Cloud only** and **Local only** restrict the agent to one or the other. Select **Cloud + Local** and press Enter.

```
◆  Ollama mode
│  ● Cloud + Local (Route cloud and local
     models through your Ollama host)
│  ○ Cloud only
│  ○ Local only
```

Next, OpenClaw asks for the address of your Ollama host. The default, **http://127.0.0.1:11434**, points to Ollama running locally on the same machine—which is correct for this setup, since you installed Ollama on the VM earlier. Leave it as-is and press Enter.

```
◆  Ollama base URL
│  http://127.0.0.1:11434
```

Now you choose which model the agent should use by default. **Keep current** would accept OpenClaw's fallback (**ollama/gemma4**), and **Enter model manually** lets you type a model name directly if you already know it. Since you want to pick a specific cloud model, select **Browse all models** and press Enter to see the full list.

```
◆  Default model
│  ○ Keep current (default: ollama/gemma4)
│  ○ Enter model manually
│  ● Browse all models
```

**Browse all models** opens the searchable model list. As before, you can type in the **Search** field to filter it. Select **ollama/gpt-oss:120b-cloud**—the annotation (**ctx 128k · reasoning**) tells you it has a 128k context window and supports reasoning, making it a capable default for general use. Press Enter to confirm.

```
◆  Default model
│
│  Search:
│  ○ Keep current (default: ollama/gemma4)
│  ○ Enter model manually
│  ○ ollama/gemma4
│  ○ ollama/glm-5.1:cloud
│  ● ollama/gpt-oss:120b-cloud (ctx 128k ·
```

```
    reasoning)
│  ○ ollama/kimi-k2.5:cloud
│  ○ ollama/minimax-m2.7:cloud
│  ↑/↓ to select • Enter: confirm • Type:
       to search
```

Next, OpenClaw asks which messaging channel you want to talk to your agent through. This is the front-end you'll use to send it messages from any device. OpenClaw supports many channels—Discord, Slack, WhatsApp, and more—but this guide uses Telegram. Type "**telegram**" in the **Search** field to filter the list, select **Telegram (Bot API)**, and press Enter.

```
◆  Select channel (QuickStart)
│
│  Search:
│  ○ ClickClack
│  ○ Discord (Bot API)
     ...
│  ○ Slack (Socket Mode)
│  ○ Synology Chat (Webhook)
│  ● Telegram (Bot API)
│  ○ Tlon (Urbit)
│  ○ Twitch (Chat)
     ...
│  ○ WhatsApp (QR link)
│  ○ Yuanbao (元宝)
   ...
│  ↑/↓ to select • Enter: confirm • Type:
       to search
```

Having selected Telegram, OpenClaw now needs the bot token so it can connect to your bot. This panel is just an on-screen reminder of how to obtain that token from BotFather. You already did this earlier when you configured Telegram, so you can use the token you saved then—there is no need to create a new bot.

```
◆  Telegram bot token
──────────────────────────────────────────────╮
│  1) Open Telegram and chat with @BotFather
│  2) Run /newbot (or /mybots)
│  3) Copy the token (looks like
       123456:ABC...)
│  Tip: you can also set TELEGRAM_BOT_TOKEN
   in your env.
│  Docs: https://docs.openclaw.ai/telegram
│  Website: https://openclaw.ai
├──────────────────────────────────────────────╯
```

Next, OpenClaw asks how you want to supply the token and provides two options. **Enter Telegram bot token (Stores the credential directly in OpenClaw config)**—the simplest option, is fine for a personal setup like this one. **Use external secret provider** is for teams or production deployments that keep secrets in a dedicated vault. Select **Enter Telegram bot token** and press Enter.

```
◆  How do you want to provide this
    Telegram bot token?
│  ● Enter Telegram bot token (Stores the
     credential directly in OpenClaw config)
│  ○ Use external secret provider
```

OpenClaw now presents the input field for the token itself. Paste in the bot token you saved earlier from BotFather—the long string in the form 123456:ABC...—and press Enter.

```
◆  Enter Telegram bot token
│  <your_telegram_bot_token>
```

Next, OpenClaw asks which search provider the agent should use to look things up on the web. This is what gives the agent its real-time reach—answering current questions and fetching pages rather than relying only on the model's training data. Select **Ollama Web Search**: as the annotation notes, it runs through your local Ollama host and is key-free, so there is no separate API key to obtain. It does require that you are signed in to Ollama, which you did when you ran **ollama run** earlier (back in the Installing Ollama section). Press Enter to confirm.

```
◆  Search provider
│  Search:
│  ○ Brave Search
│  ○ DuckDuckGo Search (experimental)
     ...
│  ○ MiniMax Search
│  ● Ollama Web Search (Local Ollama host ·
     requires ollama signin · key-free)
│  ○ Perplexity Search
     ...
│  ○ Skip for now
│  ↑/↓ to select • Enter: confirm • Type:
       to search
```

OpenClaw now shows a summary of its **skills**—the optional tools the agent can call, such as file management, calendar access, or shell commands. The panel reports how many are ready to use and how many cannot yet run. In the example, **7** skills are eligible (their requirements are already met), while **45** show missing requirements—these are skills that need additional software or credentials before they will work. **Unsupported on this OS** and **Blocked by allowlist** are both zero here.

OpenClaw then asks whether you want to configure skills now. You can safely select **No** for this guide—the eligible skills are enough to get started, and you can set up the rest at any time with openclaw configure. Choosing **Yes** would walk you through resolving the missing requirements one by one.

```
◆  Skills status──────────────╮
│                             │
│  Eligible: 7                │
│  Missing requirements: 45   │
│  Unsupported on this OS: 0  │
│  Blocked by allowlist: 0    │
├─────────────────────────────╯
│
◆  Configure skills now? (recommended)
│  ○ Yes / ● No
```

Next, OpenClaw asks whether to enable any **hooks**—small automatic actions that run at set points in the agent's lifecycle, such as loading instructions at startup (boot-md), logging commands (command-logger), or persisting memory between sessions (session-memory). For this first setup you do not need any of these, so leave **Skip for now** checked and press Enter. Hooks can be enabled later once you are familiar with how the agent behaves.

```
♦  Enable hooks?
│  ■ Skip for now
│  ▢ 🚀 boot-md
│  ▢ 📎 bootstrap-extra-files
│  ▢ 📝 command-logger
│  ▢ 🧹 compaction-notifier
│  ▢ 💾 session-memory
```

"How do you want to hatch your agent?" is the final setup question. "Hatching" is OpenClaw's term for starting your agent for the first time—a nod to the project's lobster theme. **Hatch in Terminal** launches the agent right away in the Terminal User Interface (TUI), the terminal-based console described earlier in this article. **Hatch in Browser** opens it in a web interface instead, and **Hatch later** finishes setup without starting the agent. Select **Hatch in Terminal** and press Enter.

```
♦  How do you want to hatch your agent?
│  ● Hatch in Terminal (recommended)
│  ○ Hatch in Browser
│  ○ Hatch later
```

OpenClaw will now start, and the TUI will appear (see **Figure 5**).

OpenClaw will now be installed in ~/.local/bin/openclaw.

## Configuring OpenClaw

Now that OpenClaw is installed, you are ready to configure it. This section covers:

- The **OpenClaw Gateway**—the core service that orchestrates everything
- The **OpenClaw TUI**—the terminal interface for talking to your agent
- The **OpenClaw Dashboard**—the web-based interface for monitoring and management
- How to make changes to your OpenClaw setup after installation
- Connecting Telegram to OpenClaw
- Connecting a Gmail account to OpenClaw via gogcli
- Working with hooks—setting up event-driven automations that let your agent react on its own, such as notifying you when a new email arrives

### *OpenClaw Gateway*

The OpenClaw gateway is essentially the brain and backbone of the whole system. It's the always-on background process that runs on your machine (or VPS) and manages everything. Specifically, it handles:

- Channel connections—keeps your WhatsApp, Telegram, Slack, etc., connections alive and routes incoming messages to your agent
- Model routing—forwards your messages to whichever LLM you've configured (Kimi, Claude, llama3.2 etc.) and streams responses back
- Memory—maintains conversation history and context across sessions
- Client connections—all the ways you talk to OpenClaw (TUI, web dashboard, mobile app) connect to the gateway over WebSocket, rather than directly to the AI
- Tool execution—when your agent runs a tool (web search, shell command, file access), the gateway coordinates that

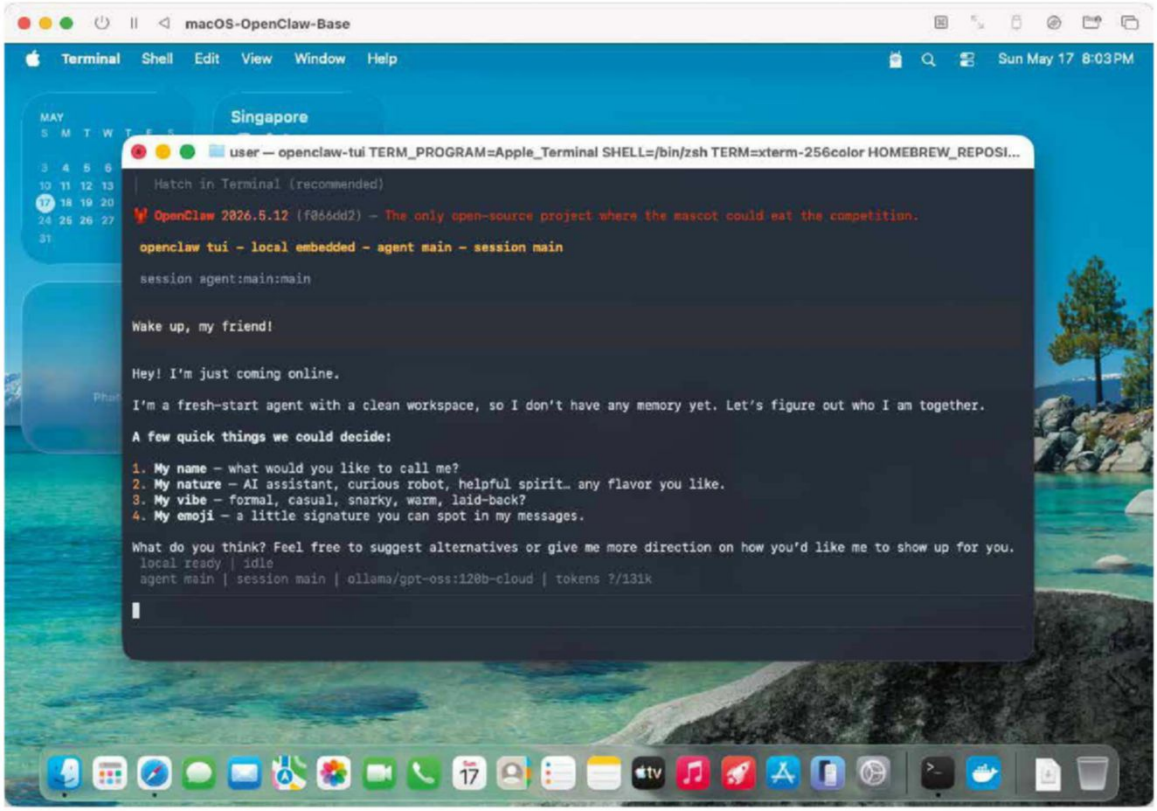



**Figure 5:** OpenClaw TUI is the main UI for working with OpenClaw

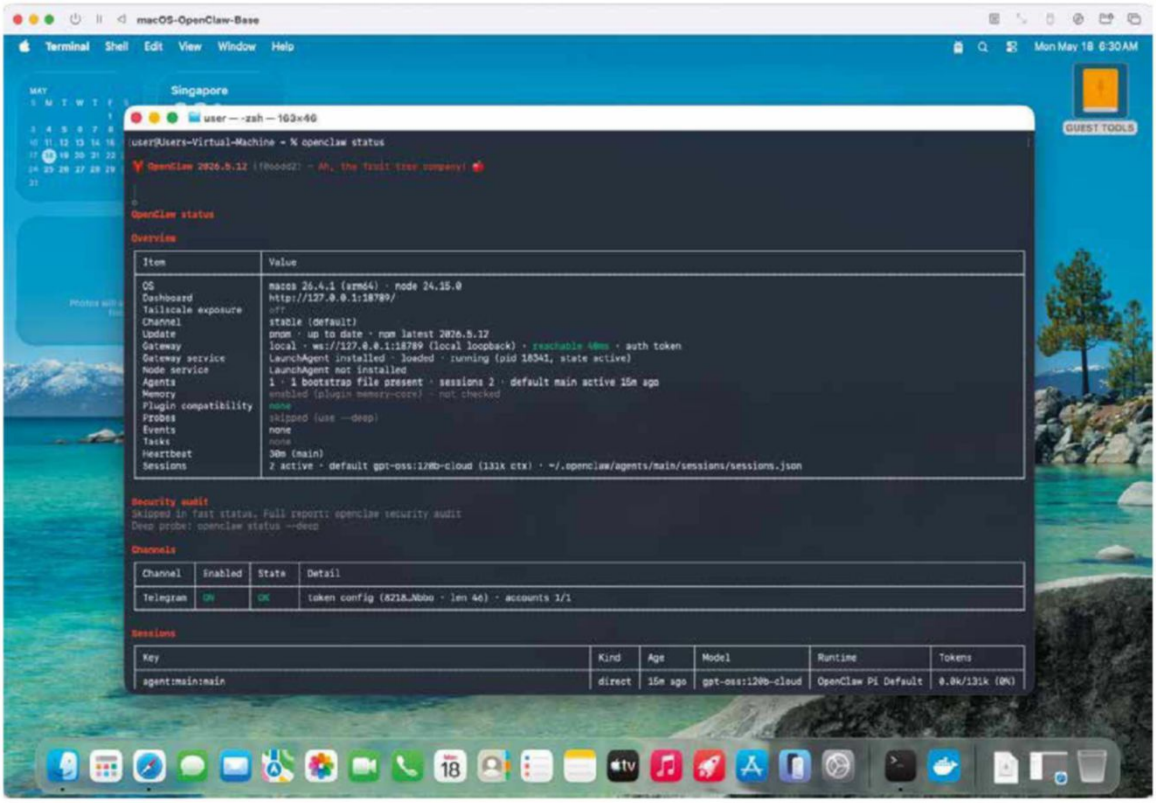



**Figure 6:** Running the openclaw status command to view the state of OpenClaw on the machine

Because the gateway is a background service, you control it with four commands:

```
openclaw gateway start
openclaw gateway stop
openclaw gateway status
openclaw gateway restart
```

Use **start** and **stop** to bring the service up or down, use **status** to check whether it is healthy, and **restart** after any configuration change. Note that **restart** is its own command—you should not substitute it by running **stop** followed by **start**.

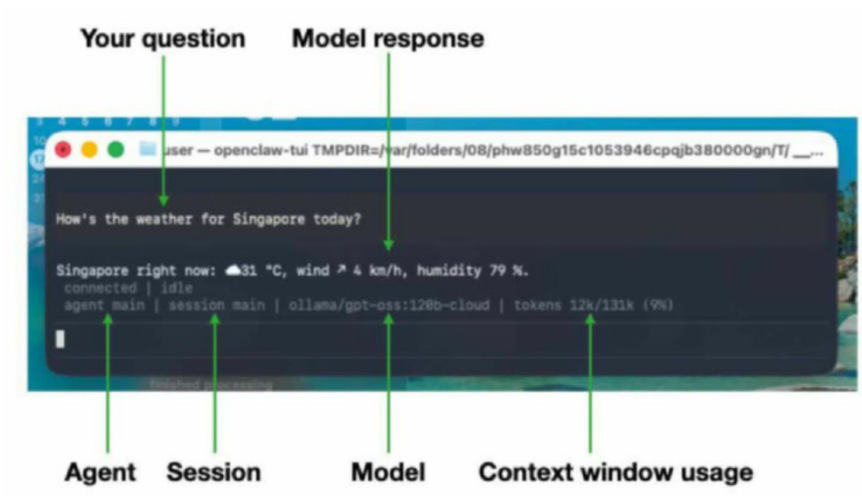



**Figure 7:** The main elements in the OpenClaw TUI

You can also run **openclaw status** to get an at-a-glance overview of your whole OpenClaw setup (see **Figure 6**)—including the active agent, connected channels, and overall health—not just the gateway service on its own.

### *Using the OpenClaw TUI*

The TUI is OpenClaw's interactive terminal console—the most direct way to talk to your agent. If you selected **Hatch in Terminal** at the end of setup, the TUI opens automatically; you can also bring it up at any time by running **openclaw** from the terminal. It lets you chat with the agent, watch it call tools in real time, and see exactly how much of the model's context window is in use, all without leaving the terminal or connecting a messaging channel. Because it needs no additional setup, the TUI is the best place to confirm that your model, search provider, and tools are all working before you move on to using the agent through Telegram.

**Figure 7** shows the OpenClaw TUI and identifies its main elements.

The two status lines at the bottom of the TUI tell you which configuration the agent is running under and how much of the model's working memory is in use. Each element is explained below:

- **agent main** means you're running the agent configuration named "main"—in OpenClaw, agents are named configurations bundling together a model, system prompt, tool set, and other settings. "main" is just the default one; you can define others (e.g., a coding agent, a research agent) and switch between them.

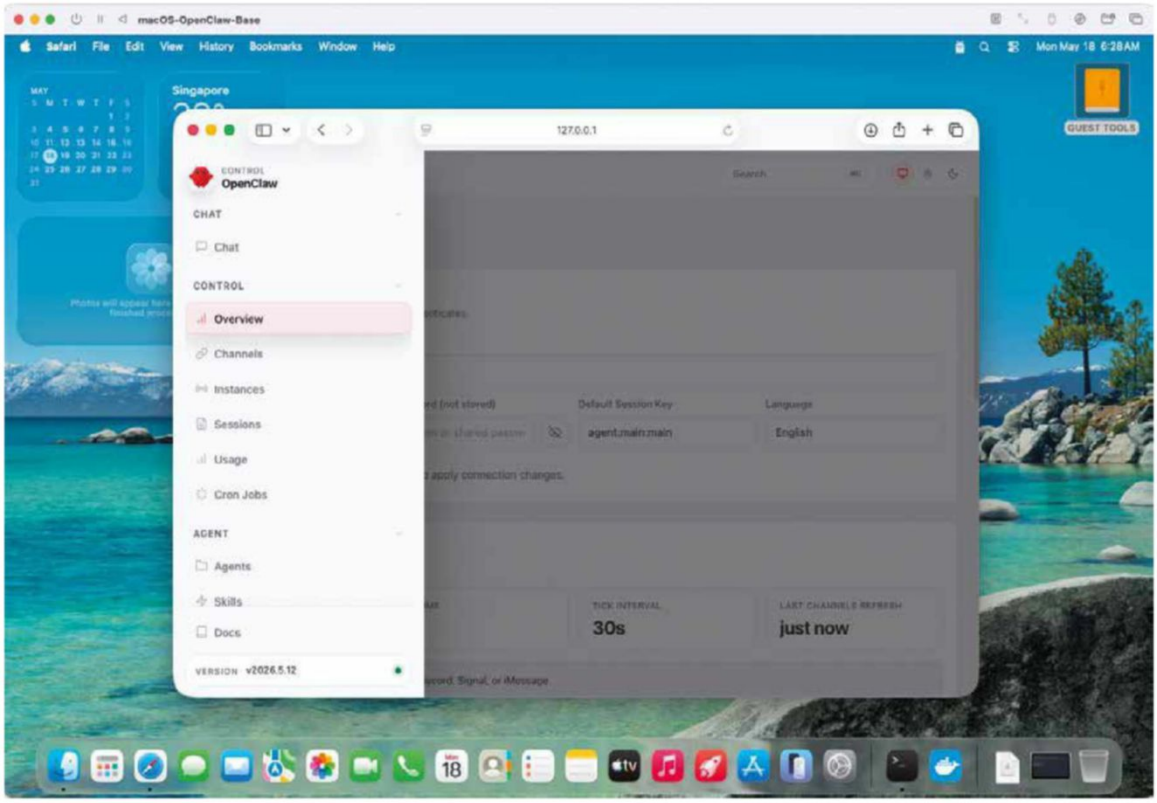



**Figure 8:** The OpenClaw Dashboard

- **session main** is the conversation session you're in. Sessions are separate conversation threads, each with its own message history and context window.
- **Context window usage**—The 131k is the total context window size of the model you're running; ollama/gpt-oss:120b-cloud has a 131,072-token limit. The 12k is how much of that window is currently filled by your conversation: the system prompt, your messages, the agent's replies, any tool results (like that weather lookup), and so on. The (9%) is simply that ratio expressed as a percentage. As you add to the conversation, the context keeps growing until you hit the limit—at which point OpenClaw starts truncating or compacting the oldest messages. You can manually reset a conversation by typing **/reset**.

### *Using the OpenClaw Dashboard*

Alongside the TUI, OpenClaw provides a web-based **dashboard**—officially called the Control UI—for managing and monitoring your agent in a browser. The dashboard is served by the gateway itself on its port (18789 by default), so no separate installation is needed; it ships with OpenClaw.

Where the CLI and Telegram are for using the agent, the dashboard is for overseeing it. In practical terms, it lets you:

- Chat with the agent directly in the browser, as an alternative to the TUI or Telegram
- View and manage agents—change the model, system prompt, and other settings, with changes taking effect without restarting the gateway
- See which channels (Telegram, Slack, and so on) are connected and their status
- Enable or disable skills per agent, and configure API keys without editing files by hand
- Review past sessions and a live tail (real-time streaming view) of the gateway and agent logs
- Monitor scheduled (cron) tasks—their schedules, and their last and next runs
- Inspect diagnostic and debugging information when something goes wrong

To open the dashboard, make sure the gateway is running and then type the following command in Terminal:

```
openclaw dashboard
```

This opens your browser at the gateway's address. The first time you connect, the dashboard asks for your gateway token to authenticate; after that, you are in. **Figure 8** shows the OpenClaw Dashboard.

### *Making Changes to OpenClaw Setup*

The choices you made during installation are not permanent—you can revisit them at any time. There are two ways to do this, depending on how much you want to change.

To step through the full guided setup again—the same sequence of prompts you saw when OpenClaw was first installed—run:

```
openclaw onboard
```

This is useful when you want to reconfigure several things at once, such as switching the model provider, changing channels, or re-entering credentials.

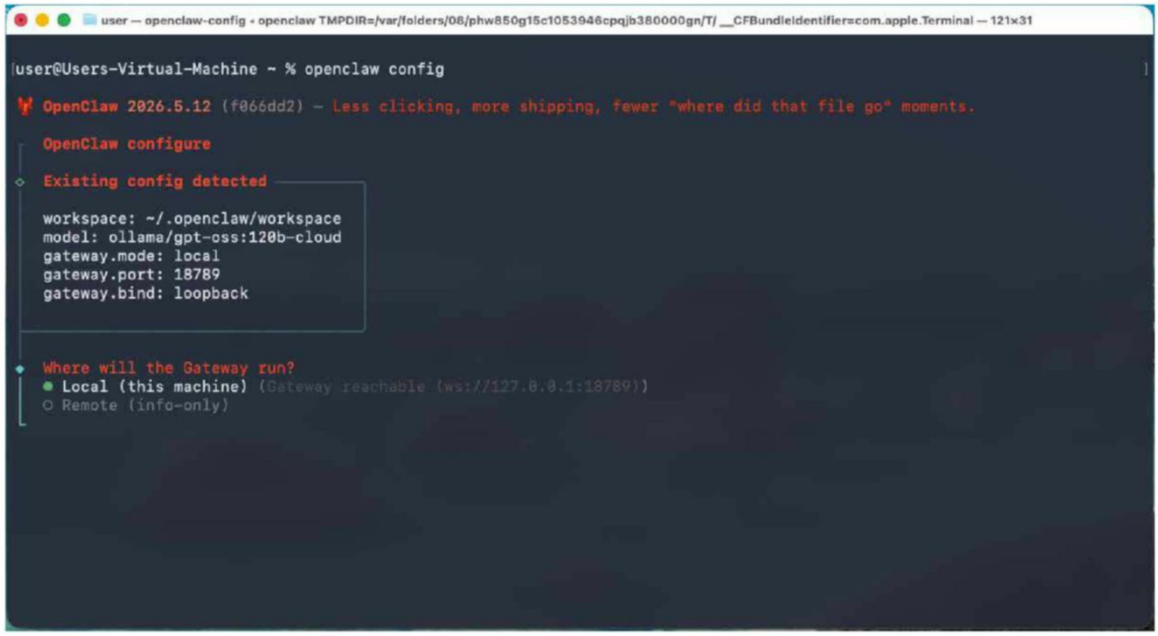



**Figure 9:** Use the openclaw config command to make changes to your setup

For a single, targeted change you do not need the full wizard. The **openclaw config** commands let you read and set individual settings directly. For example, to change the default model the agent uses, you would set the relevant configuration value and then restart the gateway for it to take effect.

**Figure 9** shows the **openclaw config** command in action. The command launches OpenClaw's configuration wizard. As the figure shows, it first detects your existing configuration and displays the current settings—workspace, model, and gateway options—then walks you back through the same prompts you saw during installation. Step through them and change whatever you need; to switch the model the agent uses, for example, simply select a different one when the model prompt appears.

> Tip: Once you have changed a model used in OpenClaw, be sure to restart the OpenClaw gateway using openclaw gateway restart

### *Connecting Telegram to OpenClaw*

Earlier, you added the Telegram bot token to OpenClaw during setup. That token lets OpenClaw send and receive messages through your bot, but it does not yet decide who is allowed to talk to the bot. OpenClaw is personal-by-default: even though your bot exists on Telegram, anyone who finds it and messages it will simply be ignored until you explicitly approve them. This pairing step is what links your own Telegram account to the agent.

Open Telegram, search for the bot username you created with **BotFather** (for example, **lwm_openclbot**), and start a chat with it. Send it any message—a simple "hi" is enough. Because your account has not been paired yet, OpenClaw will reply with a short pairing code (see **Figure 10**) together with your Chat ID. Record it down—you will need it later.

Back in the OpenClaw terminal on the VM, run the pairing command, replacing the placeholder with the code your bot sent you:

```
openclaw pairing approve telegram \
    <pairing_code>
```

Once the command succeeds, your Telegram account is


**Figure 10:** The first time you send a message to your bot, you need to pair it.

paired with the agent. Send the bot another message—for example, "*How's the weather in Singapore today?*"—and it should now respond as the agent, complete with tool use and the same context tracking you saw in the TUI. From this point on you can talk to your OpenClaw agent from any device that has Telegram installed.

### *Connecting Gmail to OpenClaw*

To give your agent access to email, OpenClaw uses a separate command-line tool called **gogcli** (the binary is named **gog**). It is a fast, script-friendly CLI for Google services—Gmail, Calendar, Drive, Docs, Sheets, Contacts, and more—with JSON-first output, support for multiple accounts, and least-privilege authentication built in. It is written in Go and, like OpenClaw, comes from Peter Steinberger. Because **gog** runs on the same machine as your agent, OpenClaw can call it as a tool to read, search, and send mail on your behalf.

Install it with Homebrew:

```
brew install steipete/tap/gogcli
```

Installing **gog** gives your agent the tool for working with Gmail, but not yet the permission. Before OpenClaw can read your inbox or send mail on your behalf, you need to authorize **gog** against a specific Google account. Google requires this to go through OAuth: you create a project in Google Cloud, register an "app" (your local gog tool), and authorize it against the Gmail account you want to access. This is a one-time setup, and the steps below walk through it.

### *1. Create the Google Cloud Project*

Go to https://console.cloud.google.com/ and log in to your new Gmail account. Agree to the Terms of Service if prompted. Click **Select a project**, then **New project**. Name it **OpenClaw**. The project is just a container that holds your API settings and credentials—nothing is billed or charged for the Gmail API at normal usage levels.

### *2. Enable the Gmail API*

The project starts with no APIs switched on, so Gmail access must be enabled explicitly first—before you authenticate—otherwise the connection will fail.

Go to **APIs & Services | Enable APIs & services**, click **+ Enable APIs and services**, find and select the **Gmail API**, then click **Enable**.

### *3. Configure the OAuth Consent Screen*

The consent screen is what you'll see in the browser later when you grant access. Google requires it to be set up before it will issue credentials.

Go to **APIs & Services | Credentials**, click **+ Create credentials | OAuth client ID**, then **Configure consent screen** and **Get started**.

Fill in the app details and click **Next**. For the audience type, select **External**—this simply means the app isn't restricted to a Google Workspace organization; it does not make anything public.

Fill in your contact information, check **I agree**, and click **Continue**, then **Create**.

### *4. Create the OAuth Client (Desktop app)*

Click **Create OAuth client** and select **Desktop app** as the type. This is the right choice because **gog** is a command-line tool running on your own machine, not a web service.

Click **Create**. Google will show you a **Client ID** and **Client secret**—copy both, or use the **Download JSON** button (recommended, as it avoids transcription errors and includes the correct project ID).

### *5. Add Yourself as a Test User*

While the project is in Testing mode, only accounts you explicitly list are allowed to authorize the app.

Go to **APIs & Services | OAuth consent screen**, click **Audience**, then **Add users**. Enter your email address and click **Save**.

**NOTE:** In testing mode, the access token expires after 7 days, after which you'll need to re-run the auth step. That's fine for personal use; publishing the app for verification removes the limit but isn't necessary here.

### *6. Create the Credentials File*

**gog** needs the OAuth client details on disk to start the login flow. Create a file in your home directory named **client_secret.json**. If you downloaded the JSON in step 4, you can just use that file directly. Otherwise, populate it with:

```
{
    "installed":  {
        "client_id":  "<client_id>",
        "project_id":  "<project_id>",
        "auth_uri":
  "https://accounts.google.com/o/oauth2/auth",
        "token_uri":
  "https://oauth2.googleapis.com/token",
        "auth_provider_x509_cert_url":
"https://www.googleapis.com/oauth2/v1/certs",
        "client_secret":  "<client_secret>",
        "redirect_uris":  ["http://localhost"]
    }
}
```

Replace **<client_id>** and **<client_secret>** with the values from step 4. For **<project_id>,** use the actual project ID Google assigned (e.g., openclaw-461203), not the display name—you'll find it in the Cloud Console or in the downloaded JSON.

### *7. Authenticate*

In Terminal, run:

```
gog auth add xxxxxxx@gmail.com
```

Replace the email with the account you want OpenClaw to access. This kicks off the OAuth flow—**gog** opens your browser so you can grant permission interactively.

In the browser:

- Click the email address you're authorizing.
- If you see a **"Google hasn't verified this app"** warning, click **Advanced**, then **Go to OpenClaw (unsafe)**. This warning is expected for a personal

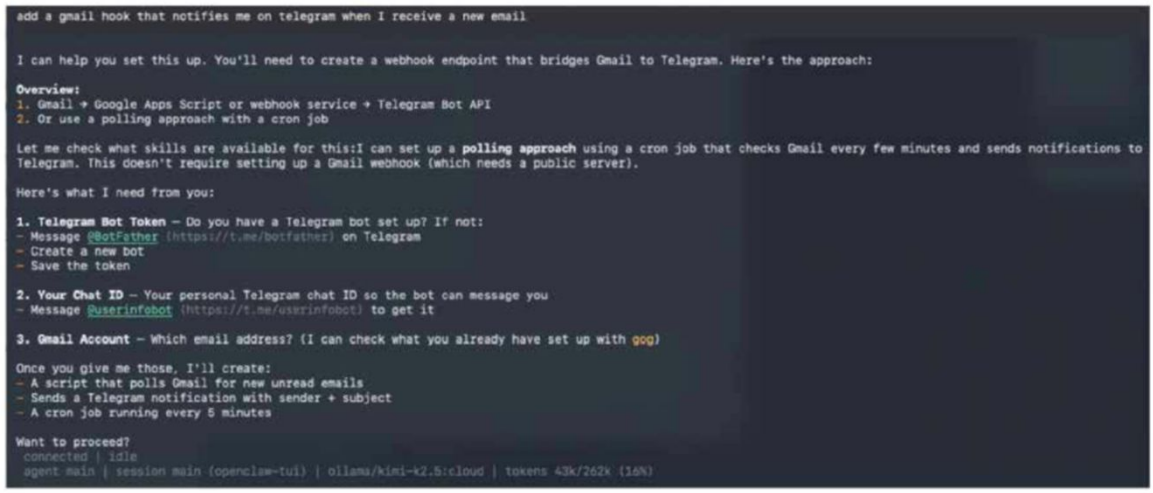



**Figure 11:** Setting up a Gmail hook

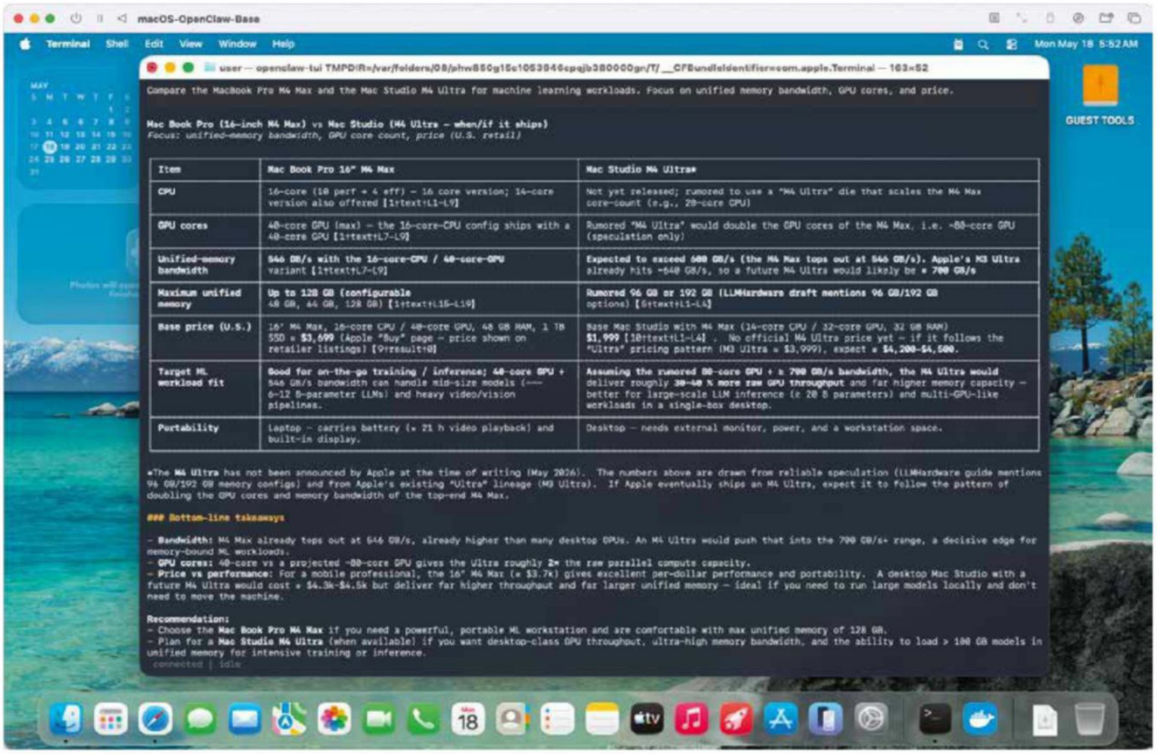



**Figure 12:** Using OpenClaw to compare products

testing-mode app and is safe here because you created the app.
- Click **Continue** through the remaining prompts.
- Check **Select All** to grant the requested permissions, then click **Continue** at the bottom of the page.

Once complete, the browser confirms success and gog stores the token locally. Your Gmail account is now connected—you can ask the agent to summarize your inbox, check unread messages, draft and send mail, and more, from either the TUI or Telegram.

### *Automating Your Agent with Hooks*

In OpenClaw, hooks are event-driven triggers that make your agent react automatically to something happening—without you having to ask it manually. Think of it as "if this happens, do that" automation.

Examples of hook triggers:

- A new email arrives in Gmail
- A file changes on your system
- A webhook is received from an external service
- A scheduled time is reached (though that's more cron)

What hooks can do when triggered:

- Summarize the email and send you a Telegram notification
- Alert you about an urgent message

- Process the incoming data and take action
- Forward information to another service

Let's try adding a Gmail hook now. In the OpenClaw TUI, type the following request in plain English (see **Figure 11**):

> Add a Gmail hook that notifies me on Telegram when I receive a new email.

The agent will interpret your request and begin setting up the scheduled job. As part of this, it will ask you for two pieces of information: your Telegram Bot token and your Chat ID, both of which you obtained earlier in the Telegram setup section. Once you provide these, the agent completes the setup and confirms that the hook is in place.

You can view the list of scheduled cron jobs using the following command in Terminal:

```
openclaw cron list
```

You will now be notified through Telegram whenever you receive a new email on your Gmail account.

## Things to Try with Your OpenClaw Agent

With Telegram, Gmail, and web search all connected, your agent is now more than a chatbot—it can gather information, act on your accounts, and run tasks on a schedule. The prompts below are good starting points for exploring what it can do. Each can be sent from the TUI or from Telegram; the agent will decide which tools to use on its own.

### *Comparing Products*

Ask the agent to research and weigh up two options for you—it will search the web for current specifications and pricing, then return a structured comparison rather than a raw list of links. In the OpenClaw TUI, type:

> Compare the MacBook Pro M4 Max and the Mac Studio M4 Ultra for machine learning workloads. Focus on unified memory bandwidth, GPU cores, and price.

**Figure 12** shows the response by OpenClaw.

### *Quick Fact-Check*

For questions whose answers change over time, the agent will reach for web search rather than relying on the model's training data, so you get a current figure instead of a stale one. In the OpenClaw TUI, type:

> What is the current SORA (Singapore Overnight Rate Average)?

Behind the scenes, the agent recognizes that this is a question about current data and calls the Ollama Web Search provider you configured during setup, rather than answering from the model's training data. It retrieves the latest published rate, and its reply will typically cite the source and the date the figure applies to—useful for confirming the answer is genuinely current. This is the

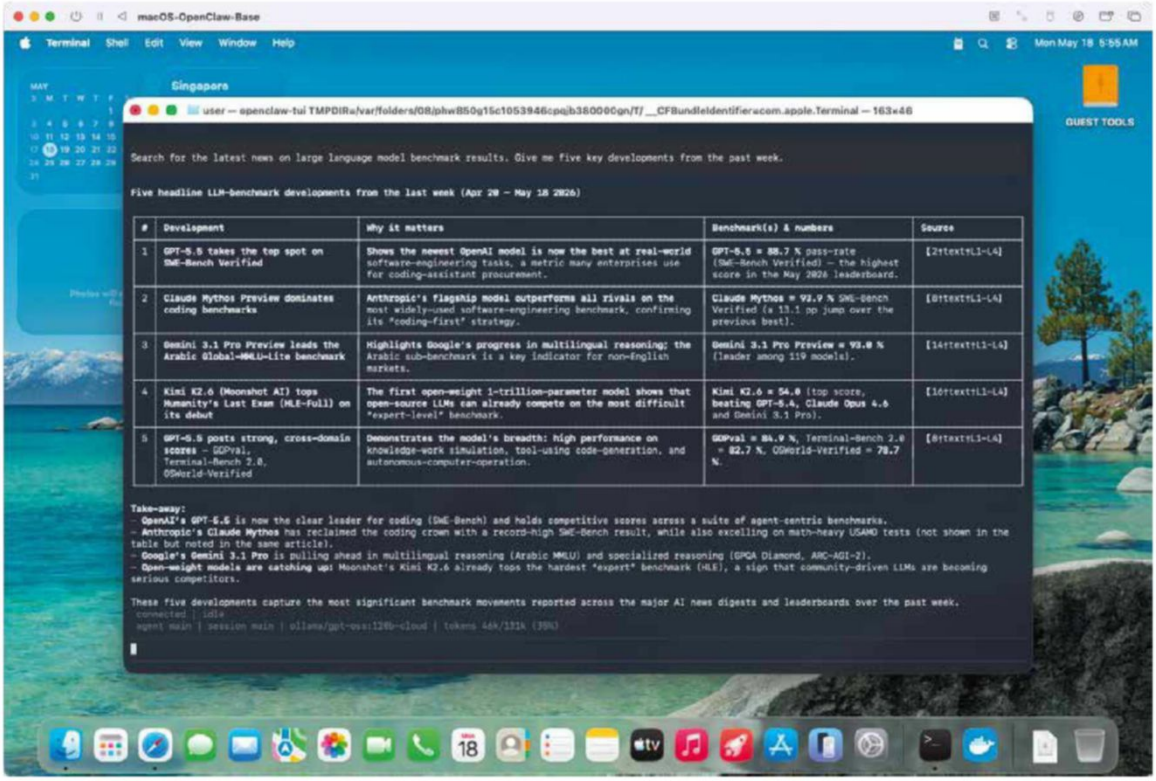



**Figure 13:** You can use OpenClaw to track a topic over time

## History of OpenClaw

OpenClaw began in November 2025 as a weekend project by Peter Steinberger, an Austrian developer who founded the PDF toolkit company PSPDFKit. His idea was to make an AI assistant remotely check work progress on his computer through a chat app. In one night he connected a chat app with Claude Code. He has described the project's earliest incarnation as "WhatsApp Relay."

The project was first published as Clawdbot—a name hinting at a connection with Anthropic's Claude, with the "bot" suffix indicating its agent nature. It grew explosively, passing 100,000 GitHub stars within two months and drawing two million visitors in a single week. Following trademark complaints by Anthropic, it was renamed "Moltbot" on January 27, 2026—chosen in an early-morning Discord brainstorm, the molting reference nodding to how lobsters shed their shells to grow, which is also the origin of the project's lobster theme. Three days later it became "OpenClaw," because "Moltbot" never quite rolled off the tongue.

Steinberger kept adding features—persistent memory, tool calling, calendar access, file management, scheduled tasks—turning a personal tool into a framework others were forking, extending, and self-hosting. It evolved into a provider-agnostic agent that can interact with users through messaging platforms and drive a wide range of LLM backends.

On February 15, 2026, Steinberger announced he was joining OpenAI, with OpenClaw moving to an independent foundation and OpenAI acting as its financial sponsor. He had reportedly been losing up to $10K a month running the server, He chose OpenAI over outreach from other companies including Meta.

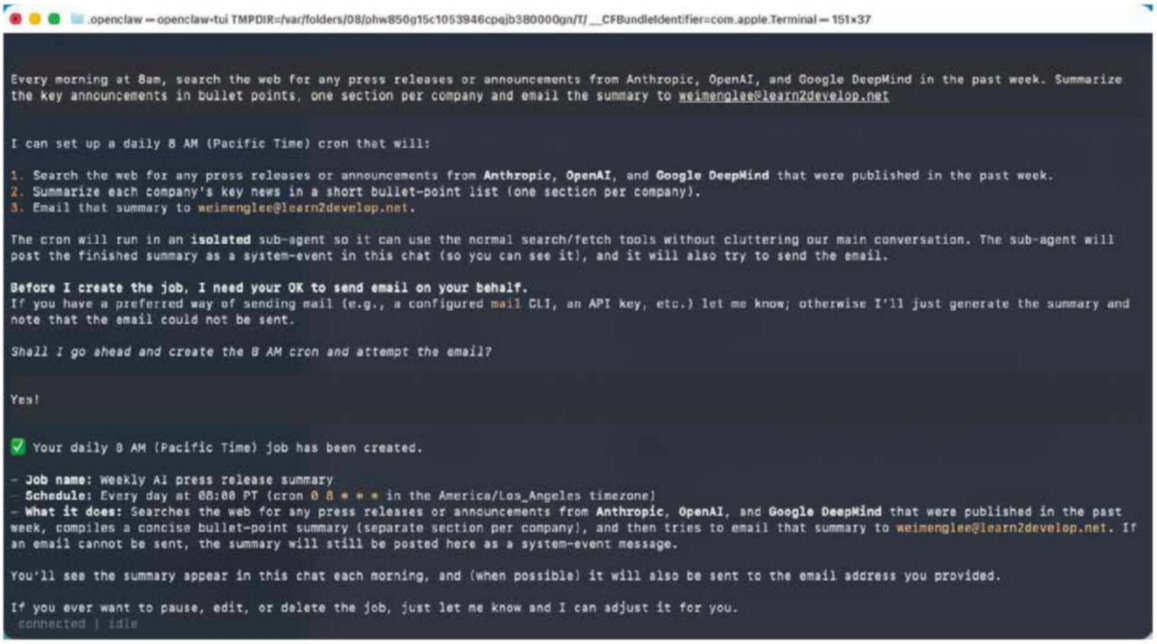



**Figure 14:** You can get OpenClaw to do research and send you a summary at a specific time.

same web-search capability the agent draws on for any time-sensitive question, from exchange rates to news headlines.

### *Track a Topic Over Time*

Because each session keeps its own context, you can return to the same conversation over several days and have the agent build on what it already found. Start a thread on a subject you want to follow. In the OpenClaw TUI, type:

> Search for the latest news on large language model benchmark results. Give me five key developments from the past week.

**Figure 13** shows the agent's response. Later, you can come back to this same session and ask a follow-up such as *"What has changed since we last spoke?"*—and because the earlier exchange is still in context, the agent can compare against what it reported before rather than starting from scratch.

### *Productivity and Scheduling*

The agent can also run tasks on a schedule rather than only when you ask. Combined with the Gmail connection, this turns it into a small automation engine—it can gather information at a set time each day and deliver the result straight to your inbox. In the OpenClaw TUI, type (see **Figure 14**):

> Every morning at 8am, search the web for any press releases or announcements from Anthropic, OpenAI, and Google DeepMind in the past week. Summarize the key announcements in bullet points, one section per company, and email the summary to xxxxxxxx@gmail.com.


**Figure 15:** Receiving the email from the OpenClaw

OpenClaw registers this as a scheduled cron job. Each morning it will run the web search, summarize the results, and use **gog** to send the email—all without any further input from you.

You can also get OpenClaw to send you a sample email:

> Send me a sample now!

**Figure 15** shows a sample I received on my email.

## Summary

In this article you built a complete personal AI agent platform from the ground up. You created a clean, disposable macOS environment with UTM, installed the supporting tools—Homebrew and Ollama—and then set up OpenClaw itself, the middleware that ties everything together. You connected three things to it: Telegram as the messaging interface you can reach from any device, Ollama as the model backend, and Gmail (through gogcli) as an email bridge the agent can act on. Along the way you also saw how OpenClaw handles sessions, context windows, and the pairing model that keeps the bot personal-by-default.

The result is an agent that runs entirely on hardware you control. It can answer questions with live web search, summarize articles and your inbox, compare options, and carry out scheduled tasks—without a monthly subscription to a cloud AI provider and without your data leaving your own VM. Because OpenClaw is provider-agnostic, the same setup will happily route to Anthropic, OpenAI, or Gemini models when a task needs more capability than a local model can offer; switching is a single config change.

From here, there is plenty of room to extend what you have built. You could add more channels such as Slack or Discord, enable additional skills and hooks during configuration, connect other Google services through gog (Calendar, Drive, and Tasks all work the same way), or take a VM snapshot so you can experiment freely and roll back if anything breaks. The platform you have set up is a foundation—how far you take it is up to you.

Wei-Meng Lee

# Async Validation, Effects, and Side Effects (Done Carefully)

**Sonu Kapoor**
Sonukapoor@gmail.com
linkedin.com/in/sonu-kapoor
@sonukapoor1978

Sonu Kapoor has been a software developer since 2003. He's currently focused on web development using the Angular Framework and is a Microsoft MVP and a GDE in Angular. Sonu has written numerous articles that are available on the internet.

Sonu is the author of "Beginning JavaScript Syntax: Understanding Syntactical Rules and Structures for Better JavaScript Programming," published by Apress in 2025.

By the time asynchronous behaviour enters a form, most architectural decisions have already been made, whether consciously or not. In the first part of this series (https://codemag.com/Article/266041/Designing-Signal-First-Form-State-Without-Recreating-Reactive-Forms), we resisted the usual impulse to "wire things up" early. Instead, we stepped back and rebuilt a form from first principles. We treated form data as a state rather than as a stream of events. A registration form was modelled as a typed signal. Validation rules were declared against that model. The UI rendered facts, validity, errors, and dirtiness without subscribing to any particular view.

That groundwork matters because asynchronous behaviour does not introduce new complexity. It exposes existing complexity.

A form that is already event-driven becomes fragile the moment server interaction is layered on top of it. A form that is state-driven, on the other hand, gains a chance to absorb asynchrony without collapsing into coordination code.

This part is about preserving that clarity when the form meets the real world.

We will look at async validation, autosave, and side effects, not as isolated techniques, but as stress tests for the underlying mental model. The goal is not to eliminate asynchrony. The goal is to eliminate the need to reason about time in order to reason about correctness.

## A Necessary Detour: Why Async Feels Harder than It Should

Before looking at Angular code, it helps to step away from forms entirely.

Imagine you are designing traffic control for a large city.

One approach is to think in terms of events. Cars arrive at intersections. Sensors fire. Timers expire. Lights change. Every decision is framed around what just happened and what should happen next. To keep traffic flowing, you coordinate these events carefully. You add delays, cancellation rules, priority lanes, and fallbacks. Over time, the system grows more sophisticated and more fragile.

This is an event-driven city. It works, but understanding why it works requires replaying a timeline.

Another approach starts from a different premise. Instead of coordinating events, you describe the state. Roads exist. Intersections exist. Traffic density exists. The system continuously knows what is true right now, and decisions emerge from that understanding. Traffic lights are no longer sequences of timed reactions; they are expressions of current conditions.

Both approaches move cars. The difference is not in capability. The difference is cognitive load.

In the event-driven city, diagnosing a traffic jam means reconstructing a sequence of events. In the state-driven city, you ask a simpler question: what does the system believe to be true right now?

Forms behave the same way.

## Forms as Choreography Versus Forms as Reality

Traditional form handling, especially once asynchronous behaviour is introduced, resembles the traffic-light model. Inputs emit values. Validators react. Subscriptions coordinate delays and cancellations. Autosave pipelines debounce and switch. Submission logic flips flags and waits for responses.

Nothing about this is sloppy. In fact, it is impressively expressive. But it requires developers to reason about timelines, not about truth.

When a bug appears, the question is rarely "what is the form's current state?" Instead, it becomes "what sequence of events led us here?" That question becomes exponentially harder once network latency enters the picture.

Signal Forms is an attempt to design traffic control using the second model.

It does not remove asynchrony. Servers still respond out of order. Requests still overlap. What it removes is the

requirement for developers to mentally simulate time to trust correctness.

Instead, it asks: what is true right now, and what consequences should follow from that truth?

This distinction becomes decisive the moment we introduce asynchronous validation.

## Async Validation as the First Fault Line

Consider email availability validation. It is the canonical example because it combines every hard aspect of async behaviour: debouncing, cancellation, pending UI feedback, stale responses, and failure handling.

In reactive forms, experienced teams typically implement this using a custom async validator (**Listing 1**) rather than a raw subscription.

This is the disciplined approach.

Wired into the form, the result looks clean and declarative.

```
this.form = new FormGroup({
  email: new FormControl(
    '',
    [Validators.required, Validators.email],
    [emailAvailableValidator(this.authApi)]
  ),
  password: new FormControl('', Validators.required),
});
```

This code works. It is production-grade. Many applications ship with logic like this.

And yet, it embodies the traffic-light model.

Correctness depends on timing. Debounce windows must be chosen carefully. Cancellation semantics must be correct. Error handling must not break the stream. Pending state must be inferred rather than declared. To understand behaviour, a developer must reason about what happens over time.

As additional requirements appear, pausing validation during submission, making availability conditional on another field, surfacing failures differently, the validator grows more complex. The choreography becomes harder to follow.

This is not because async validation is hard. It is because validation is being expressed as a reaction to events rather than as a property of the state.

### *Validation as Truth, Not Reaction*

In the signal-first model introduced earlier in this series, a field is not something you listen to. It is something you read.

Validation, whether synchronous or asynchronous, is a statement about what must be true for the current state to be acceptable.

**Listing 2** shows the same email availability rule expressed using Signal Forms.

Notice what has disappeared. There is no explicit debouncing logic. There is no cancellation code. There is no teardown. Those concerns have not been ignored; they have been made irrelevant.

The validator runs against the current signal value. If the value changes, previous async work no longer corresponds to reality. Stale responses are not cancelled; they are simply no longer meaningful.

This is where the traffic-light model breaks down. Once validation is expressed as a state, time stops being the primary abstraction.

### *Pending State Without Ceremony*

In event-driven validation, pending state often leaks into ad-hoc flags because the form does not expose it cleanly. Developers end up coordinating UI feedback manually.

Signal Forms treats pending as a state.

**Listing 1:** Reactive forms email available validator

```
export function emailAvailableValidator(api: AuthApi)
  : AsyncValidatorFn {
  return (control: AbstractControl) => {
    if (!control.value) {
      return of(null);
    }

    return timer(300).pipe(
      switchMap(() => {
        return api.checkEmailAvailability(control.value));
      }
      map(result =>
        result.available ? null : { emailTaken: true }
      ),
      catchError(() => of(null))
    );
  };
}
```

**Listing 2:** Signal Forms form setup and email available validator

```
readonly registrationForm = form(
  this.registrationModel,
  (s) => {
    required(s.email, {
      message: 'Email is required'
    });
    email(s.email, {
      message: 'Enter a valid email'
    });

  s.email.validateAsync(async (email) => {
    if (!email) return null;

    const result =
      await this.authApi.checkEmailAvailability(email);

    return result.available
      ? null:
      {
         emailTaken: 'This email is already registered'
      };
  });

  required(s.password, {
    message: 'Password is required'
  });
  required(s.acceptTerms, {
    message: 'You must accept the terms'
  });
});
```

While the async validator is running, the field is pending. That fact can be rendered directly.

```
@if (registrationForm().email.pending()) {
  <span>Checking email availability…</span>
}

@if (registrationForm().email.invalid()) {
  <span>{{ registrationForm()
    .email.errors() }}</span>
}
```

There is no choreography here. The UI reflects what the form knows to be true at that moment.

This matters because pending is not cosmetic. It influences whether submission should proceed, whether the UI should be disabled, and how confident the system is in its validity snapshot.

When pending is modelled as truth rather than as a side channel, those decisions become straightforward.

### *Submission as Intent, Not Consequence*

At this point, it is tempting to ask why submission is not automatic. If validity is reactive, why not submit as soon as the form becomes valid?

Because submission is not the truth. Submission is intent.

Submitting a form is a deliberate transition with irreversible consequences. Treating it as a reaction to the state would collapse an essential boundary.

Signal Forms enforces that boundary explicitly.

```
onSubmit(event: Event) {
  event.preventDefault();
  submit(this.registrationForm, async () => {
    const dto = this.registrationForm.value();
    await this.authApi.register(dto);
  });
}
```

The form guarantees validity. The developer declares what submission means. Server errors flow back into validation, preserving a single error model.

Validation describes reality. Submission expresses choice.

### *Autosave: the Second Fault Line*

If async validation reveals architectural cracks, autosave widens them.

Autosave is rarely essential for correctness. Because of that, it is often added late, incrementally, and without revisiting earlier assumptions. Over time, it becomes entangled with validation, submission, and UI state until the form feels fragile.

To understand why, we need to look at autosave as it is typically implemented in a mature reactive forms application.

```
this.form.valueChanges.pipe(
  debounceTime(500),
  distinctUntilChanged((a, b) => deepEqual(a, b)),
  filter(() => this.form.dirty),
  switchMap(value =>
    this.draftApi.save(value).pipe(
      catchError(() => EMPTY)
    )
  ),
  takeUntil(this.destroy$)
).subscribe();
```

This code reflects experience. It avoids excessive network traffic. It prevents redundant saves. It cancels outdated requests. It cleans itself up.

It also encodes autosave as time.

To understand its correctness, a developer must imagine sequences of typing, waiting, cancelling, retrying, and tearing down. As new requirements appear, pausing autosave during submission, saving only when certain fields are complete, and the choreography grows.

This is where the traffic-light model breaks down again.

### *A Second Analogy: the Notebook Versus the Tape Recorder*

At this point, a smaller analogy helps.

Imagine taking notes during a meeting.

One approach is to record everything on tape and later reconstruct what matters by replaying the audio. This is accurate, but it is exhausting. Understanding the outcome requires replaying time.

Another approach is to maintain a living notebook. At any moment, the notebook reflects your current understanding. When something changes, you update the page. There is no replay. There is only now.

Event-driven autosave is the tape recorder. State-driven autosave is the notebook.

Signal Forms pushes autosave toward the notebook model.

## Autosave as a Consequence of the State

In the signal-first model, autosave is not a reaction to keystrokes. It is a consequence of truth.

The form already knows whether it has been modified, whether validation is pending, whether submission is active, and what the current model looks like. Autosave logic can therefore be expressed as policy rather than choreography.

```
constructor() {
  effect(() => {
    const form = this.registrationForm();
    const model = this.registrationModel();
    if (!form.dirty()) return;
    if (form.pending()) return;
    if (!form.valid()) return;
    this.draftApi.save(model);
  });
}
```

This effect does not manage time. It evaluates conditions.

If the user keeps typing, the effect re-runs with the updated state. If submission begins, autosave pauses. If validation is pending, autosave waits until the system has a stable view of correctness.

There is no cancellation logic because cancellation is no longer the problem. Relevance is.

## When Async Goes Wrong in Real Teams

Most teams do not break their form architecture all at once. It erodes slowly, in response to reasonable decisions made under time pressure.

A common story starts with async validation. Email availability is added first, implemented cleanly as a custom async validator. Autosave follows a sprint later, implemented as a debounced valueChanges pipeline. Submission state is added after a production incident, guarded by flags. Each change is locally correct. The system still works. Tests still pass.

The problem emerges months later, when no one on the team can confidently answer simple questions.

- Why does autosave sometimes stop after a failed submission?
- Why does the email field show an error briefly and then clear itself?
- Why does disabling the form during submission sometimes cancel validation and sometimes not?

At that point, the form is no longer understood as a system. It is understood as a collection of behaviors that happen to coexist. Developers stop making changes unless absolutely necessary, because every change risks breaking a timeline they no longer fully grasp.

This is the first failure mode: temporal entanglement.

Async validation, autosave, and submission are all expressed as reactions to events. Each concern introduces its own timing assumptions. Over time, those assumptions overlap. When behavior is wrong, the only way to debug it is to reconstruct sequences of emissions, cancellations, retries, and flag transitions.

This is exactly the traffic-light model failing at scale. The city still functions but diagnosing congestion requires replaying the entire day.

## The Slow Drift of the Submission State

Another common failure mode appears around submission.

A team introduces a Boolean like isSubmitting to disable inputs and prevent duplicate requests. At first, this flag is set before the request and cleared afterward. It works.

Later, async validation is added. Now submission must wait for pending validators. The flag is moved. Another flag is added. Error handling becomes conditional. Eventually, submission state is no longer a single truth. It is inferred from several variables that must stay in sync.

This is where bugs become subtle.

A failed submission clears isSubmitting, but async validation restarts and disables the submit button again. A retry works locally but fails in production due to timing differences. No one changed the validation logic—but it still affects submission behavior.

This happens because submission state is being coordinated externally rather than derived from form state. The form knows whether it is pending, invalid, or stable, but submission logic is operating on a parallel channel.

In a signal-based model, this drift is much harder to introduce. Submission is explicit. Pending state is observable. Disabling behavior is derived rather than toggled. There is still complexity, but it is anchored.

This is the second failure mode: state duplication.

## Recreating RxJS Inside Effect()

The third failure mode is more subtle, and it only appears after teams adopt signals enthusiastically.

Developers who are comfortable with RxJS begin to recreate familiar patterns inside effect(). Debouncing logic is added manually. Local variables track previous values. Conditional guards multiply. Before long, the effect contains more coordination logic than the original stream.

At that point, the benefit of signals evaporates. Time-based reasoning sneaks back in. The effect becomes a subscription in disguise.

What went wrong is not misuse of the API. Developers started to misuse the mental model.

Effects are not meant to orchestrate time. They are meant to express consequences of truth. When developers try to force temporal choreography into them, they recreate the same problems they were trying to escape.

This failure mode is less about code and more about habit. Teams carry their old instincts forward and apply them to a new abstraction without adjusting how they think.

Signal Forms do not eliminate complexity automatically. It rewards discipline. When used as intended, it prevents these failure modes. When misused, it simply relocates them.

## Why Complexity Stops Compounding

The real advantage of the signal-first model is not that it handles async better in isolation. It is that complexity does not multiply as requirements grow.

In event-driven systems, every new async concern interacts with every existing one. Validation timing affects autosave. Autosave timing affects submission. Submission timing affects UI state. Each interaction introduces another axis of reasoning.

In a state-driven system, new concerns attach to existing truths rather than to timelines. Async validation attaches

to validity. Autosave attaches to dirtiness and stability. Submission attaches to explicit intent.

The system still grows, but it grows linearly. New rules add conditions, not timelines. New features add state dependencies, not event choreography.

This is why the traffic analogy matters. A city designed around current conditions absorbs growth more gracefully than one designed around fixed sequences of reactions.

Forms are no different.

## What Changes After Six Months of Using this Model

The most telling shift does not appear in code. It appears in conversations.

Teams stop asking "what fires when?" and start asking "under what conditions should this happen?" Debugging sessions become shorter because state can be inspected directly. Features like autosave and async validation stop feeling dangerous to touch.

Most importantly, forms stop being treated as fragile artifacts. They become systems that can evolve.

That is the real test of an abstraction. Not whether it works on day one, but whether it remains understandable long after the original author has moved on.

## Async Without Anxiety

Async validation works in Signal Forms because it is expressed as truth. Autosave works because it is expressed as a consequence. Submission remains explicit because it expresses intent.

Asynchrony still exists. That is the difference between coordinating behavior and modeling reality.

## The Illusion of Steps

The first shift is conceptual. A multi-step form does not actually have steps. It has state that is revealed progressively.

This distinction matters because once steps are treated as independent units, developers begin to split state across multiple form groups, each with its own validation lifecycle, submission logic, and coordination rules. Navigation becomes a problem of synchronizing independent systems rather than moving through a single, evolving truth.

In a signal-based model, the form remains one structure:

```
readonly registrationModel = signal({
 account: {
   email: '',
   password: '',
 },
 profile: {
   firstName: '',
   lastName: '',
 },
 preferences: {
   newsletter: false,
 }
});
```

The UI introduces steps, not the data model.

```
readonly currentStep = signal(0);
```

Each step becomes a projection of the same underlying state:

```
@if (currentStep() === 0) {
 <!-- account fields -->
}

@if (currentStep() === 1) {
 <!-- profile fields -->
}

@if (currentStep() === 2) {
 <!-- preferences -->
}
```

Nothing about validation, async rules, or submission changes because the form itself has not been fragmented. The system still answers a single question: what is true right now?

Steps are a UI concern. State remains unified.

### *Partial Validity as a First-Class Concept*

The moment a form spans multiple steps, validity stops being binary.

A form can be globally invalid while still allowing forward progression. A step can be "complete enough" to proceed even if other parts of the model are untouched. In traditional approaches, this leads to step-level validators, duplicated rules, or conditional checks scattered across navigation logic.

In a signal-first model, partial validity is not inferred. It is expressed.

```
const stepValidity = computed(() => {
 const form = this.registrationForm();

 switch (this.currentStep()) {
   case 0:
     return form.account.valid();
   case 1:
     return form.profile.valid();
   case 2:
     return true; // optional step
 }
});
```

Navigation becomes a consequence of the state:

```
nextStep() {
 if (!this.stepValidity()) return;
 this.currentStep.update(s => s + 1);
}
```

What changes here is subtle but important. We are not "running validation before navigation." We are asking

whether the current slice of state satisfies the conditions required to proceed.

This eliminates an entire category of bugs where navigation and validation drift apart, because both are derived from the same underlying truth.

### *Cross-Field Dependencies Without Coordination*

Multi-step flows often introduce dependencies that span across steps. A decision made early in the process influences what is required later. A field in one step may invalidate a field in another. In event-driven systems, this typically results in subscriptions or imperative checks that attempt to "keep things in sync."

This is where the signal model becomes decisive.

Dependencies are not coordinated. They are declared.

```
readonly isBusinessAccount = computed(() => {
 return this.registrationModel().
    account.email.endsWith('@foo.com');
});
```

Validation can then depend on that derived truth (**Listing 3**).

No subscriptions. No manual synchronization. No step awareness baked into validation.

The system evaluates the current state and derives correctness from it. If the email changes, the dependency updates automatically. If the user navigates back and modifies earlier input, downstream validation reflects that change without additional code.

This is the difference between coordination and composition.

### *Async Rules Across Multiple Pieces of State*

Async validation becomes more complex in multi-step flows because it often depends on more than a single field. Consider a scenario where availability depends on both email and region, or where pricing validation depends on a combination of selected options.

In traditional systems, this introduces combinatorial complexity: multiple streams must be merged, debounced, cancelled, and coordinated.

In a signal-first model, async validation still answers the same question: what is true right now?

```
s.account.validateAsync(async (account) => {
 const { email } = account;
 const region =
   this.registrationModel().profile.region;

 if (!email || !region) return null;

 const result = await this.api
   .checkAvailability(email, region);

 return result.available
   ? null :
    { notAvailable:
        'This combination is not allowed'
    };
});
```

The validator reads from multiple signals. It does not subscribe to them.

When either input changes, the validation becomes stale. The system does not need to cancel anything explicitly because the previous result no longer corresponds to the current state. This is the same principle introduced earlier, now applied across boundaries.

Async complexity does not increase with the number of dependencies. It remains constant because the abstraction does not change.

### *Debounce Without Reintroducing Time*

One of the common mistakes when dealing with multi-step async logic is reintroducing temporal reasoning through manual debouncing. Developers attempt to optimize network calls by layering timing logic into effects or validators, often recreating the very coordination problems they were trying to escape.

Angular Signal Forms now provides a built-in debounce capability, which allows us to express delay without collapsing back into event streams.

```
s.account.email.debounce(300).validateAsync(async
 (email) => {
 if (!email) return null;

 const result = await this.api.checkEmailAvailability(email);

 return result.available ? null :
   { emailTaken: true };
});
```

What matters here is not the delay itself, but where it lives.

The debounce is attached to the signal, not orchestrated externally. It becomes a property of how the state stabilizes, not a timeline the developer must manage. There is no switchMap, no cancellation logic, no replaying of emissions. The system still evaluates truth; it simply waits for input to settle before doing so.

**Listing 3:** Validating an account

```
readonly registrationForm = form(this.registrationModel,
  (s) => {
   required(s.account.email);
   required(s.account.password);

   const lastNameReq =
     'Last name is required for business accounts';

   s.profile.validate((profile) => {
     if (this.isBusinessAccount() && !profile.lastName) {
       return {
         lastNameRequired: lastNameReq
       };
     }
     return null;
   });
});
```

This is an important boundary. The moment debounce logic leaks into effects or navigation, time-based reasoning returns. Keeping it at the signal level preserves the mental model.

## Multi-Step Flows as State Composition

At scale, the challenge is not validation, navigation, or async behavior in isolation. The challenge is composing all of them without losing the ability to reason about the system.

A multi-step form built on signals remains understandable because every concern attaches to state. Step progression is derived from partial validity. Cross-step dependencies emerge through computed relationships. Async rules evaluate combinations of current state rather than isolated inputs. Debounce stabilizes input without introducing timelines, and submission remains an explicit transition rather than a side effect of navigation.

Nothing is coordinated through events. Nothing requires reconstructing a sequence of actions to explain behavior.

This is the discipline that prevents collapse.

### *When Traditional Abstractions Fail*

This is the point where most traditional abstractions break down.

They were designed around individual controls, not evolving systems. They assume validation happens locally, not across distributed state. They treat async behavior as something to coordinate rather than something to derive.

As requirements grow, developers compensate by adding layers: step-level forms, shared services, synchronization flags, and increasingly complex pipelines. Each layer solves a local problem while making the global system harder to reason about.

Eventually, the form becomes something teams avoid touching.

The signal-first model does not avoid complexity. It changes where complexity lives.

Instead of spreading across timelines and coordination logic, complexity is concentrated in state relationships. That concentration is what makes it manageable.

### *The Real Outcome*

After implementing multi-step flows in this model for a while, the shift is noticeable. Developers stop asking how to synchronize steps. They start asking what conditions define progress. They stop debugging sequences of events. They inspect current state. They stop fearing async behavior. They treat it as another property of truth. This is not a small improvement.

It is a change in how forms are understood.

A multi-step form is no longer a fragile workflow stitched together over time. It becomes a system that can grow without losing coherence.

And that is the real goal.

## From Coordination to Composition: What Actually Changed

This article was never about introducing a new API surface. It was about removing an entire category of problems that most teams have quietly accepted as unavoidable.

At the beginning, the focus was deliberately narrow. A single form. A single model. A controlled environment where the core idea could be established without distraction. That idea was simple, but not trivial: form state is not something that unfolds over time. It is something that exists, continuously, as truth.

Asynchronous validation was the first pressure test. Not because it is conceptually difficult, but because it exposes how fragile event-driven systems become once time enters the picture. Debounce windows, cancellation semantics, and race conditions are not inherent to validation. They are artifacts of how validation is expressed. When validation is modeled as a property of state, those concerns do not disappear, but they stop being something the developer must actively manage.

Autosave extended that pressure further. What is often treated as a background feature quickly turns into a coordination problem in traditional systems. Streams are composed, emissions are filtered, and correctness becomes tied to sequences of events. In a signal-first model, au-

tosave becomes something fundamentally different. It is no longer a pipeline. It is a policy. It answers a question: under what conditions should the current state be persisted? Once that question is answered declaratively, the need to reason about timing fades.

The failure modes that followed were not edge cases. They were the natural consequences of mixing timelines, duplicating state, and carrying over habits from event-driven systems. Temporal entanglement, submission drift, and recreating RxJS inside effect() are not mistakes made by inexperienced teams. They are what happens when the abstraction does not match how the system is being reasoned about.

This is where the transition to multi-step forms becomes critical.

A multi-step flow is not a new feature layered on top of a form. It is a multiplication of complexity. Partial validity, cross-step dependencies, and async rules that depend on combinations of state introduce interactions that quickly overwhelm traditional approaches. The moment state is fragmented across steps, correctness becomes something that must be coordinated again. Navigation logic, validation logic, and async behavior begin to drift apart.

The signal-first model holds precisely because it does not change under this pressure.

The form remains a single structure. Steps are not units of state; they are projections of it. Partial validity is not inferred through flags; it is derived from the current state. Cross-field and cross-step dependencies are not synchronized manually; they are expressed as relationships. Async validation continues to evaluate truth, even when that truth depends on multiple inputs. Debounce, now part of the signal model itself, stabilizes input without reintroducing timelines.

Nothing new is added in terms of mental overhead. The same rules apply. That is the point.

What changes is not capability. It is stability under growth.

In event-driven systems, every new requirement introduces new interactions between timelines. Validation affects autosave. Autosave affects submission. Submission affects UI state. Each interaction compounds complexity. Over time, the system becomes something that works, but cannot be easily reasoned about.

In a state-driven system, new requirements attach to existing truths. Validation attaches to correctness. Autosave attaches to stability. Navigation attaches to partial validity. Submission remains an explicit boundary. The system grows, but it does not lose coherence.

This is the difference between coordination and composition. After working in this model for a while, the most noticeable change is not in the codebase. It is in how teams talk about forms. Questions shift from “what happens when this fires?” to “under what conditions should this be true?” Debugging stops being an exercise in reconstructing timelines and becomes an exercise in inspecting state. Features that once felt risky to modify, async validation, autosave, multi-step flows, become predictable.

That predictability is not accidental. It is a consequence of choosing the right abstraction.

The goal was never to eliminate asynchrony, nor to simplify forms to the point of triviality. Real-world forms are inherently complex. They deal with incomplete data, delayed responses, conditional requirements, and user intent that evolves over time.

The goal was to ensure that this complexity does not compound.

Signal Forms achieves this not by reducing what the system can do, but by ensuring that everything it does can be understood as a function of what is true right now.

That is the difference.

Not between synchronous and asynchronous code.

Not between simple and complex forms.

But between systems that must be coordinated…and systems that can be composed.

Sonu Kapoor

# Advanced Operations Using .NET Aspire

**Joydip Kanjilal**
joydipkanjilal@yahoo.com

Joydip Kanjilal is an MVP (2007-2012), software architect, author, and speaker with more than 20 years of experience. He has more than 16 years of experience in Microsoft .NET and its related technologies. Joydip has authored eight books, more than 500 articles, and has reviewed more than a dozen books.

.NET Aspire is a unified, cloud-ready software development stack designed to efficiently handle orchestration, configuration, service integration, and observability in distributed applications. You can use Aspire to simplify the entire lifecycle of distributed applications—from development to deployment and monitoring—while promoting consistent architecture and operational practices that can help make your applications scalable, reliable, and easy to maintain. Additionally, with a vast set of tools at your disposal, by using Aspire, you can streamline the communication between services and infrastructure components throughout the application development process.

If you're going to work with the code examples discussed in this article, you need the following installed on your system:

- Visual Studio 2026
- .NET 10.0
- ASP.NET 10.0 Runtime
- Entity Framework Core
- Azure CLI
- Aspire CLI

If you don't already have Visual Studio 2026 installed on your computer, you can download it from here: https://visualstudio.microsoft.com/downloads/.

> There is a common misconception that Aspire is used only for building microservices-based applications. In fact, you can use Aspire to build distributed applications that can comprise several services and external dependencies. For example, you can have a frontend that uses Blazor, one or more backend services that use Web APIs, one or more queues that use RabbitMQ, databases such as SQL Server, PostgreSQL, and external APIs.

## Integrations in .NET Aspire

In .NET Aspire, integrations are used to attach additional features such as health checks, logging, caching, etc. These integrations are available in .NET Aspire as prebuilt NuGet packages. You can use integrations in your preferred IDE such as Visual Studio or Visual Studio Code, or via the .NET CLI. You can use .NET Aspire to connect to an external service or component, typically available as a NuGet package, enabling you to connect to a database, cache, queue, message broker, storage device, or cloud platform.

Using .NET Aspire integration, you can describe what happens in the AppHost at runtime without writing boilerplate code to define connection strings, health checks, telemetry, etc., simply by installing the NuGet package and referencing it. Aspire will connect them together automatically. In this section, we will discuss how Aspire integrations work and provide implementations you can apply directly to your own cloud-compatible solutions.

### *The Need for Integrations*

In a monolithic application, you can manage dependencies easily—by referencing libraries, injecting them into services, etc. However, things become more complex when you're working with a distributed system. For example, consider an e-commerce application that comprises several components and services needed to retrieve service or component health information, implement graceful failures, and capture and store logs and traces for observability.

If you don't have a proper integration strategy in place, you'll need to hard-code health-check logic throughout the application. Moreover, if you don't use integrations, your application will exhibit inconsistent logging and tracing patterns. Your application will be fragile and difficult to test and maintain. .NET Aspire simplifies these issues by providing support for integrations using a declarative, composable model.

### *Types of Integrations*

Aspire integrations can be roughly categorized into two categories; hosting integrations that provide a way to

describe resources in the AppHost, such as containers or cloud services, and client integrations that provide a means of connecting to libraries, such as Redis, with support for dependency injection, configuration, health checks, telemetry, and testability.

.NET Aspire includes built-in support for integrations with popular cloud-native services using one or more of the following:

- **Client libraries:** You can connect your .NET Aspire application to external services and components, such as Azure Service Bus, Redis, or SQL Server or PostgreSQL databases, using configurable client libraries.
- **Built-in support for telemetry:** You can leverage the built-in support for logging, metrics, and tracing in your .NET Aspire application.
- **Register services using dependency injection:** You can register services with the dependency injection container and use them in your .NET Aspire application.

### *Hosting Integrations*

Hosting integrations are used to provide infrastructure support in a .NET Aspire application via the AppHost project. Using hosting integrations enables you to connect your .NET Aspire application to external resources like databases, caches, message brokers, storage devices, and services.

Hosting integrations in .NET Aspire extend the IDistributedApplicationBuilder interface thereby enabling the AppHost project to express resources within the app model. Hosting integrations can be used to create a resource by spinning up a docker container or configuring a cloud-hosted service and creating environment variables that represent the endpoints, ports, and connection strings of the resource. You can use hosting integrations to inject the configuration of the resource into any project that depends on it.

> Note that hosting integrations are not limited to .NET projects. You can use a hosting integration in a Python application or a Node.js-based container, if the application or service has access to the injected environment variables from Aspire.

The InventoryManagement.AppHost project is where you can configure and orchestrate the application components. Here's how a typical AppHost project is organized:

```
InventoryManagement/
├── InventoryManagement.AppHost/
        (Service orchestration)
├── InventoryManagement.ServiceDefaults/
        (Shared configuration)
├── InventoryManagement.Web/
        (Web application)
├── InventoryManagement.Api/
        (API service)
└── InventoryManagement.Database/
        (Database initialization)
```

### *Client Integrations*

Client integrations use dependency injection (DI) to define configuration schema and create health checks, resiliency, and telemetry. Note that client integration libraries are prefixed with the word "Aspire" followed by the full package name—for example, Aspire.StackExchange.Redis.

The following snippet shows the Program.cs file of a typical AppHost project:

```
var builder =
  DistributedApplication.CreateBuilder(args);

var database = builder
  .AddPostgres("postgres")
  .AddDatabase("inventorydb");

var redis = builder
  .AddRedis("redis");

var messageQueue = builder.AddRabbitMq("rabbitmq");

builder.AddProject<
  Projects.InventoryManagement_Web>("webfrontend")
    .WithExternalHttpEndpoints()
    .WithHttpHealthCheck("/health")
    .WithReference(cache)
    .WithReference(database)
    .WithReference(redis)
    .WaitFor(cache)
    .WithReference(apiService)
    .WaitFor(apiService);

builder.Build().Run();
```

### *SQL Server Integration*

Aspire provides support for SQL Server integration in the same way it supports PostgreSQL integration. The first step in configuring a SQL Server project is to define the resource—SQL Server—as an AppHost and create a new database in SQL Server. The database can then be referenced from your consuming project. Note that you can use a connection string to provision a SQL Server container and also connect to an existing SQL Server instance.

The following code snippet shows how you can configure SQL Server integration in a .NET Aspire application.

```
var builder =
  DistributedApplication.CreateBuilder(args);

var sqlServer = builder.AddSqlServer("sql")
  .AddDatabase("inventorydb");

var api = builder.AddProject
  <Projects.InventoryManagement_ApiService>
  ("apiservice").WithReference(sqlServer);
builder.Build().Run();
```

### *Redis Integration*

Caching is a technique that can be used to store relatively stale data for faster retrieval when needed by the application. Redis is an open-source, high performance, in-memory data store available for commercial and non-commercial use to store and retrieve data in your applications.

Redis supports several data structures such as hashes, lists, sets, sorted sets, bitmaps, etc. Used primarily as a database, cache, or message broker, you'll notice only negligible performance overhead when reading or writing data using Redis. Redis is an excellent choice if your application requires a large amount of data to be stored and retrieved, and memory availability is not an issue.

Redis is often used for Aspire Integration since it serves as not only a great cache, but also as a performant data store. Aspire can create a local Redis container or connect to an already existing Redis instance and the corresponding client integration provides your application DI-friendly access to Redis with additional telemetry and health checking, all in a single integration layer.

The following code snippet shows how you can integrate Redis into your .NET Aspire application.

```
var builder =
  DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
  .AddDatabase("inventorydb");

var redis = builder.AddRedis("cache")
  .WithDataVolume();

var api = builder.AddProject
  <Projects.InventoryManagement_ApiService>
    ("apiservice")
      .WithReference(postgres)
      .WithReference(redis);

var web = builder.AddProject
  <Projects.InventoryManagement_Web>("web")
    .WithReference(api)
    .WithReference(redis);
builder.Build().Run();
```

### *RabbitMQ Integration*

RabbitMQ is an open-source, high-performance messaging broker that takes advantage of the Open Telecom Platform and implements AMQP (an acronym for Advanced Message Queuing Protocol) for communication across applications, processes, and servers.

RabbitMQ fits into the Aspire context when you want to produce or dispatch messages to a message queue without pausing to wait for a response from the message consumer. When you create an instance of RabbitMQ in Aspire, it automatically provisions a RabbitMQ resource and automatically injects configuration settings into services that produce or consume messages to and from RabbitMQ, including health checks and lifecycle behavior.

The following code snippet shows how you can configure RabbitMQ integration in a .NET Aspire project.

```
var builder =
  DistributedApplication.CreateBuilder(args);

var postgres = builder.AddPostgres("postgres")
  .AddDatabase("inventorydb");

var rabbitmq = builder.AddRabbitMQ("messaging");

var api = builder.AddProject
  <Projects.InventoryManagement_ApiService>
  ("apiservice")
    .WithReference(postgres)
    .WithReference(rabbitmq);

var worker = builder.AddProject
  <Projects.InventoryManagement_Worker>("worker")
    .WithReference(rabbitmq);

var web = builder.AddProject
  <Projects.InventoryManagement_Web>("web")
    .WithReference(api);
builder.Build().Run();
```

## Implementing a Real-life Application Using .NET Aspire

The term "inventory management system" refers to an application that supports a business in the management of its inventory. It allows you to gain greater visibility into product availability and purchasing trends, while capabilities such as reorder alerts and stock monitoring will assist with controlling costs and keeping your operation running smoothly. You can use it to track units of merchandise being held, the location of inventory items, and how much inventory flows in and out of your warehouse facility. Additionally, an inventory management system helps minimize stock outages, avoid excess stock, and enhance business efficiency.

For the purposes of this discussion, we will focus primarily on the product and purchase order modules, including the product, warehouse, inventory, purchase order, and purchase order detail tables. These core tables will enable you to model product master data, the quantities of product held in stock, and purchasing activities without adding unnecessary complexity to your overall inventory system. The sections that follow walk through implementing the application.

We'll start by creating the inventory system database. **Listing 1** contains the complete database script for creating the product, warehouse, inventory, purchase order, and purchase order detail tables. Next, we'll create the solution structure for the application by creating the necessary projects in Visual Studio. You can create a project in Visual Studio 2026 in several ways. When you launch Visual Studio 2026, you'll see the "Getting started" window. You can choose "Continue without code" to launch the main screen of the Visual Studio 2026 IDE.

To create a new ASP.NET Core 10 Project in Visual Studio 2026:

1. Start the Visual Studio 2026 IDE.
2. In the Create a new project window, select **Aspire Starter App**, and click **Next** to move on as shown in **Figure 1**.

**Listing 1:** Create database tables

```
DROP TABLE IF EXISTS inventory CASCADE;
DROP TABLE IF EXISTS purchase_order CASCADE;
DROP TABLE IF EXISTS warehouse CASCADE;
DROP TABLE IF EXISTS product CASCADE;

DROP TABLE IF EXISTS purchase_order_detail CASCADE;

CREATE TABLE product (

    product_id serial PRIMARY KEY,
    product_code varchar(100) NOT NULL UNIQUE,
    barcode varchar(100) UNIQUE,
    product_name varchar(100) NOT NULL,
    product_description varchar(2000),
    product_category varchar(100),
    reorder_quantity int CHECK
(reorder_quantity IS NULL

OR reorder_quantity > 0),
    created_date timestamp(0)

NOT NULL DEFAULT CURRENT_TIMESTAMP,
    modified_date timestamp(0)

);


CREATE TABLE warehouse (
    warehouse_id serial PRIMARY KEY,
    warehouse_name varchar(100) NOT NULL,
    warehouse_address varchar(200),
    is_refrigerated boolean

NOT NULL DEFAULT false,
    created_date timestamp(0)
NOT NULL DEFAULT CURRENT_TIMESTAMP,
    modified_date timestamp(0)
);


CREATE TABLE purchase_order (

    purchase_order_id serial PRIMARY KEY,
    order_number varchar(30) NOT NULL UNIQUE,
    order_date date NOT NULL,
    order_status varchar(30)

NOT NULL DEFAULT 'Open'
        CHECK (order_status IN
('Open', 'PartiallyReceived',
'Closed', 'Cancelled')),
    remarks varchar(500),
    created_date timestamp(0)
NOT NULL DEFAULT CURRENT_TIMESTAMP,
    modified_date timestamp(0)
);


CREATE TABLE inventory (
    inventory_id serial PRIMARY KEY,
    product_id int NOT NULL
REFERENCES product(product_id),
    warehouse_id int NOT NULL
REFERENCES warehouse(warehouse_id),
    quantity_available int
NOT NULL DEFAULT 0 CHECK
(quantity_available >= 0),
    minimum_stock_level int
CHECK (minimum_stock_level IS NULL
OR minimum_stock_level >= 0),
    maximum_stock_level int
CHECK (maximum_stock_level IS NULL
maximum_stock_level >= 0),
    reorder_point int CHECK
(reorder_point IS NULL OR reorder_point >= 0),
    created_date timestamp(0)
NOT NULL DEFAULT CURRENT_TIMESTAMP,
    modified_date timestamp(0),
    UNIQUE (product_id, warehouse_id),
    CHECK (minimum_stock_level
IS NULL OR maximum_stock_level
IS NULL OR minimum_stock_level
<= maximum_stock_level),
    CHECK (reorder_point
IS NULL OR minimum_stock_level
IS NULL OR reorder_point >=
minimum_stock_level)
);

CREATE TABLE purchase_order_detail (
    purchase_order_detail_id
serial PRIMARY KEY,
    purchase_order_id int
NOT NULL REFERENCES purchase_order

(purchase_order_id),
    product_id int

NOT NULL REFERENCES product(product_id),
    warehouse_id int

NOT NULL REFERENCES warehouse(warehouse_id),
    order_quantity int

NOT NULL CHECK (order_quantity > 0),

    expected_date date,
    actual_date date,
    created_date timestamp(0)

NOT NULL DEFAULT CURRENT_TIMESTAMP,

    modified_date timestamp(0),
    CHECK (actual_date IS
NULL OR expected_date IS NULL
OR actual_date >= expected_date)
);
```

3. Specify the project name as InventoryManagement and the path where it should be created in the **Configure your new project** window as shown in **Figure 2**.
4. If you want the solution file and project to be created in the same directory, you can optionally check the **Place solution and project in the same directory** checkbox. Click **Next** to move on.
5. In the next screen, specify the target framework. Ensure that the "Configure for HTTPS" and "Use Redis for caching…" checkboxes are checked as shown in **Figure 3**.
6. Click **Create** to complete the process.

This creates a new .NET Aspire application in Visual Studio. At first glance, the Solution Explorer of this application will look similar to **Figure 4**.

In this example, the proposed solution comprises the following four projects:

- InventoryManagementSystem.AppHost
- InventoryManagementSystem.Api
- InventoryManagementSystem.Web
- InventoryManagementSystem.ServiceDefaults

The API project contains the entity mappings for each of the entities—product, warehouse, inventory, purchase_order, and purchase_order_detail—along with the repository types and controllers.

By default, the AppHost is the starter application: when the application starts, the AppHost is executed.

### *Explore the .NET Aspire Dashboard*

The .NET Aspire Dashboard is a web-based tool that provides a real-time, comprehensive overview of your projects' status, dependencies, and performance metrics. By providing you with a unified view of your application, the

Aspire dashboard can help you to quickly identify issues and optimize your development workflow. It is launched in the web browser, along with other projects in the solution, when you start the AppHost project. In this example, the ShoppingCart.ApplicationApiService and the ShoppingCartApplication.Web projects will be automatically launched when you invoke the AppHost project in your solution.

The information presented in the .NET Aspire Dashboard can be organized into the following sections:

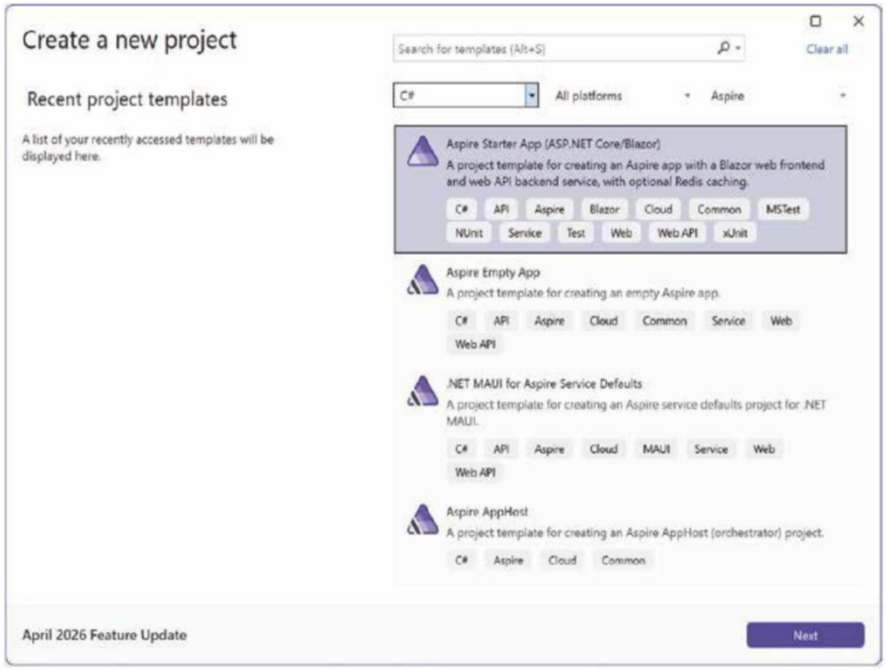



**Figure 1:** Creating a new project in .NET Aspire

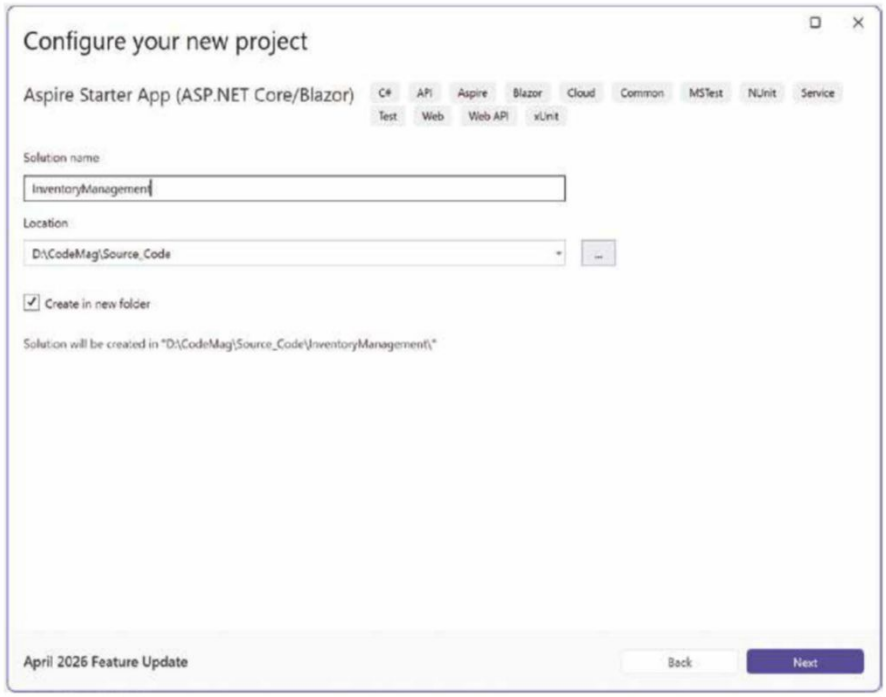



**Figure 2:** Configuring a new project

- **Resources:** This section contains important information about every .NET project in your .NET Aspire configuration. It enables you to view the status of your application, the endpoint addresses used by your application, and the environmental variables loaded by your application.
- **Console:** This section displays the textual output sent to the console, which can be used for tracking events and displaying status messages.
- **Structured:** In this section, the content is organized into a tabular format, enabling you to filter and search the content, and also expand the log details if need be.
- **Traces:** This section helps you to trace the complete paths of all requests sent to your application starting from the origin to the endpoint that produces the response. You can take advantage of this feature to determine how long it takes to serve requests sent to your application.
- **Metrics:** This section contains instrumentation and metering for your application and provides optional filters based on the available dimensions of the reported metrics.

### *Install Entity Framework Core*

The next step is to install the necessary NuGet package(s) for working with Entity Framework Core and SQL Server. To install these packages into your project, right-click on the solution and then select **Manage NuGet Packages for Solution....**

Once the window pops up, search for the NuGet packages to add to your project. To do this, type in Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.Design, Microsoft.EntityFrameworkCore.Tools, and Microsoft.EntityFrameworkCore.SqlServer in the search box and install them one after the other. Alternatively, you can type the commands shown below at the NuGet Package Manager command prompt:

```
PM> Install-Package
  Microsoft.EntityFrameworkCore
PM> Install-Package
  Microsoft.EntityFrameworkCore.Design
PM> Install-Package
  Microsoft.EntityFrameworkCore.Tools
PM> Install-Package
  Microsoft.EntityFrameworkCore.SqlServer
```

Alternatively, you can install these packages by executing the following commands at the Windows shell:

```
dotnet add package
  Microsoft.EntityFrameworkCore
dotnet add package
  Microsoft.EntityFrameworkCore.Design
dotnet add package
  Microsoft.EntityFrameworkCore.Tools
dotnet add package
  Microsoft.EntityFrameworkCore.SqlServer
```

If you don't have Aspire project templates installed on your computer, you can install them using the following command:

```
dotnet new install Aspire.ProjectTemplates -force
```

### *Create the Model Classes*

In this example, we'll use the following model classes:

- Product
- Warehouse
- Inventory
- PurchaseOrder
- PurchaseOrderDetail

Here's a brief description of each of these model classes:

- **Product:** A product is what your company purchases, stores, and keeps track of in inventory. When an item is in inventory, it may have various identifiable attributes, including a product code, barcode, product description, product category, and more. These are all identified by the Product which serves as the main entity in the Catalog; it connects the Stock Records with the Purchase Order Line Item for the Inventory and Purchase of any particular Product.
- **Warehouse:** A warehouse is a storage facility or a physical location where you can store and manage your goods. It is described by attributes specific to the location, such as its address.
- **Inventory:** The inventory is the current quantity of a specific product available to a customer within a specific warehouse, along with the controls on this quantity—minimum stock, maximum stock, and reorder point. The purpose of the inventory is to determine whether you've sufficient stock to prevent stock-outs (or out-of-stock) and to trigger a reorder when stock falls below established thresholds.
- **PurchaseOrder:** The purchase order is the header/master record for a procurement transaction intended to replenish stock and contains overall information, such as the order number, order date, status, and comments. The purpose of the purchase order is to manage the supply chain of a supplier purchase order independently of the items on that order; therefore, the purchase order is the parent to all purchase order detail records.
- **PurchaseOrderDetail:** Each purchase order detail record represents the line items that make up a purchase order, connecting a purchase order to a specific product, warehouse, and quantity; it also captures the expected and actual delivery dates. The purchase order detail records capture the actual business intent of the purchase order: which products will be received, the quantity of each, and the warehouse to which they will be delivered.

Create new files named Product.cs, Warehouse.cs, Inventory.cs, PurchaseOrder.cs, and PurchaseOrderDetail.cs with each of them corresponding to the respective model classes inside the Models folder and add code like that shown in **Listing 2**.

### *Create the Data Context*

In Entity Framework Core (EF Core), a data context is a component used by an application to interact with the database, manage database connections, and query and persist data. Next, create the data context class to enable the application to perform CRUD (Create, Read, Update, and Delete) operations against the database.

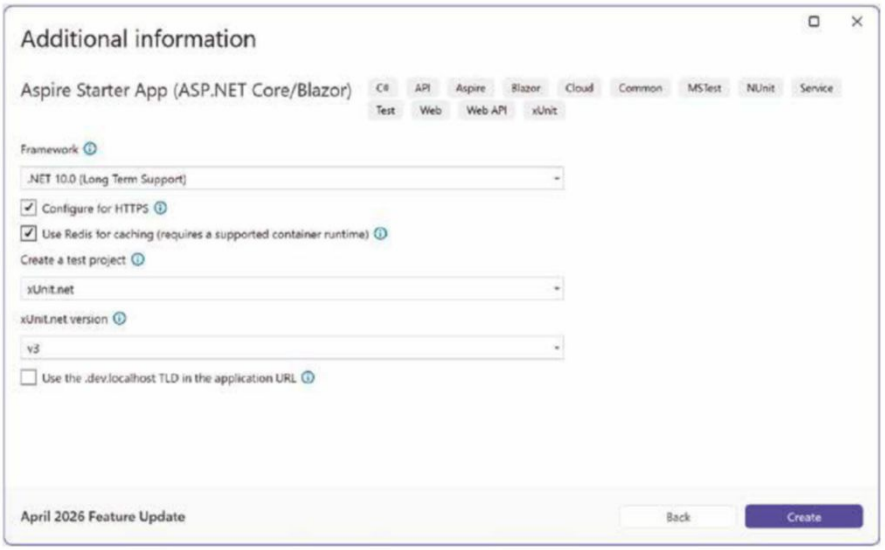



**Figure 3:** Specifying additional information for the project

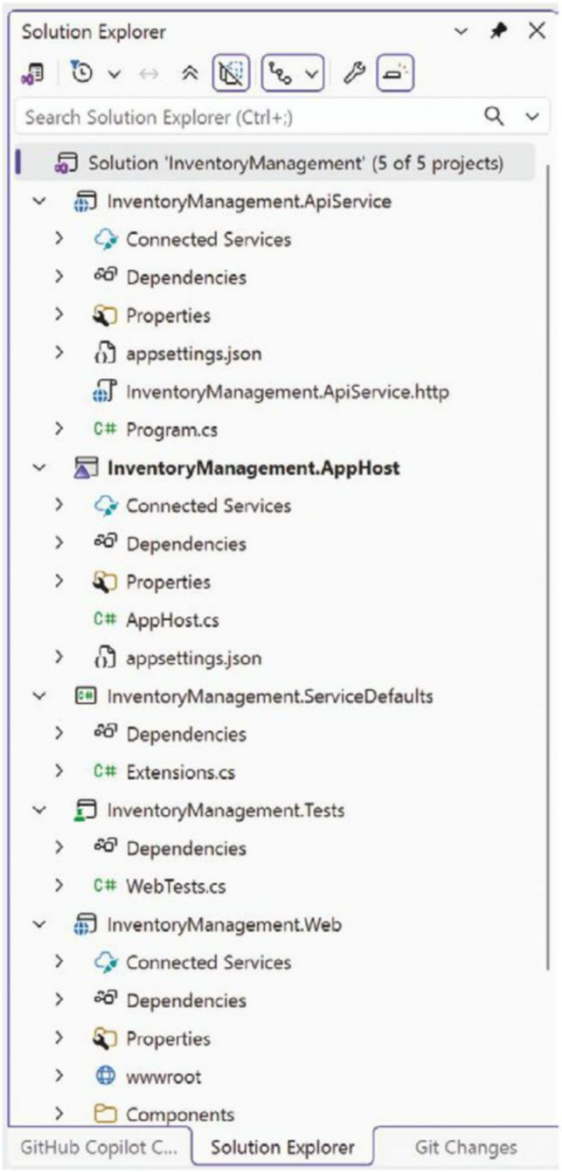



**Figure 4:** The Solution Explorer window

**Listing 2:** Creating the models

```
namespace InventoryManagement.ApiService.Models
{

    public sealed class Inventory
    {
        public int InventoryId { get; set; }
        public int ProductId { get; set; }
        public int WarehouseId { get; set; }
        public int QuantityAvailable { get; set; }
        public int? MinimumStockLevel { get; set; }
        public int? MaximumStockLevel { get; set; }
        public int? ReorderPoint { get; set; }
        public DateTime CreatedDate { get; set; }
        public DateTime? ModifiedDate { get; set; }
        public Product Product { get; set; } = null!;
        public Warehouse Warehouse { get; set; } = null!;
    }
}

namespace InventoryManagement.ApiService.Models
{

    public sealed class Product
    {
        public int ProductId { get; set; }
        public string ProductCode { get; set; } = string.Empty;
        public string? Barcode { get; set; }
        public string ProductName { get; set; } = string.Empty;
        public string? ProductDescription { get; set; }
        public string? ProductCategory { get; set; }
        public int? ReorderQuantity { get; set; }
        public DateTime CreatedDate { get; set; }
        public DateTime? ModifiedDate { get; set; }
        public ICollection<Inventory> Inventories
          { get; set; } = new List<Inventory>();
        public ICollection<PurchaseOrderDetail>
          PurchaseOrderDetails { get; set; } =
          new List<PurchaseOrderDetail>();
    }
}


namespace InventoryManagement.ApiService.Models
{
    public sealed class PurchaseOrder
    {
        public int PurchaseOrderId { get; set; }
        public string OrderNumber { get; set; } = string.Empty;
        public DateTime OrderDate { get; set; }
        public string OrderStatus { get; set; } = "Open";
        public string? Remarks { get; set; }
        public DateTime CreatedDate { get; set; }
        public DateTime? ModifiedDate { get; set; }
        public ICollection<PurchaseOrderDetail>
          PurchaseOrderDetails
          { get; set; } = new List<PurchaseOrderDetail>();
    }
}

namespace InventoryManagement.ApiService.Models
{
    public sealed class PurchaseOrderDetail
    {
        public int PurchaseOrderDetailId { get; set; }
        public int PurchaseOrderId { get; set; }
        public int ProductId { get; set; }
        public int WarehouseId { get; set; }
        public int OrderQuantity { get; set; }
        public DateTime? ExpectedDate { get; set; }
        public DateTime? ActualDate { get; set; }
        public DateTime CreatedDate { get; set; }
        public DateTime? ModifiedDate { get; set; }
        public PurchaseOrder PurchaseOrder
          { get; set; } = null!;

        public Product Product
          { get; set; } = null!;

        public Warehouse Warehouse
          { get; set; } = null!;
    }
}


namespace InventoryManagement.ApiService.Models
{
    public sealed class Warehouse
    {
        public int WarehouseId { get; set; }
        public string WarehouseName
          { get; set; } = string.Empty;
        public string? WarehouseAddress { get; set; }
        public bool IsRefrigerated { get; set; }
        public DateTime CreatedDate { get; set; }
        public DateTime? ModifiedDate { get; set; }
        public ICollection<Inventory>
          Inventories { get; set; } = new List<Inventory>();
        public ICollection<PurchaseOrderDetail>
          PurchaseOrderDetails
          { get; set; } = new List<PurchaseOrderDetail>();
    }
}
```

To do this, create a new class named InventoryDbContext that extends the DbContext class of EF Core and add the following code.

```
public class InventoryDbContext: DbContext
{
    public InventoryDbContext
      (DbContextOptions
      <CartDbContext> options) :
        base(options) { }

public DbSet<PurchaseOrder>
  PurchaseOrders => Set<PurchaseOrder>();

    protected override void
      OnModelCreating(ModelBuilder modelBuilder)
    {
        //Not yet implemented
    }
}
```

You can specify the table and column mapping information in the OnModelCreating overloaded method of the InventoryDbContext class. The following code shows how you can override the OnModelCreating method in your data context class to connect to the database.

```
protected override void
  OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<PurchaseOrder>
  (entity =>
  {
    entity.ToTable("purchase_order");
    entity.HasKey(x => x.PurchaseOrderId);
```

```
    entity.Property(x => x.PurchaseOrderId).
    HasColumnName ("purchase_order_id");

      entity.Property(x =>
        x.OrderNumber).HasColumnName
          ("order_number").HasMaxLength(30).
           IsRequired();
      entity.Property(x =>
        x.OrderDate)
        .HasColumnName("order_date");

      entity.Property(x => x.OrderStatus)
        .HasColumnName
        ("order_status").HasMaxLength(30).
         IsRequired();

       entity.Property(x => x.Remarks)
         .HasColumnName
         ("remarks").HasMaxLength(500);

  entity.Property(x => x.CreatedDate).
    HasColumnName("created_date");

  entity.Property(x => x.ModifiedDate).
    HasColumnName("modified_date");
            entity.HasIndex(x =>
              x.OrderNumber).IsUnique();
        });
}
```

You can also specify the connection string in the appsettings.json file and read it in the Program.cs file to establish a database connection, as shown here:

```
builder.Services.AddDbContext
  <CartDbContext>(options => options.UseSqlServer(
    builder.Configuration.GetConnectionString
    ("DefaultConnection")));
```

**Listing 3** shows the complete source code of the InventoryDbContext class.

### *Create the Purchase Order Repository*

The purchase order repository will comprise two types:

- **IPurchaseRepository:** This interface represents the contract for purchase order operations without exposing the implementation details.
- **PurchaseOrderRepository:** This class represents the concrete implementation of the purchase order repository contract.

The following code snippet shows how the operations in the purchase order repository contract are declared:

```
using InventoryManagement.Api.Models;

namespace InventoryManagement.Api.Repositories
{
    public interface IPurchaseOrderRepository
    {

      Task<IEnumerable<PurchaseOrder>>
        GetAllAsync(CancellationToken
        cancellationToken = default);

      Task<PurchaseOrder?>
        GetByIdAsync(int id, CancellationToken
        cancellationToken = default);
      Task<PurchaseOrder?>

        GetByOrderNumberAsync(string orderNumber,
        CancellationToken cancellationToken =
        default);

      Task<PurchaseOrder>
        AddAsync(PurchaseOrder entity,
```

**Listing 3:** The InventoryDbContext class

```
public class InventoryDbContext : DbContext
{
    public InventoryDbContext
      (DbContextOptions<InventoryDbContext> options)
        : base(options)
    {

    }

    public DbSet<PurchaseOrder>
      PurchaseOrders => Set<PurchaseOrder>();

    protected override void
      OnModelCreating(ModelBuilder modelBuilder)
    {

        modelBuilder.Entity<PurchaseOrder> (entity =>
        {
            entity.ToTable("purchase_order");
            entity.HasKey(x => x.PurchaseOrderId);

            entity.Property(x => x.PurchaseOrderId)
              .HasColumnName("purchase_order_id");

            entity.Property(x => x.OrderNumber)
              .HasColumnName("order_number").

            HasMaxLength(30).IsRequired();

            entity.Property(x => x.OrderDate)
              .HasColumnName("order_date");

            entity.Property(x => x.OrderStatus)
              .HasColumnName("order_status").

            HasMaxLength(30).IsRequired();

            entity.Property(x => x.Remarks)
              .HasColumnName("remarks").

            HasMaxLength(500);

            entity.Property(x => x.CreatedDate)
              .HasColumnName("created_date");

            entity.Property(x => x.ModifiedDate)
              .HasColumnName("modified_date");

            entity.HasIndex
              (x => x.OrderNumber).IsUnique();
        });
    }
}
```

**Listing 4:** PurchaseOrderRepository class

```
using Microsoft.EntityFrameworkCore;
using InventoryManagement.Api.Data;
using InventoryManagement.Api.Models;

namespace InventoryManagement.Api.Repositories
{
    public class PurchaseOrderRepository :
      IPurchaseOrderRepository
    {
        private readonly InventoryDbContext _context;

         public PurchaseOrderRepository
           (InventoryDbContext context)
        {
            _context = context;
        }

         public async Task<IEnumerable<PurchaseOrder>>
           GetAllAsync(
             CancellationToken cancellationToken = default)
        {
            return await _context.PurchaseOrders
                .AsNoTracking()
                .OrderByDescending(x => x.OrderDate)
                .ThenBy(x => x.OrderNumber)
                .ToListAsync(cancellationToken);
        }

        public async Task<PurchaseOrder?>
          GetByIdAsync(int id, CancellationToken
          cancellationToken = default)
        {
            return await _context.PurchaseOrders
                .AsNoTracking()
                .FirstOrDefaultAsync (x =>
                  x.PurchaseOrderId == id, cancellationToken);
        }

        public async Task<PurchaseOrder?> GetByOrderNumberAsync(
          string orderNumber,
            CancellationToken cancellationToken = default)
        {
            return await _context.PurchaseOrders
                .AsNoTracking()
                .FirstOrDefaultAsync(x => x.OrderNumber ==
              orderNumber, cancellationToken);
        }

        public async Task<PurchaseOrder>
          AddAsync(PurchaseOrder entity,
            CancellationToken cancellationToken = default)
        {
            await _context.PurchaseOrders
              .AddAsync(entity, cancellationToken);

            await _context.SaveChangesAsync
              (cancellationToken);

            return entity;
        }

        public async Task<PurchaseOrder?>
          UpdateAsync(PurchaseOrder entity,
          CancellationToken cancellationToken = default)
        {
            var existingEntity = await _context.PurchaseOrders
                .FirstOrDefaultAsync
                (x => x.PurchaseOrderId ==
                  entity.PurchaseOrderId, cancellationToken);

             if (existingEntity is null)
            {
                return null;
            }

            existingEntity.OrderNumber =
               entity.OrderNumber;
            existingEntity.OrderDate =
               entity.OrderDate;
            existingEntity.OrderStatus =
              entity.OrderStatus;
            existingEntity.Remarks =
              entity.Remarks;
            existingEntity.ModifiedDate =
              entity.ModifiedDate;

            await _context.SaveChangesAsync
              (cancellationToken);

            return existingEntity;
        }

        public async Task<bool> DeleteAsync(int id,
          CancellationToken cancellationToken = default)
        {

            var existingEntity = await _context.PurchaseOrders
                .FirstOrDefaultAsync
                (x => x.PurchaseOrderId == id, cancellationToken);

            if (existingEntity is null)
            {
                return false;
            }

            _context.PurchaseOrders.Remove(existingEntity);
            await _context.SaveChangesAsync (cancellationToken);
            return true;
        }

        public async Task<bool> ExistsAsync(int id,
          CancellationToken cancellationToken = default)
        {
            return await _context.PurchaseOrders
                .AnyAsync(x => x.PurchaseOrderId == id,
                  cancellationToken);
        }
    }
}
```

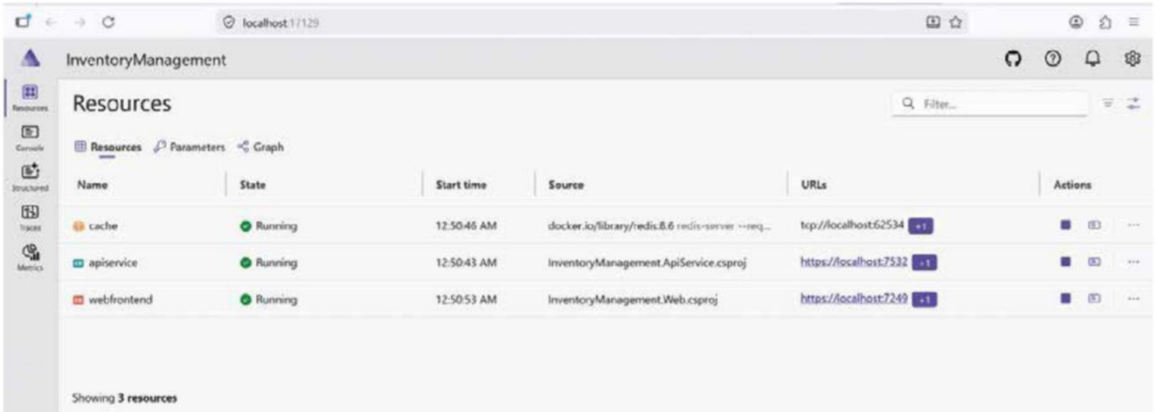



**Figure 5:** The inventory management application running in the browser

**Listing 5:** PurchaseOrderController class

```
using Microsoft.AspNetCore.Mvc;
using InventoryManagement.Api.Models;
using InventoryManagement.Api.Repositories;

namespace InventoryManagement.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]

    public class PurchaseOrderController : ControllerBase
    {
        private readonly IPurchaseOrderRepository _repository;

          public PurchaseOrderController
            (IPurchaseOrderRepository repository)
          {
            _repository = repository;
          }

         [HttpGet]
         public async Task<ActionResult
                  <IEnumerable<PurchaseOrder>>>
                  GetAll(CancellationToken cancellationToken)
         {
            var items = await _repository
             .GetAllAsync(cancellationToken);
            return Ok(items);
         }

         [HttpGet("{id:int}")]
         public async  Task<ActionResult<PurchaseOrder>>
           GetById(int id, CancellationToken cancellationToken)
        {
            var item =  await _repository
              .GetByIdAsync(id, cancellationToken);

            if (item is null)
            {
                return NotFound();
            }

            return Ok(item);
        }

        [HttpGet("by-order-number/{orderNumber}")]
        public async Task<ActionResult<PurchaseOrder>>
          GetByOrderNumber(string orderNumber,
            CancellationToken cancellationToken)
        {

            var item = await _repository.GetByOrderNumberAsync
              (orderNumber, cancellationToken);

            if (item is null)
            {
                return NotFound();
            }

            return Ok(item);
        }

        [HttpPost]
        public async Task<ActionResult<PurchaseOrder>>
          Create([FromBody] PurchaseOrder purchaseOrder,
            CancellationToken cancellationToken)
        {

            if (purchaseOrder is null)
            {
                return BadRequest();
            }

            purchaseOrder.CreatedDate = DateTime.UtcNow;
            purchaseOrder.ModifiedDate = null;

            var createdEntity = await _repository.AddAsync
              (purchaseOrder, cancellationToken);

             return CreatedAtAction(

                nameof(GetById),
                new { id = createdEntity.PurchaseOrderId },
                createdEntity);
        }

        [HttpPut("{id:int}")]
        public async Task<ActionResult <PurchaseOrder>>
          Update(int id, [FromBody] PurchaseOrder purchaseOrder,
          CancellationToken cancellationToken)
        {
            if (purchaseOrder is null || id !=
              purchaseOrder.PurchaseOrderId)
            {
                return BadRequest();
            }


            var exists = await _repository.ExistsAsync(id,
              cancellationToken);

            if (!exists)
            {
                return NotFound();
            }

            purchaseOrder.ModifiedDate = DateTime.UtcNow;

            var updatedEntity = await _repository.UpdateAsync
              (purchaseOrder, cancellationToken);

           if (updatedEntity is null)
            {
                return NotFound();
            }

             return Ok(updatedEntity);
        }

        [HttpDelete("{id:int}")]

        public async Task<IActionResult> Delete(int id,
          CancellationToken cancellationToken)
        {
            var deleted = await _repository.DeleteAsync(id,
              cancellationToken);

            if (!deleted)
            {
                return NotFound();
            }

            return NoContent();
        }
    }
}
```

```
      CancellationToken cancellationToken =
      default);

    Task<PurchaseOrder?>
      UpdateAsync(PurchaseOrder entity,
      CancellationToken
      cancellationToken = default);

    Task<bool> DeleteAsync(int id,
      CancellationToken
       cancellationToken = default);

     Task<bool> ExistsAsync(int id,
       CancellationToken cancellationToken =
       default);
   }
}
```

**Listing 4** shows the complete PurchaseOrderRepository class.

**Listing 6:** The Program.cs file of the API project

```
using InventoryManagement.ApiService.Data;
using InventoryManagement.ApiService.Repositories;
using System.Text.Json.Serialization;

var builder = WebApplication.CreateBuilder(args);

builder.Services
    .AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.ReferenceHandler =
        ReferenceHandler.IgnoreCycles;
        options.JsonSerializerOptions.DefaultIgnoreCondition =
        JsonIgnoreCondition.WhenWritingNull;
    });

builder.AddServiceDefaults();
builder.Services.AddProblemDetails();
builder.Services.AddOpenApi();
builder.Services.AddEndpointsApiExplorer();
builder.AddNpgsqlDbContext
<InventoryDbContext>("inventorydb");

builder.Services.AddScoped <IProductRepository,
  ProductRepository>();
builder.Services.AddScoped<IWarehouseRepository,
  WarehouseRepository>();

builder.Services.AddScoped<IInventoryRepository,
  InventoryRepository>();

builder.Services.AddScoped<IPurchaseOrderRepository,
  PurchaseOrderRepository>();

builder.Services.AddScoped <IPurchaseOrderDetailRepository,
  PurchaseOrderDetailRepository>();

var app = builder.Build();

app.UseExceptionHandler();

if (app.Environment.IsDevelopment())
{
    app.MapOpenApi();
}

app.MapDefaultEndpoints();
app.MapControllers();
app.Run();
```

### *Create the PurchaseOrder Controller*

The PurchaseOrderController in this example is a thin API layer that follows HTTP semantics and delegates control to the purchase order repository. Essentially, the PurchaseOrderController translates REST requests into repository calls and DTO responses, enabling clean, secure, and easy access to purchase order operations via RESTful APIs.

The PurchaseOrderController class is an ASP.NET Core Web API controller that extends the ControllerBase class and exposes RESTful endpoints for performing CRUD operations as shown in **Listing 5**.

**Listing 6** shows the Program.cs file of the API project that contains the necessary code to wire up the services, configure middleware, and sets up controllers, handles JSON cycle errors, and uses dependency injection to enable the API to connect to and work with PostgreSQL, OpenAPI, and the repositories.

### *Execute the Application*

**Figure 5** shows the application running in the web browser.

In another tab in the web browser, you'll observe that the dashboard of the application is launched. **Figure 6** shows what the dashboard of the inventory management application looks like when viewed in the web browser.

When you click the Products button in the left navigation menu, you'll see the product data displayed in the web browser, as shown in **Figure 7**.

## What Is Observability and Why Does It Matter?

Identifying issues in distributed systems, such as those in a microservices-based application, is challenging because a single request can flow through multiple services, and any of those services can fail. Observability provides insight into an application's state by capturing and displaying its internal state through logging metrics and tracing. Distributed tracing provides a time-stamped view of each service a request passes through, identifies errors along the way, and provides other related information.

OpenTelemetry is an open-source distributed tracing framework for collecting, storing, and analyzing telemetry data (metrics, logs, and traces). It provides a high-level API for distributed tracing and metrics that can be used by applications in different environments, including monoliths, microservices, and serverless applications.

You can combine .NET Aspire with OpenTelemetry to allow the automation of distributed tracing (.NET) by capturing traces, metrics, and log entries and then presenting the information through the Aspire Dashboard. Additionally, you can use Azure Application Insights to extend your application's capabilities for monitoring, performance improvement and debugging.

There are three main types of data that are important for understanding the performance of a system: tracing data, metrics, and logs.

### *Traces*

Tracing is a process that records details of all events, including method calls and exceptions raised. It captures data about the flow of a particular request or transaction across multiple components in your system, providing information such as the event type, method calls, and exceptions raised. Tracing data is very useful for understanding how a system works and for finding bottlenecks.

### *Metrics*

Metrics consist of numerical data that provide insights into the performance of an application, such as the number of requests per second or the average response time. For example, you could collect the average number of milliseconds it took to render a web page over time.

Metrics are useful for evaluating performance, capacity planning, and other aspects of an application's health. The data captured represents specific events within defined time boundaries, such as average response times or throughput rates.

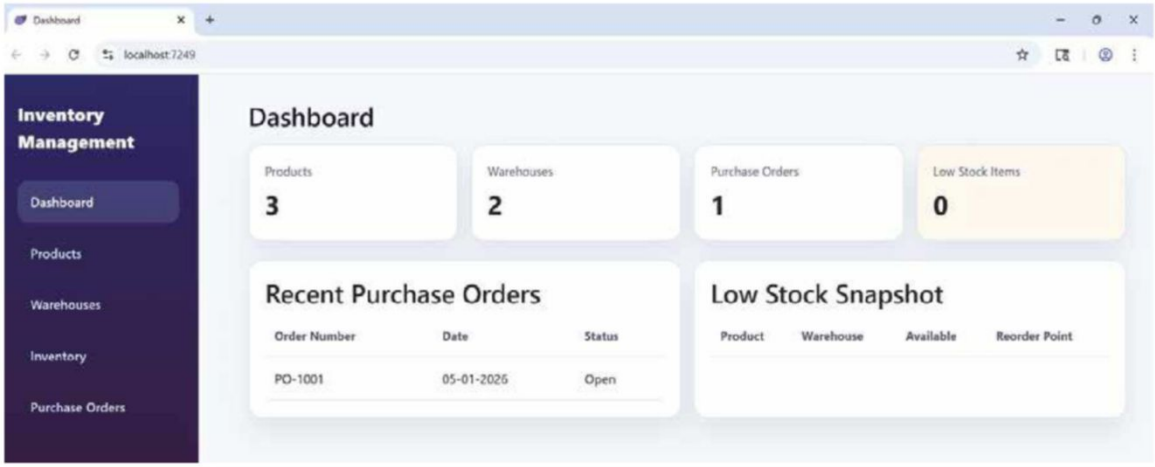



**Figure 6:** The dashboard of the inventory management application

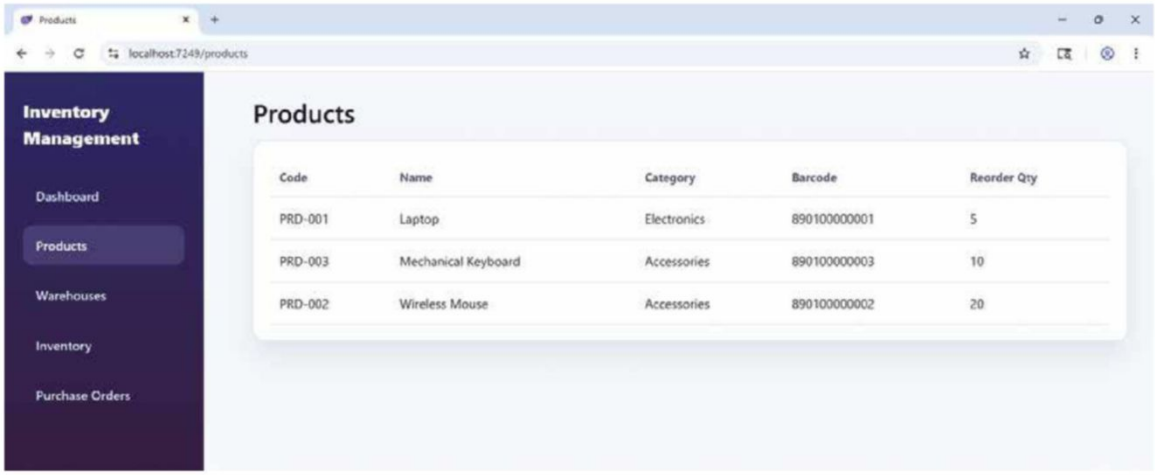



**Figure 7:** Products data displayed in the web browser

### *Logs*

Logs are records of events that occur when an application is running, including information about how long a snippet of code or a method takes to execute, errors, warnings, and so on. Logs are typically used for debugging and troubleshooting problems in an application.

These historical records describe events in an application during a given period (such as user login and authentication). You can use logs to debug problems by reviewing what went wrong when an issue occurs.

### *Configure Observability in .NET Aspire*

Observability improves operational visibility by capturing data and proactively monitoring distributed systems to surface relevant insights based on how an application performs in production over time.

You can read more about observability and open telemetry from this article: https://www.codemag.com/Article/2301061/An-Introduction-to-Distributed-Tracing-with-OpenTelemetry-in-.NET-7.

.NET Aspire provides built-in support for observability. You can enable this support in your AppHost project using the AddOpenTelemetry() method, as shown here:

```
var builder = DistributedApplication.
  CreateBuilder(args);

// Add services
var apiService = builder.AddProject
  <ShoppingCart_ApplicationApiService>
  ("shoppingcart-api");

var webService = builder.AddProject
  <ShoppingCartApplication_Web>
  ("shoppingcart-web")
  WithReference(apiService);

// Enable observability
builder.AddOpenTelemetry();
builder.Build().Run();
```

Once you've configured your AppHost project as shown here, traces and metrics will be sent to the Aspire Dashboard.

### *Integrate Telemetry with Azure Application Insights*

You can integrate with Azure Application Insights as well. The following code snippet shows how you can send traces and metrics to Application Insights.

**Listing 7:** Configuring the Logging Providers

```
public async Task
  GetWebResourceRootReturnsOkStatusCodeWithLogging()
{
  // Arrange

  var builder = await DistributedApplicationTestingBuilder
    .CreateAsync<Projects.AspireApp_AppHost>();

  builder.Services.ConfigureHttpClientDefaults(
    clientBuilder =>
      {
        clientBuilder.AddStandardResilienceHandler();
      });

  builder.Services.AddLogging
   (logging => logging.AddConsole() // Outputs logs to console

     .AddFilter("Default", LogLevel.Information)
     .AddFilter("Microsoft.AspNetCore", LogLevel.Warning)
     AddFilter("Aspire.Hosting.Dcp", LogLevel.Warning));


  await using var app =
    await builder.BuildAsync();

  await app.StartAsync();

   // Act

  var httpClient = app.CreateHttpClient("webfrontend");

  using var cts = new CancellationTokenSource
    (TimeSpan.FromSeconds(30));

  await app.ResourceNotifications
    .WaitForResourceHealthyAsync(
       "webfrontend",
       cts.Token);

  var response =
    await httpClient.GetAsync("/");

   // Assert

  Assert.Equal(
    HttpStatusCode.OK,
    response.StatusCode);
}
```

```
builder.Services.AddOpenTelemetry()
  .WithTracing(tracing =>
     tracing.AddAzureMonitorTraceExporter
     (o => o.ConnectionString =
       "Specify your instrumentation key here;
IngestionEndpoint=
  https://dc.services.visualstudio.com"))

  .WithMetrics(metrics =>
    metrics.AddAzureMonitorMetricExporter
    (o => o.ConnectionString =
      "Specify your instrumentation key here;
IngestionEndpoint=
  https://dc.services.visualstudio.com"));
```

Note that you should install the Azure.Monitor.OpenTelemetry.Exporter NuGet package to run this code.

> When deploying your local database to Azure, a new database will be created in Azure as a replica of your local database. If you want, you can specify a different name for the Azure database.

### *Configure Logging in .NET Aspire*

The following snippet of auto-generated code in the Extensions.cs file of the ServiceDefaults project can be used to configure logging in your .NET Aspire application:

```
builder.AddOpenTelemetry()
 .WithMetrics(metrics =>
 {
   metrics.AddMeter(
     "Microsoft.AspNetCore.Hosting");

   metrics.AddMeter(
     "Microsoft.AspNetCore.Server.Kestrel");

   // Add custom metrics
   metrics.AddMeter(
     "AspireDemo.CustomMetrics");
})

.WithTracing(tracing =>
 {
    tracing.AddSource(
      "AspireDemo.CustomTracing");
 });
```

**Listing 7** shows how you can use the AddLogging method on the builder.Services to configure the logging providers.

Next, create a new API controller named ObservabilityController in a file having the same name with a .cs extension. This controller will be used to aggregate database and telemetry data in a single place. **Listing 8** shows the complete source code of the ObservabilityController class.

**Figure 8** shows product telemetry data displayed in the web browser.

When you click the Traces tab, the traces will be displayed as shown in **Figure 9**.

## Testing .NET Aspire Distributed Applications

Enforcing software quality involves several strategies, including testing. There are several ways in which you can test your application:

- Unit Testing—testing individual parts/units of code
- Integration Testing—testing how parts of your application work together
- Functional Testing—testing independent features in your application
- System Testing—testing that your application works according to both functional and non-functional requirements after the integration testing is complete

**Listing 8:** The ObservabilityController class

```
using InventoryManagement.ApiService.Data;
using InventoryManagement.ApiService.Models;
using InventoryManagement.ApiService.Telemetry;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace InventoryManagement.ApiService.Controllers;

[ApiController]
[Route("api/[controller]")]
public sealed class ObservabilityController : ControllerBase
{
    private readonly InventoryDbContext _context;

    private readonly ProductTelemetryState _telemetryState;

    private readonly ILogger<ObservabilityController> _logger;

     public ObservabilityController(
        InventoryDbContext context,
        ProductTelemetryState telemetryState,
        ILogger<ObservabilityController> logger)
    {

        _context = context;
        _telemetryState = telemetryState;
        _logger = logger;
    }

    [HttpPost("products/page-view")]
    public IActionResult TrackProductsPageView()
    {
        _telemetryState.IncrementProductsPageViews();
        _telemetryState.AddLog("Information",
          "Products page viewed");
        return Accepted();
    }


    [HttpGet("products")]
    public async Task<ActionResult
      <ProductTelemetrySnapshotDto>>
      GetProductsTelemetry(CancellationToken cancellationToken)

    {

        var totalProducts = await
          _context.Products.CountAsync(cancellationToken);
        var totalInventoryQuantity =
        await _context.Inventories.SumAsync
        (x => x.QuantityAvailable, cancellationToken);

        var lowStockProducts = await _context.Inventories
            .CountAsync(x =>
              x.ReorderPoint.HasValue &&
              x.QuantityAvailable <=
              x.ReorderPoint.Value, cancellationToken);

        var dto = new ProductTelemetrySnapshotDto
        {
            Summary = new ProductTelemetrySummaryDto
            {
              TotalProducts = totalProducts,
              LowStockProducts = lowStockProducts,
              TotalInventoryQuantity =
              totalInventoryQuantity,

              ProductsListedCount =
                _telemetryState.ProductsListedCount,
              ProductsRetrievedCount =
                telemetryState.ProductsRetrievedCount,
              ProductsCreatedCount =
                _telemetryState.ProductsCreatedCount,
              ProductsUpdatedCount =
                _telemetryState.ProductsUpdatedCount,
              ProductsDeletedCount =
                _telemetryState.ProductsDeletedCount,
              ProductsPageViews =
                _telemetryState.ProductsPageViews,
                GeneratedAtUtc = DateTime.UtcNow
            },

            Metrics =
            [
               new ProductTelemetryMetricDto
               {
                  Name = "Total Products",
                  Value = totalProducts,
                  Unit = "count",
                  Description = "Current number of products"
               },

               new ProductTelemetryMetricDto
               {
                 Name = "Low Stock Products",
                 Value = lowStockProducts,
                 Unit = "count",
                 Description = "Products at or below
                   reorder point"
               },

               new ProductTelemetryMetricDto
               {
                 Name = "Inventory Quantity",
                 Value = totalInventoryQuantity,
                 Unit = "items",
                 Description = "Total available
                   inventory quantity"
               },

               new ProductTelemetryMetricDto
               {
                 Name = "Products Listed",
                 Value = _telemetryState.ProductsListedCount,
                 Unit = "count",
                 Description = "Total product rows returned
                   by list requests"
               },

               new ProductTelemetryMetricDto
               {
                 Name = "Products Retrieved",
                 Value = _telemetryState.ProductsRetrievedCount,
                 Unit = "count",
                 Description = "Total successful single-product
                   fetches"
               },

               new ProductTelemetryMetricDto
               {
                 Name = "Products Created",
                 Value = _telemetryState.ProductsCreatedCount,
                 Unit = "count",
                 Description = "Total created products"
               },

               new ProductTelemetryMetricDto
               {
                 Name = "Products Updated",
                 Value = _telemetryState.ProductsUpdatedCount,
                 Unit = "count",
                 Description = "Total updated products"
               },

               new ProductTelemetryMetricDto
               {
                 Name = "Products Deleted",
                 Value = _telemetryState.ProductsDeletedCount,
                 Unit = "count",
                 Description = "Total deleted products"
               },

               new ProductTelemetryMetricDto
               {
                 Name = "Products Page Views",
                 Value = _telemetryState.ProductsPageViews,
                 Unit = "count",
                 Description = "Observed UI visits to the
                   products page"
               }
            ],

            RecentLogs = _telemetryState.GetRecentLogs(),
            RecentTraces = _telemetryState.GetRecentTraces()
        };

        _logger.LogInformation ("Generated product
          telemetry snapshot at {GeneratedAtUtc}",
          dto.Summary.GeneratedAtUtc);
        return Ok(dto);
    }
}
```

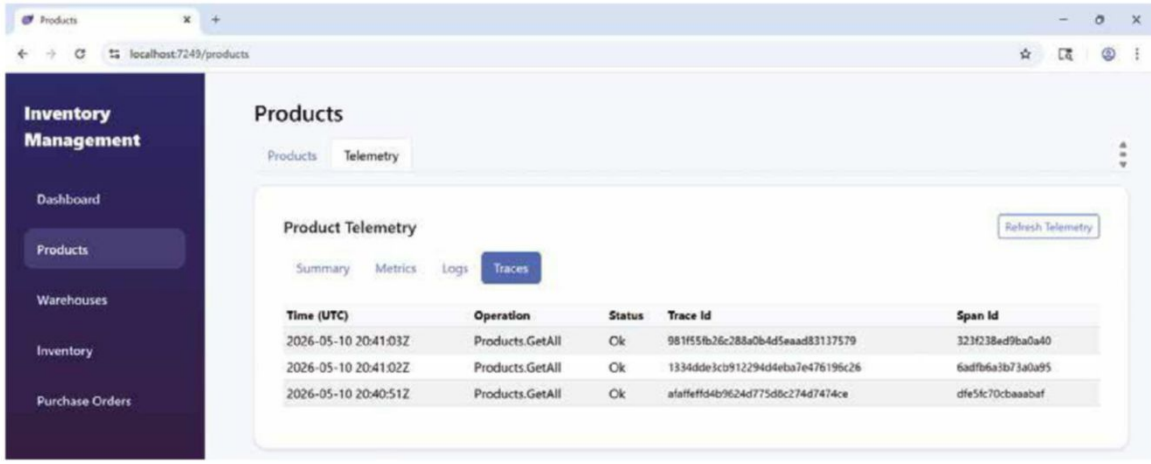



**Figure 8:** Displaying Product Telemetry Data in the web browser

**Listing 9:** ProductRepository Unit Tests: AddAsync, GetAllAsync, and UpdateAsync

```
[Fact]

public async Task AddAsync_Should_Add_Product()
{
  await using var context = CreateContext(nameof(
    AddAsync_Should_Add_Product));

  var repository =
    new ProductRepository( context,
      NullLogger<ProductRepository>.Instance,
      new ProductTelemetryState());

  var product = new Product
    {
      ProductCode = "PRD-100",
      ProductName = "Keyboard",
      ProductCategory = "Electronics",
      CreatedDate = DateTime.UtcNow
    };


  var result = await repository.AddAsync(product);

  Assert.NotNull(result);
  Assert.Equal("PRD-100", result.ProductCode);
  Assert.Single(context.Products);
}

[Fact]
public async Task GetAllAsync_Should_Return_Products_
  Ordered_By_Name()
{

  await using var context = CreateContext(nameof(
  GetAllAsync_Should_Return_Products_
  Ordered_By_Name));

context.Products.AddRange(
  new Product
  {
    ProductCode = "PRD-200",
    ProductName = "Mouse",
    CreatedDate = DateTime.UtcNow
  },

  new Product
  {
    ProductCode = "PRD-201",
    ProductName = "Adapter",
    CreatedDate = DateTime.UtcNow
  });

await context.SaveChangesAsync();

var repository =
  new ProductRepository(
    context,
    NullLogger<ProductRepository>
      .Instance,
    new ProductTelemetryState());


var result =
  (await repository.GetAllAsync())
    .ToList();
  Assert.Equal(2, result.Count);
  Assert.Equal(
  "Adapter",
  result[0].ProductName);
  Assert.Equal(
    "Mouse",
    result[1].ProductName);
}


[Fact]
public async Task
  UpdateAsync_Should_Return_Null_When_
  Product_Does_Not_Exist()
{
  await using var context =
    CreateContext(nameof(
      UpdateAsync_Should_Return_Null_
        When_Product_Does_Not_Exist));

  var repository =
    new ProductRepository(
      context,
      NullLogger<ProductRepository>.Instance,
        new ProductTelemetryState());

  var product = new Product
  {
     ProductId = 999,
     ProductCode = "PRD-999",
     ProductName = "Unknown"
  };

  var result = await repository.UpdateAsync(product);
  Assert.Null(result);
}
```

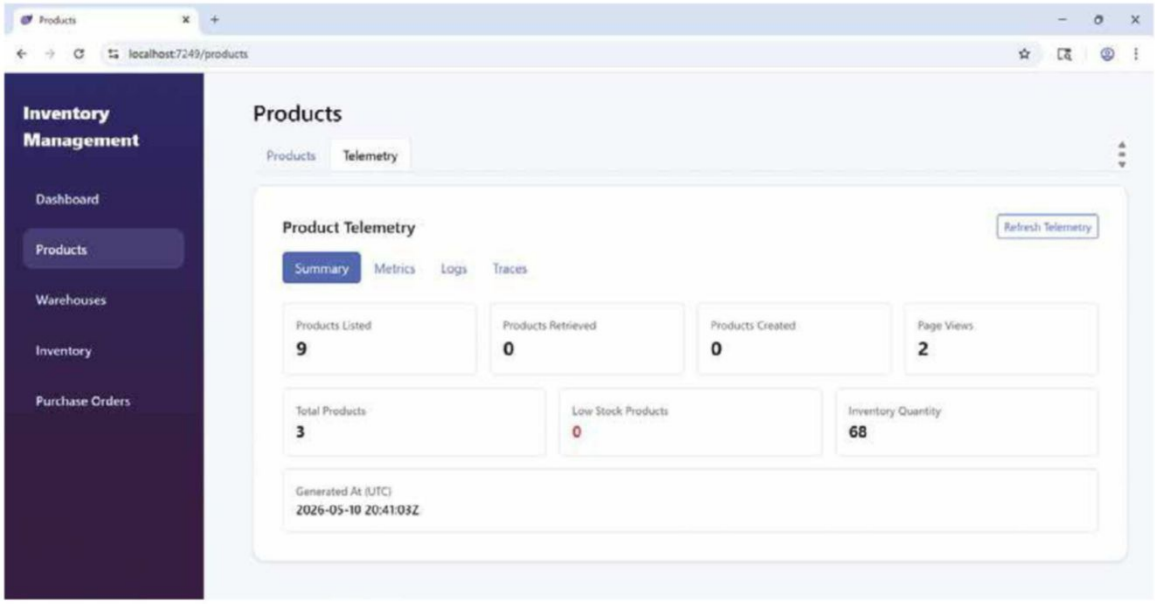



**Figure 9:** Displaying Trace data for Products

- User Acceptance Testing—testing that has been done on the whole app with real users
- Regression Testing—retesting the application to confirm that nothing has broken as a result of recent code changes

Integration tests are used to test an application on a much broader scope than unit tests. While unit tests are typically used to test individual code units in isolation, integration tests are used to test the application together with its components.

In this section, we'll examine how to implement unit and integration tests for our Aspire app. Unit testing focuses on testing each independent component in isolation, often using mocks and stubs for any dependencies. Integration testing, by contrast, examines how multiple components work together and typically uses real dependencies, such as a database or an API.

### *Install NuGet Packages*

To implement unit and integration testing for our application, we'll need the xunit, Moq, and FluentAssertions packages. To install the required packages into your project, right-click on the solution and then select Manage NuGet Packages for Solution….

Once the window pops up, search for the NuGet packages to add to your project. To do this, type in xunit, Moq, and FluentAssertions in the search box and install them one after the other. You can also type the commands shown below at the NuGet Package Manager command prompt:

```
PM> Install-Package Aspire.Hosting.Testing
PM> Install-Package xunit
PM> Install-Package Moq
PM> Install-Package FluentAssertions
```

Alternatively, you can install these packages by executing the following commands at the Windows shell:

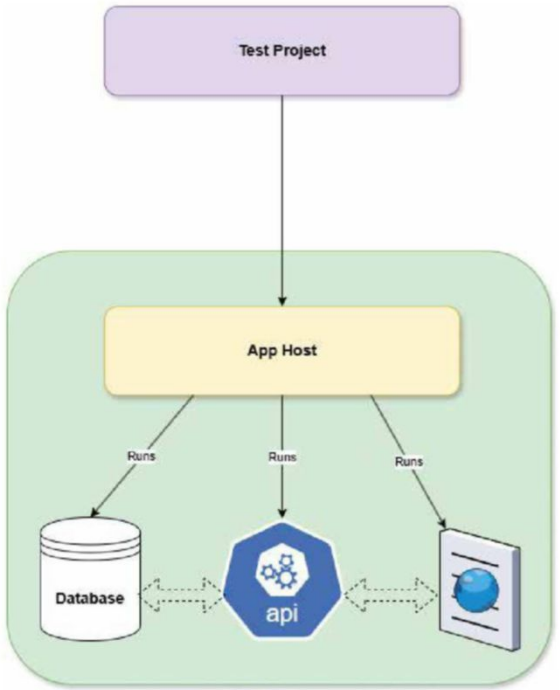



**Figure 10:** Demonstrating the complete flow of testing an Aspire application

```
dotnet add package xunit
dotnet add package Moq
dotnet add package FluentAssertions
dotnet add package Aspire.Hosting.Testing
```

### *Understanding the Flow*

When you test an Aspire application, the Aspire runtime will load the complete solution including the AppHost and all its resources in the memory. **Figure 10** shows how a .NET Aspire test project starts the AppHost which in turn starts the application and its components and resources.

**Listing 10:** IntegrationTests Class with a Test for the GET /api/product Endpoint

```
using InventoryManagement.ApiService.Models;
using Microsoft.Extensions.Logging;
using System.Net.Http.Json;

namespace InventoryManagement.Tests;

public class IntegrationTests
{
  private static readonly
    TimeSpan DefaultTimeout =
      TimeSpan.FromSeconds(30);


[Fact]
public async Task
  GetAll_Should_Return_Success_And_Seeded_Products()
  {
    // Arrange
    var cancellationToken =
      TestContext.Current.CancellationToken;

    var appHost = await
      DistributedApplicationTestingBuilder.CreateAsync
        <Projects.InventoryManagement_AppHost>
          (cancellationToken);

    appHost.Services.AddLogging(logging =>
    {
      logging.SetMinimumLevel(LogLevel.Debug);
      logging.AddFilter(
        appHost.Environment.ApplicationName,
        LogLevel.Debug);
      logging.AddFilter("Aspire.", LogLevel.Debug);
    });

    appHost.Services.ConfigureHttpClientDefaults(
      clientBuilder =>
        {
          clientBuilder.AddStandardResilienceHandler();
        });

    await using var app = await appHost.BuildAsync(
      cancellationToken)
      WaitAsync(DefaultTimeout,
        cancellationToken);

    await app.StartAsync(
      cancellationToken).WaitAsync(
        DefaultTimeout, cancellationToken);


    // Act

    var resourceNotificationService = app.Services
      .GetRequiredService<ResourceNotificationService>();
    await resourceNotificationService
      .WaitForResourceAsync("apiservice",
         KnownResourceStates.Running,
           cancellationToken)
     .WaitAsync(TimeSpan.FromSeconds(60),
           cancellationToken);

    var httpClient = app.CreateHttpClient("apiservice");

    using var response = await httpClient.GetAsync(
      "api/product", cancellationToken);

    response.EnsureSuccessStatusCode();

    var products = await response.Content.ReadFromJsonAsync
      <List<Product>>(cancellationToken);
    Assert.NotNull(products);

    // Assert

    Assert.Equal(HttpStatusCode.OK,
      response.StatusCode);
    }
}
```

Here is how the complete flow works:

1. When you run the test, the test project in turn starts the AppHost.
2. The AppHost orchestrates all dependent app resources.
3. Next, the AppHost runs other components such as the database, API, and any frontend applications.
4. The test project sends an HTTP request to the frontend application (Blazor, React, etc.).

### *Unit Tests*

Since we've selected the testing feature when creating the Aspire application, a test project will be created automatically. In the test project, create a new class named ProductRepositoryTests and add the following code.

```
private static InventoryDbContext
  CreateContext(string dbName)
{
  var options =  new DbContextOptionsBuilder
    <InventoryDbContext>()
      .UseInMemoryDatabase(dbName)
      .Options;

  var context = new InventoryDbContext(options);
    context.Database.EnsureDeleted();
    context.Database.EnsureCreated();
    return context;
}
```

The CreateContext(string dbName) method builds an InventoryDbContext for unit tests using EF Core's in-memory database feature, configured by calling UseInMemoryDatabase(dbName). The method also calls EnsureDeleted, which drops any database left over from a previous test run, and EnsureCreated, which creates a fresh database for the current test.

In **Listing 9** we'll add the following three test methods to the ProductRepositoryTests class.

As you read **Listing 9**, the following is the purpose of each of the unit test methods in brief:

- The AddAsync_Should_Add_Product unit test method creates a new in-memory database context, instantiates the repository with the context, creates an instance of Product and then calls the AddAsync method to add the Product instance to the database. It also verifies if the returned product instance has the right data and the database contains one record.

- The GetAllAsync_Should_Return_Products_Ordered_By_Name unit test method seeds two product records into the in-memory database and calls the GetAllAsync method to get product data and then converts the data to a list. Lastly, it verifies if the data in the list contains both products and is ordered correctly by ProductName.
- The UpdateAsync_Should_Return_Null_When_Product_Does_Not_Exist method works by first creating an empty database—one that contains a product table without any records—and then calling UpdateAsync with a product instance whose ProductId does not exist in the database. Lastly, the method verifies that the returned value is null, since no matching product record exists for the nonexistent ProductId.

### *Integration Tests*

Create a new file named IntegrationTests.cs and replace the auto-generated code with the code in **Listing 10.**

The test method in **Listing 10** uses the Aspire.Hosting.Testing package, which allows for conducting complete end-to-end integration testing of your .NET Aspire distributed application. The integration test method validates whether the web client receives the expected result—a list of products.

Let's now look at how the integration test method works. The following code snippet creates a test instance of your Aspire AppHost project. It uses the DistributedApplicationTestingBuilder to create the full distributed app configuration, such as the API, web frontend, database, and so on.

```
var appHost = await
  DistributedApplicationTestingBuilder
    .CreateAsync
       <Projects.InventoryManagement_AppHost>
         (cancellationToken);
```

The following snippet configures logging for the app and Aspire components.

```
appHost.Services.AddLogging(logging =>
{
   logging.SetMinimumLevel(LogLevel.Debug);
   logging.AddFilter(
     appHost.Environment.ApplicationName,
       LogLevel.Debug);
   logging.AddFilter
     ("Aspire.", LogLevel.Debug);
});
```

Next, the following snippet adds resilience handlers such as retry and circuit breaker to the HttpClient instance.

```
await using var app =
await appHost.BuildAsync
  (cancellationToken).WaitAsync
    (DefaultTimeout, cancellationToken);

await app.StartAsync(cancellationToken)
  .WaitAsync(DefaultTimeout, cancellationToken);
```

Finally, it builds and starts the .NET Aspire distributed application.

## Conclusion

In this article, we built a clean, modular, and testable .NET Aspire application using EF Core and C#. We also examined how to implement observability in the application and discussed how to test a .NET Aspire application using xUnit.

.NET Aspire is not just an orchestration system but a cloud-native control plane for developer experiences. The future will see a tighter feedback loop between services, observability and deployment, allowing developers to rapidly view all their configuration, logs, metrics, and traces in one place without bloating the application.

When combined with service-to-service resiliency, environment-specific configuration, secure secret management, and reproducible infrastructure provisioning, .NET Aspire will provide increased value to teams. Building upon the abilities of Aspire, teams can utilize not only local app startup but create a common toolset to build, observe, and deliver distributed systems in development and deployment environments.

In the future, Aspire will become the foundational layer for establishing an opinionated and consistent platform for building Microsoft .NET applications. This will simplify local development and eliminate many CI/CD, cloud deployment, and operational visibility patterns.

## Takeaways

Here are the key takeaways at a glance:

- .NET Aspire helps you simplify the integration of local development resources into your application, thereby enabling a simple way to discover services across your entire distributed application.
- The App Host is where you can write your code to specify your dependencies and manage connectivity, thereby reducing boilerplate code and ensuring architectural consistency.
- In .NET Aspire, the Service Default project is where you specify centralized configuration for OpenTelemetry, health checks, logging, and service discovery.
- Creating integration points with external systems allows you to write code that is independent of the service you are calling, which simplifies the way you write service-to-service communications.
- Any service that connects to another service can use Aspire's service discovery capabilities, which means you will have all of the benefits of using a resilient, type-safe communication method without worrying about how you connect to the other service.
- With Aspire's support for observability, you can monitor and debug your distributed application using the Aspire Dashboard, your own telemetry endpoints, or through integrated tracing.

Joydip Kanjilal

ONLINE QUICK ID 268061

# Professional Grade AI-Assisted Coding: Context Is Everything with BMAD and Spec Kit

**Bill Catlan**
billcat@distens.com

Bill is presently an AWS Solutions Architect with RapidScale, focused on helping clients leverage cloud infrastructure to achieve competitive advantages. Bill has enjoyed telling computers what to do since he first taught himself to program a Commodore VIC-20 as a young kid. He continues to enjoy exploring how modern platforms and capabilities can add value to businesses and solve some of the world's most vexing challenges. He maintains a current interest and research focus on automating and optimizing AI development and operations. The current state of the art in AI reminds him very much of the early days of the commercial Internet, bringing with it the same kinetic excitement and even greater potential for growth and innovation.

The early days of AI-assisted code generation primarily involved developers sending prompts to an AI coding assistant, such as Anthropic's Claude Code, and taking the generated code and inserting it into a codebase. This technique is generally done in a conversational style and is often referred to as "vibe coding." When vibe coding, the context being managed by the AI model during a single session (i.e., what the model "knows" about your ephemeral set of prompts and decisions at a given time) can become very large in scope and fairly unfocused as the developer continues a session while tackling a series of different tasks. Moreover, because the context reflects an interactive conversation—often with little or no structure—preserving the context is impossible. To help limit this kind of "context sprawl," one easy technique is to clear the context session and begin a fresh session for a new task or feature.

However, when you clear a context session in this way, foundational decisions and other provenance (i.e., instructions and decisions formulated during the context session) are also lost. Moreover, any code that was generated during the cleared session now exists solely on its own, and does not capture any of the specifications, framework choices, or other factors and constraints that resulted in the generated code. This means that someone who wanted to subsequently revise some of the specifications or perhaps change the programming language framework or other aspects would not have the prior work to reference and build upon. Lastly, it also means that the AI is unconstrained if it were to be asked to regenerate similar code or modify existing code. When unconstrained, the AI model can draw from numerous sources to assess its options on how to proceed in answering a prompt, thereby allowing widely varying outcomes. Any foundational decisions or other instructions—artifacts collectively known as "provenance" when properly preserved—which were formulated during the original context session and then cleared, are lost. This is the key deficiency of vibe coding which spec-driven development and context engineering seek to address.

Context engineering moves a developer from a vibe coding technique to a structured approach to generating code that preserves provenance using key artifacts. This allows context to be restored by other developers, reviewed by other stakeholders, or retrieved at any point if someone wanted to audit the inputs that generated a given codebase or simply wanted to continue building upon them. Different methodologies have emerged that offer different perspectives on how to effectively manage context and preserve provenance. BMAD (Breakthrough Method for Agile AI-Driven Development) and GitHub's Spec Kit are methodologies that offer different approaches to engineering context and preserving provenance. Being methodologies, they can be used with multiple AI assistants and models and are not tied to just one. After looking at some of the primitives and mechanics used for context engineering in general, both methodologies, along with their artifacts and philosophies, will be discussed to illustrate how they each allow the developer to engineer context, preserve provenance, and generate cleaner code.

> BMAD and Spec Kit are methodologies that offer different approaches to engineering context and preserving provenance. Being methodologies, they can be used with multiple AI assistants and models and are not tied to just one.

## Context Primitives

Context primitive is almost a misnomer in the age of AI-assisted code generation for how applications get specified and built. Even the lowest level artifacts used for code generation are often quite sophisticated. Nonetheless, they are primitives in the sense that they are the low-level building blocks for your context session, and ultimately, your generated code. Whether the primitive is instructions in a markdown file, an agent, an MCP resource, or something else, it remains the case that the primitives exist to be assembled into a context session and better inform the generated outputs of that session.

Moreover, as sophisticated as they often are, the primitives are typically of little use or meaning until they are assembled, and what is selected into a context session and what is left out is very important for obtaining reliable results.

### *Scoping Context*

When you start to think about scoping context, the early steps are very intuitive. Perhaps you had the experience of your AI tool automatically identifying one or more open documents and automatically including them as part of your prompt's context? Perhaps that is what you wanted at the time and perhaps it wasn't.

The next level of scoping context is to take control over which files and directories you want to be included in the scope of any context session. While this is still a naïve or simple form of context engineering, it nonetheless begins to add a repeatable structure and intention to AI-assisted coding. The following techniques are the early primitives of methodologies like BMAD and Spec Kit. They include the following:

- **Selection.** Intelligently identify only the most relevant files for your session.
- **Compression & Compaction.** Summarize long conversation histories or technical logs to retain key decisions.
- **Ordering.** Place critical rules at the "top of the model's memory" and artifacts pertaining to the immediate task at the end in order to exploit the model's positional bias.
- **Isolation.** Split complex tasks across specialized "sub-agents" (e.g., one for planning, one for testing) for cleaner context.

These techniques are both a good way to get started with shaping the context of your vibe coding sessions and, as we'll see, also remain present if you decide to adopt a more sophisticated methodology, such as BMAD or Spec Kit.

### *Selection Is Primary*

While each of the previous techniques are important, the first one to embrace and master is **selection**. Selection allows you to target important sections of your codebase and provenance files so that the artifacts that get loaded into the current context session are the ones that are most relevant to the current task. The artifacts that are allowed in the context session may be either foundational or reflect any intentional choices that you made for the immediate functional area or component that you are working on. The easiest way to think about this is to simply consider the difference between artifacts that might exist in your global namespace for a given application, as compared to those that would clearly be limited to the UI front-end components or the API back-end components.

Selection allows you to target important sections of your codebase and provenance files so that the artifacts that get loaded into the current context session are the ones that are most relevant to the current task.

A tangible example of this can be shown using the CLAUDE.md file utilized by the Claude Code coding assistant. As a general matter, CLAUDE.md is a low-level primitive artifact that Claude Code will identify and, when placed in the root directory of the project, will load into every context session. CLAUDE.md is Claude Code's primary "memory" file.

Typically, you will find a CLAUDE.md file at the root of the project, perhaps like this:

```
/app/CLAUDE.md
```

However, it will often make sense for different decisions—such as programming language or framework information—to be made specifically for that component. As such, a project whose developers are interested in preserving this kind of intention, and not rely upon the model choosing something for itself, can indicate choices and opinions in a CLAUDE.md file place in a lower-level directory, such as like this:

```
/app/ui/CLAUDE.md
/app/api/CLAUDE.md
```

These are then called "layered" rules, and Claude Code is smart enough to give them precedence. In addition, when setting up a context session, you can explicitly state the area of the code that you want to focus the session on, using direct instructions to the model, such as:

```
$ claude --ignore "!(/app/api/**)"
```

This is a hard boundary that tells **claude** to ignore everything that is not within the /app/api directory. Here we have "slipped" back into a vibe coding-like technique, which is often good for trying things out. But remember, this directive will not be preserved for review or use during future context sessions. For that, you will want to make an entry in the CLAUDE.md file within that directory. A simple example might look like this:

```
## Context Scope
- You are currently working in a specialized sub-module.
- NEVER suggest changes or read files outside of the current
  directory (@./) unless explicitly asked.
- Use `ls -R .` to discover files rather than searching the
  global index.
```

### *The Other Major Techniques*

While selection may be the primary technique for scoping a context, the other techniques mentioned above are also very powerful.

As we will see at length in the sections on both the BMAD and the Spec Kit methodologies, the entire mechanisms for preserving provenance are built upon Compression & Compaction. We will see the specific artifact hierarchy that each methodology employs to capture the results of prompts and other inputs. While these notions are perhaps more intuitively understood as a kind of summariza-

### A Note About Enforcing Locality

You might be tempted to just cd into the folder and run Claude. While Claude Code will see those files, it will still try to find the Git root to understand the full context. Using the --ignore or @./ methods allows Claude to stay aware of the Git context while limiting the session context strictly to your specific subfolder.

tion, the general technical term is Compression & Compaction, primarily because those terms better represent the impact on the session context—namely, providing a succinct and efficient representation of the outputs of prior context sessions for use going forward.

Ordering is a straightforward concept and technique, and useful to keep in mind. While it may seem that earlier inputs would "fade" into the background as new context is developed, the fact of the matter is that prompts and inputs at the "top of the model's memory" for a context session are given a positional bias for appearing first. This is why we want to inject foundational rules and preferences earlier in a context session than the particular requirements of the item at hand.

Lastly, *isolation* is all about addressing different questions using different agents with specialized knowledge or using a specialized agent or leveraging certain external resources that require specific skills by the agent, such as RAG or MCP resources. As such, isolation is perhaps best thought of as a targeted activity type, where selection is all about identifying and targeting specific input files. As the sidebar "A Note About Selection and Isolation" describes, the two techniques are very often used in concert as complementary techniques.

As we will see in the context engineering section below, BMAD harnesses a multi-agent approach, where specialized AI agents are asked to provide solutions to problems in their particular knowledge domain. For example, there is an Architect agent with which it is most appropriate to submit architecture level prompts and develop architecture artifacts. Similarly, there is a specialized Implementation agent, which, you guessed it, is highly tuned to develop artifacts describing implementation guidelines as well as completing the code implementation.

### A Note About Selection and Isolation

It is often the case that selection techniques are used in concert with isolation techniques to limit certain tasks to specific directories, files, and other resources. Being primary, selection is employed to scope just about every process, including when isolating certain activities from other activities. Whether used in concert with each other or used separately, each represents a distinct capability for shaping the context session to allow the AI model to deliver results most in-line with your goals and expectations.

## Context Engineering

As described above, context engineering is the practice of honing or sharpening the baseline information that an AI model will use when suggesting solutions and generating code in response to a prompt. While the ability of models to handle more tokens in their context windows continues to expand every day, it is still important that the context session be focused and well-curated in order to achieve a strong command of what the AI model will do for you, and hence, professional-grade results.

### *Preserving Provenance*

A major component of context engineering is preserving provenance. Again, preserving provenance is the structured approach to capturing the outputs of earlier, higher order, context sessions. In general, provenance means knowing the origin and history of something. With AI-assisted coding, documenting product features and expected outcomes, capturing design decisions, capturing foundational decisions (e.g., coding style guides or frameworks), enforcing architectural patterns, and specifying implementation decisions are all examples of areas where you want to preserve provenance (i.e., preserve outputs for later use by either your own project team or even other projects.) When you preserve your outputs in a well-organized set of artifacts, they can be picked up at any time to be reviewed, reused, tested, or refined by you, someone on your team, or anyone that you would like to invite to contribute to your code.

When you preserve your outputs in a well-organized set of artifacts, they can be picked up at any time to be reviewed, reused, tested, or refined by you, someone on your team, or anyone that you would like to invite to contribute to your code.

Incorporating this strategy into your AI-assisted coding practice is essentially what allows you to move from simple vibe coding (i.e., what in the past might have been referred to as free-form hacking) to a more rigorous process with discipline and structure. As with moving from improvised hacking to software design, moving from vibe coding to developing and preserving provenance using a structured methodology and well-defined set of artifacts has many long-term benefits. Those benefits include:

- **Predictability.** Foundational artifacts capture the core intentions and resulting design decisions you made, rather than leaving the AI model with wide latitude to infer choices, thereby producing expected outputs downstream; the result is cleaner context with many more intentions expressed, leaving less for the AI model to resolve on its own.
- **Reusability.** Other developers can read the higher order artifacts to better understand the framework and structure of your code; they can leverage these artifacts when they want to contribute to your project, or, they can download them and use them like reusable code libraries, providing a full set of design choices and other decisions that they can use in their own new projects.
- **Maintainability.** Because earlier outputs are preserved and always available for injection into a new context session, maintaining a codebase and adding new features is much more efficient and continuity is preserved when the new outputs continue to comply with earlier decisions; because they are well preserved, earlier decisions can also be further interrogated and explored in a later context session, and perhaps revised when it makes sense to do so.
- **Auditability.** Anyone can inspect the higher order artifacts, such as original product requirements and architecture decisions, and discover how and why the AI model is making certain choices during lower order activities like implementation; with vibe coding, those outputs would have been lost with a context session long ago, if they were ever truly developed in the first place.

### *Getting Started with BMAD and Spec Kit*

The best way to get started with AI-assisted coding is, well, to get started! Both of the methodologies described here basically involve the same steps—install a code generation AI assistant CLI, select a model, and install the methodology's specific artifacts.

To get started with BMAD (https://docs.bmad-method.org/), if you are starting from scratch, head over to your favorite web-based AI agent and enter the following prompt to get started:

> How do I get started using the BMAD methodology with Claude Code?

For Spec Kit (https://github.github.com/spec-kit/), enter:

> How do I get started using the Spec Kit methodology with Claude Code?

In both cases, you should be provided with the necessary steps for your platform.

For this article, I am running a JetBrains IDE on my local Windows machine as my IDE. I do not use an integrated plugin for my AI coding assistant, but rather, I use the native CLI version; the native CLI versions of various coding assistants are more broadly available and are often ahead of the IDE plugins that may also be available in terms of features and capabilities. In my own case, I also opt to install the CLI on a remote, private AWS instance running Linux, from which I remotely mount the filesystem to use with my local IDE. See my article in the March/April 2026 issue of CODE Magazine "Remote Debugging Using Mounted Code" (https://www.codemag.com/Article/264061/Remote-Debugging-Using-Mounted-Code) to learn more about that powerful technique. This article uses the Claude Code CLI to drive both methodologies. When set up, a typical coding session looks like what is shown in **Figure 1**.

Note: the repositories containing all the code in this article can be found at https://github.com/billcat-codemag/bmad-pong.git and https://github.com/billcat-codemag/speckit-pong.git.

### *Agents, Skills, and Steps—Oh My!*

When discussing agents with respect to the BMAD methodology, we are referring to the different roles or personas that you can invoke to handle your prompts as part of a particular context session. BMAD is perhaps best understood as multi-agent role simulation. Spec Kit, however, relies more on the human developer to advance phases by issuing slash commands. Where BMAD agents can link workflows and autonomously orchestrate unique agents to analyze similar questions from different perspectives, Spec Kit is more tuned to allowing the human to drive each step, primarily through a simple set of commands.

An agent in this context is just another tool for conveniently shaping or engineering the context session. The artifact that expresses an agent's role and capabilities is typically a set of markdown files containing a persona/role definition, a set of skills and responsibilities, instructions and constraints (how it should behave, what it should refuse), available tasks or commands (workflows it can execute), and references to other resources (checklists, templates, other files it should use). Instruction files and prompt files sitting on a disk are inert. They only become useful when something assembles them into a context window at the right time. An agent is what does that assembly.

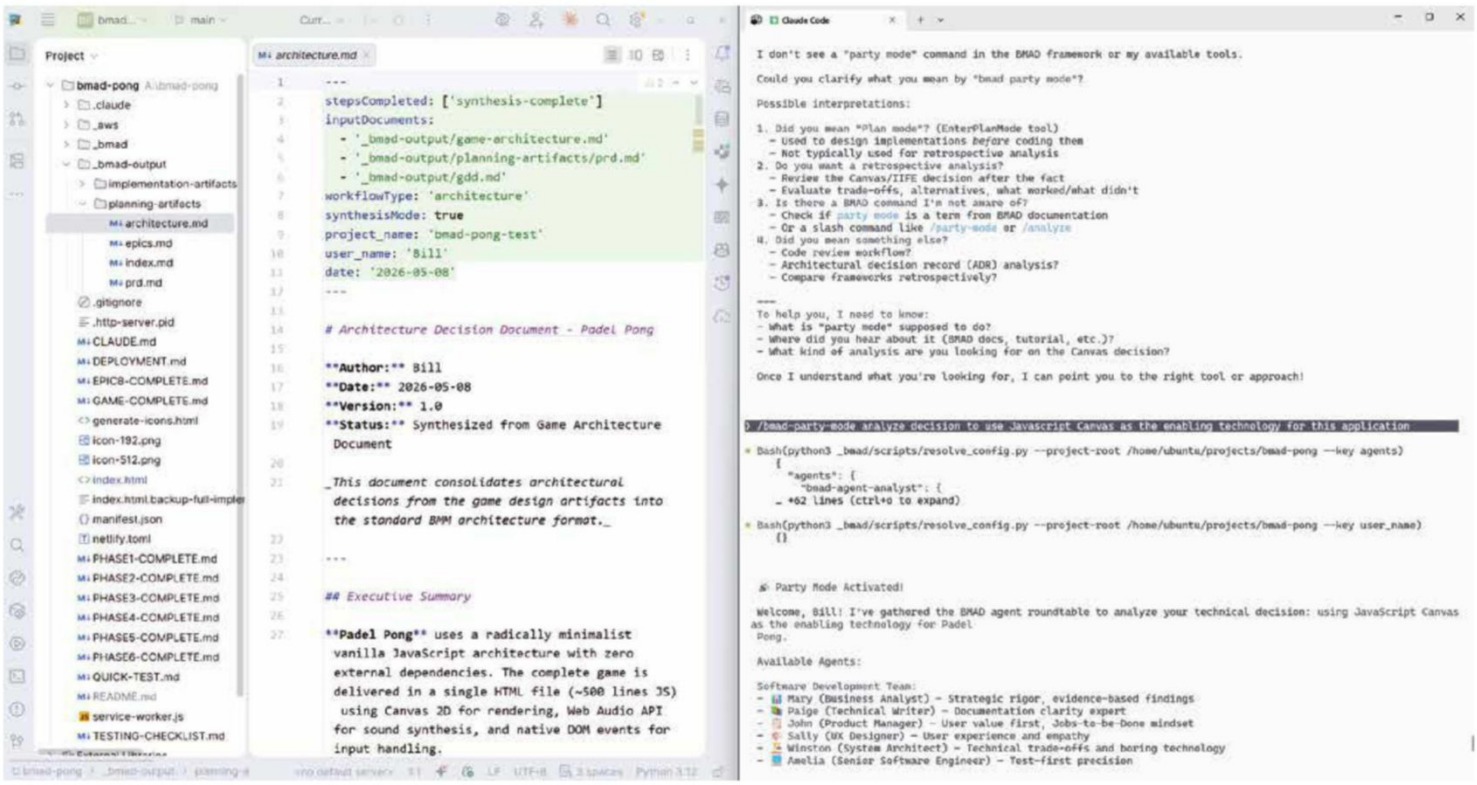



**Figure 1:** A development setup showing the IDE and the Claude Code CLI with BMAD on a remote Linux host

| Function | BMAD Files | Spec Kit Files | Purpose |
|---|---|---|---|
| **Global Rules** | architecture.md | constitution.md | **Same:** Defines tech stack and standards. Coding styles, language preferences, other opinions and other "hard landscape" elements of the project are stated at this level in both methodologies.<br>**Different:** constitution.md is an immutable "law"; BMAD mixes this into architecture and role definitions. |
| **Requirements** | brief.md, prd.md | spec.md | **Same:** Defines what is being built.<br>**Different:** BMAD separates the initial concept and scope (brief.md) from detailed functional requirements (prd.md); Spec Kit captures both in the spec.md file. |
| **Design** | architecture.md | spec.md | **Same:** Provides high order technical design information.<br>Different: BMAD uses a dedicated file for system flow and data schemas; Spec Kit bundles technical design into the spec. |
| **Execution Plan** | epics.md | plan.md | **Same:** Translates requirements into code changes.<br>**Different:** epics.md is an agile construct that consists of a grouping of related features or stories; plan.md is a step-by-step implementation strategy. |
| **Discrete Tasks** | <story_name>.md | tasks.md | **Same:** The smallest unit of work for the AI.<br>**Different:** <story_name>.md includes executable coding instructions for the Developer agent; tasks.md is often a checklist. |
| **Organization** | index.md | (folder based) | **Same:** Provides fine-grained context session management for low order operations.<br>**Different:** BMAD uses index.md to catalog and describe every file in a directory to maintain AI context; Spec Kit relies on a folder structure to organize units of work. |

**Table 1:** Structural hierarchy of the primary file artifacts for BMAD and Spec Kit

When we "activate" an agent, we are just injecting that agent's markdown files into the context session as additional top-of-memory foundational context. Different agents often have access to different tools, giving each one distinct properties or "personas." Another property of the BMAD agents is that one agent can hand off to another by passing context. As we will see below, this is more significant in BMAD, which is a multi-agent methodology, as compared to Spec Kit, which primarily employs only a single agent. (Note, there are some plugins for Spec Kit that seek to introduce a multi-agent element, but it is not the primary objective of the methodology.)

### *BMAD & Spec Kit Outputs*

This section provides a brief overview of the artifacts employed by each of the methodologies. As shown, both methodologies facilitate what is often referred to as spec-driven development, characterized primarily by what we have discussed above in terms of context engineering and preserving provenance. While both frameworks facilitate spec-driven development, they differ in the philosophy, tools, and techniques that they each employ. BMAD, for example, relies heavily on the use of multiple agents with their own personas and specialized skills for developing the spec, whereas Spec Kit employs a single agent and is widely viewed as employing a more streamlined process with a shorter learning curve.

| BMAD Artifact Types | Artifact Type Definitions |
|---|---|
| **Persona Agents** | bmad-agent-*, bmad-cis-agent-*, gds-agent-*<br>(conversational, named, open-ended) |
| **Orchestrator** | bmad-party-mode<br>(multi-agent coordination) |
| **Workflow Skills** | bmad-create-*, bmad-dev-*, gds-create-*, etc.<br>(task-invoked, artifact-producing) |
| **Utility Skills** | bmad-help, bmad-distillator, bmad-shard-doc, etc.<br>(cross-cutting, meta-level) |
| **Module Builders** | bmad-agent-builder, bmad-module-builder, bmad-workflow-builder (meta: builds other skills/agents) |

**Table 2:** BMAD agents and other artifact types found in .claude/skills directory

**Table 1** shows the different artifacts that each methodology generates and relies upon for managing the context session. Items are shown from higher order functions to lower order functions. As you can see, each employs a set of markdown files that fills a set of largely similar functions. In both methodologies, higher order functions facilitated by markdown files that the agent (or agents—in the case of BMAD) load into the context session in a particular order, exploiting the Ordering primitive, and obtain the positional bias that artifacts loaded earlier have by virtue of being loaded in the top of the model's memory. As such, the files follow a strict cascading hierarchy where high-level documents inform and constrain low-level ones.

Some major elements of the context session not represented by these artifacts are the activated agent as well as other prior code that might be selected into a given context session, when implementing stories (BMAD) or tasks (Spec Kit). In addition, for the AI coding assistant that we will be using to exercise each methodology below, namely, Claude Code from Anthropic, the ever-present CLAUDE.md file is still present and important for both bootstrapping the methodology we seek to employ as well as providing for other fundamental elements pertaining to the Claude models.

## A Quick Tour of the BMAD Methodology

In this section, we will take a closer look at the different types of artifacts that inform the context in the BMAD methodology and, in turn, guide the developer in generating high-quality AI-assisted code. BMAD provides a rich hierarchy of agents, workflows, and builders for your project. **Table 2** provides a summary view of this hierarchy.

### *BMAD Knows How to Party*

As mentioned above, BMAD is mostly known for the many personas (i.e., agents) that it can bring to a task. BMAD can orchestrate multiple agents using a command called "Party Mode" (i.e., /bmad-party-mode). Party mode is an orchestration command that can spawn multiple agents to analyze a piece of work, each providing unique insights and often highlighting the tension between the different perspectives and the different choices that you can make. See **Listing 1** to see how the actual BMAD frontmatter and definition describe Party Mode and its purpose.

Below, we will see what it looks like when I asked BMAD to throw a party around which game framework was best for the custom "Padel Pong" game that I created for this article. But first, let's look a little closer at what makes a BMAD agent an agent.

### *What Is a BMAD Agent?*

There is a lot of talk about AI agents everywhere these days, but what is an agent really? Above, I provided one definition of what an agent is, but with BMAD, we can see clearly the tangible artifacts that make up a BMAD agent and its persona. If you look at the git repository for this project, you will see all the unique agents / personas that BMAD offers by looking for files named bmad-agent-* under the .claude/skills directory.

```
$ cd ~/projects/bmad-pong/.claude/skills
$ ls -1 | grep bmad-agent

  bmad-agent-analyst
  bmad-agent-architect
  bmad-agent-builder
  bmad-agent-dev
  bmad-agent-pm
  bmad-agent-tech-writer
  bmad-agent-ux-designer
```

Note that the BMAD methodology installs a full suite of agents and workflows for game development. In fact, to create the "Padel Pong" demo app shown below, I employed the /gds-create-gdd workflow skill to leverage a workflow that factors every aspect of game development. Even with a simple game like "Padel Pong," I spent roughly 8 hours completing the 14-step workflow—with several sub-workflows—to share my intentions and vision for my brand new version of the classic Pong video game that I called Padel Pong. Later, I asked BMAD to convert the outputs into the typical BMAD artifacts described above to be sure to preserve provenance in the typical BMAD way.

A quick look at an agent's frontmatter tells you a few things about agents right away. First, they are typically given formal names, and those names will be used when the output is generated by that agent. Here is what the frontmatter looks like for the BMAD Architect agent in its SKILL.md file:

```
---
name: bmad-agent-architect
description: System architect and technical design
leader. Use when the user asks to talk to Winston
or requests the architect.
---
```

**Listing 1:** Party mode description and purpose from .claude/skills/bmad-party-mode/SKILL.md

```
---
name: bmad-party-mode
description: 'Orchestrates group discussions between installed BMAD
agents, enabling natural multi-agent conversations where each agent
is a real subagent with independent thinking. Use when user
requests Party Mode, wants multiple agent perspectives, group
discussion, roundtable, or multi-agent conversation about their
project.'
---

# Party Mode

Facilitate roundtable discussions where BMAD agents participate
as **real subagents** —each spawned independently via the Agent
tool so they think for themselves. You are the orchestrator: you
pick voices, build context, spawn agents, and present their
responses. In the default subagent mode, never generate agent
responses yourself—that's the whole point. In `--solo` mode,
you roleplay all agents directly.

## Why This Matters

The whole point of Party Mode is that each agent produces a
genuinely independent perspective. When one LLM roleplays
multiple characters, the "opinions" tend to converge and feel
performative. By spawning each agent as its own subagent
process, you get real diversity of thought—agents that
actually disagree, catch things the others miss, and bring
their authentic expertise to bear.
```

From there, there are usually a predictable set of sections that define the agent's unique capabilities and constraints. For the bmad-architect-agent, those sections include the following:

```
# Winston — System Architect
## Overview
## Conventions
## On Activation
### Step 1: Resolve the Agent Block
### Step 2: Execute Prepend Steps
### Step 3: Adopt Persona
### Step 4: Load Persistent Facts
### Step 5: Load Config
### Step 6: Greet the User
### Step 7: Execute Append Steps
### Step 8: Dispatch or Present the Menu
```

The SKILL.md file uses placeholder values such as {agent.identity} and {user_name} so that they can be dynamic. Those values can be found in the customize.toml file that accompanies each SKILL.md file.

### *Introducing Padel Pong*

As described above, I spent several hours to leverage the BMAD agents and workflows tailored to game development. After completing the lengthy specification process with expert game development guidance, I had a brand-new version of Padel Pong that I could play immediately! **Figure 2** shows what the key planning artifacts look like in the IDE. The implementation-artifacts folder contains all the stories needed to implement the build. There are too many to show them all here, but I encourage you to download the project and look at them for yourself.

I then ran Party Mode (/bmad-party-mode) against a selected epic—namely, the epic where I had chosen the JavaScript framework and architecture. **Listing 2** shows some of the resulting output from the "always-ready-to-party" BMAD agents.

#### A Note on Manual Edits in the BMAD and Spec Kit Frameworks

Manual edits should primarily occur only in the top two levels of either hierarchy (Architecture /Constitution or PRD/Spec). The AI then uses these as the hard landscape for regenerating lower-level plans and tasks when requirements change. Techniques can be applied to protect manual changes from being updated by an AI agent.

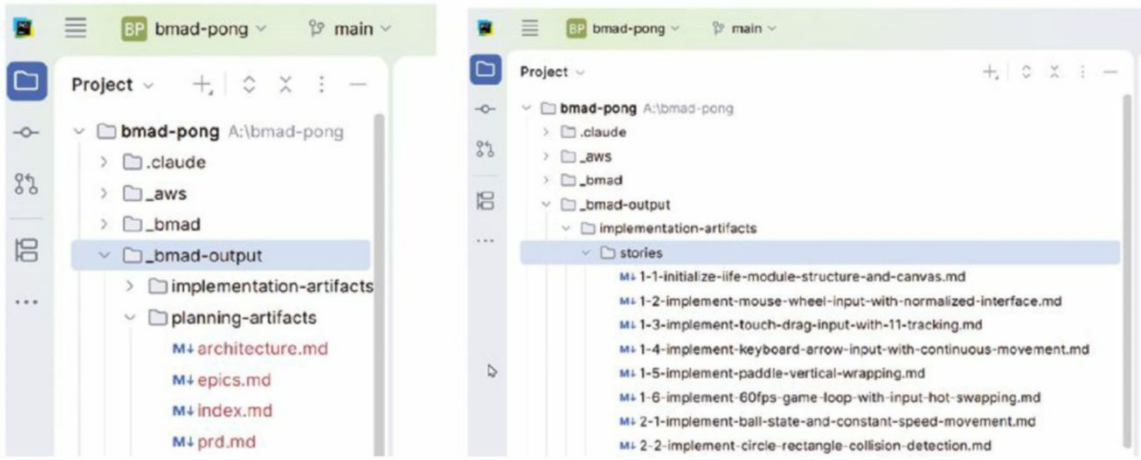



**Figure 2:** BMAD planning and implementation artifacts.


**Figure 3:** The Padel Pong start screen and some early gameplay.

Many elements that were my unique ideas for a "modern pong" have been captured. For example, since the consensus was pretty clear, I decided to go with a simple Canvas-based implementation. I also decided that the mouse wheel would be the primary controller on desktop browsers but was able to include support for touch for mobile devices as well. I also specified the sound effects, colors, and effects that happen during gameplay.

Many other decisions are reflected as well—see if you can tell how Padel Pong is different from classic Pong! **Figure 3** shows the Padel Pong start screen and some early gameplay. As exciting as that is, the most exciting part for me is that by following a disciplined approach, managing context, and preserving provenance in a structured way, I am ready to go back into any part of the game to further refine it or add new features, without losing all of the valuable information that was curated during this build process.

## A Quick Tour of the Spec Kit Methodology

Where BMAD seeks to bring domain experts (i.e., agents with personas and specialized skills) with multiple perspectives and comprehensive workflows to the party, Spec Kit brings a refined process that primarily relies on the human developer to drive the process. The idea here is that you have a human product development and software engineering team for assessing the market and developing requirements, but the developer is looking to use a methodology to reduce human input and feature requests to a specification that either a human or a coding agent can implement. Again, the key benefit being sought here is managing context, preserving provenance, and creating artifacts that can be reused and refined by anyone in the future.

### *The Spec Kit Development Loop*

The Spec Kit methodology primarily revolves around the idea that you already know what you want to do, or you are picking up a brownfield project, and you would like Spec Kit to infer what has been done in the past so that you can iterate on it. There are five phases in total, but as we will see, most of the work involves looping through the middle three phases. The five phases are:

1. **Global Rules.** (/speckit-constitution) Define or update your project's guiding principles and development guidelines.
2. **Requirements.** (/speckit-specify) Outline what you want to build, including requirements and user stories.
3. **Execution Plan.** (/speckit-plan) Create a detailed technical implementation plan using your chosen tech stack.
4. **Discrete Tasks.** (/speckit-tasks) Generate actionable and structured task lists for development.
5. **Implement.** (/speckit-implement) Execute all defined tasks to build features according to your plan.

**Listing 2:** BMAD Party Mode analyzing choice of JavaScript Canvas

```
❯ /bmad-party-mode analyze decision to use JavaScript Canvas as the enabling
technology for this application

• 🎉 Party Mode Activated!

  Welcome, Bill! I've gathered the BMAD agent roundtable to analyze
  Your technical decision: using JavaScript Canvas as the enabling
  technology for Padel Pong.

  Available Agents:

  Software Development Team:
  - 📊 Mary (Business Analyst) — Strategic rigor, evidence-based
      findings
  - 📚 Paige (Technical Writer) — Documentation clarity expert
  - 📋 John (Product Manager) — User value first,
      jobs-to-be-done mindset
  - 🎨 Sally (UX Designer) — User experience and empathy
  - 🏗️ Winston (System Architect) — Technical trade-offs and
      boring technology
  - 💻 Amelia (Senior Software Engineer) — Test-first precision
  - 🧪 Murat (Test Architect) — Risk-based quality strategy

  Game Development Team:
  - 🏛️ Cloud Dragonborn (Game Architect) — Engine architecture,
      60fps discipline
  - 🎲 Samus Shepard (Game Designer) — Player feel and game
      mechanics
  - 🎮 Indie (Solo Dev) — Ship-it mindset, prototype fast
  - 🕹️ Link Freeman (Game Developer) — Hands-on implementation

  Creative Team:
  - 📖 Sophia (Storyteller) — Mythic narrative structure
  - 🎨 Maya (Design Thinking) — Human-centered empathy
  - 🧠 Carson (Brainstorming) — Wild idea generation
  - 🔬 Dr. Quinn (Problem Solver) — Root cause analysis
  - ⚡ Victor (Innovation Strategist) — Disruption theory
  - 🎬 Caravaggio (Presentation Expert) — Visual communication

  ---
  For this technical architecture discussion, I'm bringing in
  the voices most qualified to analyze the Canvas decision:

• 4 agents finished (ctrl+o to expand)
   ├ Winston analyzes Canvas choice · 0 tool uses
   │  └ Done
   ├ Cloud Dragonborn analyzes Canvas choice · 0 tool uses
   │  └ Done
   ├ Indie analyzes Canvas choice · 0 tool uses
   │  └ Done
   └ Amelia analyzes Canvas choice · 0 tool uses
      └ Done

  ---
  🏗️ Winston:

  Let me lay out what I see here, starting with the constraints and
  working backward to the decision.

  The core trade-off was this: Canvas gave you a single-file,
  zero dependency artifact that ships at 35KB and runs at
  60fps everywhere. That's not exciting technology — it's boring
  technology that's been stable since 2010. And that's exactly
  why it was the right call.

  What you avoided: ...

  What you gained: ...

  The counter-argument I'd watch for: ...

  The business lens: ...

  My verdict: This was a textbook example of choosing boring
  technology. You matched the tool to the problem instead of
  over-engineering for hypothetical futures. If I were doing
  post-mortem, I'd clone this decision-making pattern for
  the next project.

  One question: ...

  ---
  🏛️ Cloud Dragonborn:

  Ah, traveler... you seek wisdom about the foundation stones
  upon which this temple was built. Let me examine the weight-
  bearing walls of this choice.
  pauses, gazing at the architecture
  The Canvas decision reveals itself as structurally sound
  for this specific keep. Here is what the bones tell me:

  Where the foundation holds strong: ...

  Where cracks might show at scale: ...

  The deeper truth: ...

  This wasn't a technology decision — it was a scope-bounded
  Constraint satisfaction. Canvas was the right hammer because
  you correctly identified the nail: a performance-critical,
  input-diverse, minimalist game that ships fast and runs
  everywhere.

  The wisdom here isn't "Canvas is always right" — it's that
  you matched the abstraction level to the problem domain. No
  WebGL overkill, no framework ceremony, no build-tool
  yak-shaving. You built a game, not a game engine.

  If tomorrow you add multiplayer, leaderboards, achievements,
  replays...the Canvas foundation will start groaning. But today?
  The temple stands, the framerate holds, and the pilgrims can
  play offline on their phones.

  That is architectural pragmatism. You built for the mission you
  had, not the one you imagined.

  ---
  🎮 Indie:

  Canvas was the right call. Here's why it shipped:

  What worked: ...

  The reality check: ...

  The real win:
  Bill shipped a production-ready game in one file with zero
  deps. That's not just a tech choice, that's a philosophy —
  keep it simple, keep it shippable, prove the gameplay works
  before you gold-plate the renderer.

  Canvas + vanilla JS was the indie move. Scope-matched,
  Performance proven, and it got across the finish line. Would
  I make the same call for a bigger game? Hell no. But for Pong?
  Textbook.

  ---
  💻 Amelia:

  Analyzing Canvas decision requires baseline data. Need to see: ...

  Without the implementation file I'm working blind. What's
  the absolute path to the game HTML?

  Canvas was probably right for zero-dependency 60fps
  requirement, but "probably" isn't good enough. Show me
  /home/ubuntu/projects/bmad-pong/[filename].html and I'll
  give you hard numbers on whether this tech choice
  delivered or left performance/maintainability debt.

  ---
  Orchestrator Note:

  Strong consensus from Winston, Cloud, and Indie that Canvas
  was the right call for this scope. Amelia wants to audit the
  actual implementation (/home/ubuntu/projects/bmad-pong/index.html)
  before giving her verdict. All four flag the same boundary
  condition: this works for Pong, but would show strain at larger
  scale.
```

As mentioned, when working with Spec Kit, you will spend most of your time looping through requirements, execution plan, and discrete tasks. However, you can use /speckit-constitution at any time to revise your global rules. Likewise, you can move to the Implement step—/speckit-implement—at any time to generate an implementation that includes the latest feature.

When working with Spec Kit, you will see that the experience is far less chatty than working with a BMAD agent. With BMAD, you are signing up for a guided tour with a very knowledgeable tour guide. As such, you are inviting in-depth questioning and feedback from one or more experts in the areas of market research, technical frameworks, user experience, and more. With Spec Kit, you are simply asking for a methodology that helps with certain bookkeeping and other techniques around preserving provenance. There is no off-the-shelf team of chatty agents ready and eager to spend hours walking you through workflows covering every stage of software development. However, you do get well-structured outputs, a clean methodology, and the ability to bring your own thinking to either a brownfield or greenfield project very quickly.

### *Padel Pong Unleashed*

To showcase Spec Kit's strength in adding features to an existing codebase, I decided to take the resulting code from the BMAD methodology and add it—and it alone—to a new project. The goal here is to show how it is possible to take a brownfield project and get started on it using a structured AI-assisted methodology just as quickly as if you were simply vibe coding.

Of course, you are leaving a lot up to Claude Code to infer from the code itself as to why certain things exist the way that they do, and you don't immediately obtain the surgical control over the generated code that you can achieve when carefully developing the specs. However, it does demonstrate that you can get started quickly and you can begin generating artifacts that capture your decisions and preserve provenance for future sessions. Further, as the old saying goes, you can't edit a blank page. By getting started quickly in this way, you begin generating the provenance artifacts that reflect the current state and can be edited moving forward, whether to add features or just better document the current code and the product intent.

As mentioned above, Spec Kit is lightweight and excels at providing powerful developer assistance in methodology and tracking. For example, after inferring constitution.md using the /speckit-constitution command against just the index.html code file, and perhaps further refining it with some additional prompts, you are now ready to use the /speckit-specs command to begin a new feature. Doing so can look like this:

```
> /speckit-specify simple leaderboard that shows at
  the end of the match
```

As shown in **Figure 4**, that simple command will result in Spec Kit generating a new feature branch in git and proceeding to fill in all the artifacts that are needed to implement the feature! Of course, it is unlikely that the new leaderboard with show everything we want in just the way we want it, but we have just set up the full scaffolding needed to take that feature wherever we want! Moreover, your results will be preserved in a well-organized set of artifacts that can be picked up at any time to be reviewed, reused, tested, or refined by you, someone on your team, or anyone that you would like to invite to contribute to your code.

As shown in **Figure 5**, the original end screen (left) does not call out the winner and the loser. However, the new version (right), which implements the new "001-match-end-leaderboard" feature, overlays an initial version of our leaderboard showing the two players and even allowing some user feedback on how the gameplay was.

## Use AWS Bedrock for Private AI-Assisted Coding

When you are engineering context and preserving provenance, depending on the nature of the work that you are doing, there is a good chance that you are also developing high-value intellectual property while doing that work. Whether you should allow your session inputs and outputs to get shared back to the model that you are working with is an important question. Many organizations have a compelling interest in treating the work product that their product teams and developers create during AI-assisted development as proprietary and/or confidential; they want to ensure that their newly developed work is not being fed back into a public large language model.

### *The Model Deployment Account and Other AI Data Protections*

Services such as AWS Bedrock offer a service where they make the foundation large language model available to your developers—something that would take millions of dollars in processing power alone, and a lot of time and know-how, for your organization to curate on its own—while also offering guarantees that your session inputs and outputs belong to you alone, and will not be ingested back into the large language model or otherwise used to inform responses to prompts by other users that use the model.

As stated on the AWS website, "Amazon Bedrock has a concept of a Model Deployment Account—in each AWS Region where Amazon Bedrock is available, there is one such deployment account per model provider. These accounts

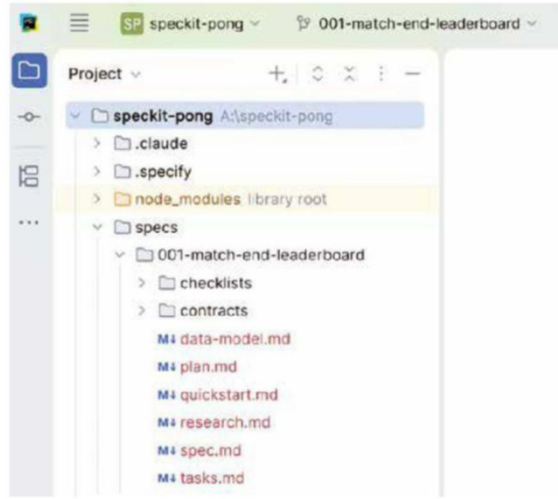



**Figure 4:** A new feature directory and branch "001-match-end-leaderboard" created by /speckit-specify.

### Dive Deeper Into Data Protection Policies

The most authoritative answers around the full scope and attestations of data protection can be found in Amazon Bedrock FAQs (aws.amazon.com/bedrock/faqs/) and their data protection docs (docs.aws.amazon.com/bedrock/latest/userguide/data-protection.html). For Anthropic's published statements on such topics, see **Settings > Privacy > How we protect your data** and **Settings > Privacy > How we use your data** (https://claude.ai/settings/data-privacy-controls) in the Claude console. Also see, "Is my data used for model training? (Consumer)" (https://privacy.claude.com/en/articles/10023580-is-my-data-used-for-model-training) and "Is my data used for model training? (Commercial)" (https://privacy.claude.com/en/articles/7996868-is-my-data-used-for-model-training) to compare use cases.


**Figure 5:** The original end screen (left) and the new end screen (right) showing the first version of a leaderboard.

are owned and operated by the Amazon Bedrock service team. Model providers don't have any access to those accounts. After delivery of a model from a model provider to AWS, Amazon Bedrock will perform a deep copy of a model provider's inference and training software into those accounts for deployment. Because the model providers don't have access to those accounts, they don't have access to Amazon Bedrock logs or to customer prompts and completions." Aside from keeping your inputs and outputs safe from being ingested into the model, AWS and Anthropic also offer certain copyright indemnifications that are also worth taking a quick look at.

> Because the model providers don't have access to those accounts, they don't have access to Amazon Bedrock logs or to customer prompts and completions.

As indicated in the sidebar "Dive Deeper Into Data Protection Policies," Anthropic makes many similar assertions about not using your session data to train the model that AWS does. However, the AWS framework is more clearly documented and has more structure that can be administered by your enterprise administrators, eliminating the chance for individual users to make mistakes when making selections for how data should be shared. For example, while Anthropic's policy asserts that session data sharing is not enabled by default, and a user would need to opt in, I was surprised when I found the "Help improve Claude" toggle enabled in my account, thereby opting me in to sharing my conversation and coding sessions back to the public model. I trust that I did indeed opt in at some point, but I simply don't recall doing so. See **Figure 6** for what the sharing session data toggle looks like for Claude Code.

Moreover, Anthropic has wide-ranging concepts of enterprise (commercial) customers and consumers for their multiple services. Each classification attaches separate policies and protections. I personally found it confusing and somewhat surprising that the use of Claude Code falls exclusively under the policies and protections for consumer customers.

### *An Air-gapped AI Development and Runtime Environment*

For a long time now, AWS has offered support for completely private access to its resources and services. AWS began with supporting private networking pipes from external datacenters into AWS accounts in order to connect hybrid deployments and continues to this day to support the private network use case with current offerings of its PrivateLink endpoints in the Virtual Private Cloud (VPC).

PrivateLink allows compute and other AWS services to establish service endpoints within customer VPCs that are reachable via private IP addresses in any of the RFC 1918 defined ranges and which are addressable over private routes—without any routing to the public internet required. Being a PrivateLink enabled service, in addition to the standard public AWS API, the AWS Bedrock service offers PrivateLink endpoints for accessing its full array of AI models sourced from multiple providers over routes that are completely isolated from the public internet. Implementing a private networking strategy like the one shown in **Figure 7** has the advantage of both reducing your attack surface and benefiting from the performance benefits of traffic remaining on the AWS networking rails.

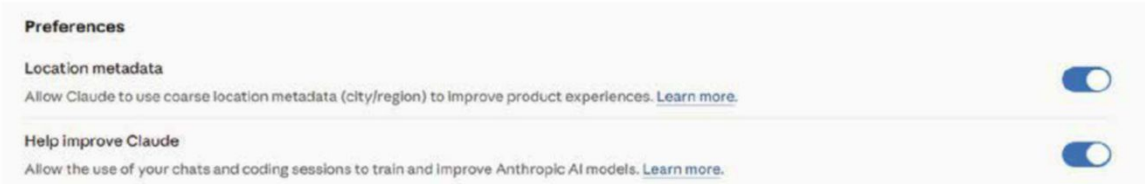



**Figure 6:** Anthropic setting for sharing session data back to the model is shown here as enabled. That may not be what you want.

### Allowing Egress to the Public Internet

It is often the case that a completely air-gapped solution is not desirable, and your requirements will dictate that your development environment needs access to various resources over the public internet. In such cases, the typical enterprise solution would be to provide a route to an AWS Managed NAT Gateway, which only allows outbound connections to the internet.

However, for individuals and small teams, you could also opt to host your own NAT gateway instance on the same EC2 instance that you are using as a bastion server. Or, you could even avoid a separate instance altogether by giving your AI development instance a public IP address and route to the internet for egress only, while blocking all ingress traffic using AWS security groups. While it might not check all of the boxes for an environment subject to strict compliance audits—and it can be more prone to configuration mistakes or certain malfeasance—such a configuration can provide a secure and fairly low-cost internet egress solution for individuals and small teams. You could even use this outbound internet capability to establish an outbound VPN link to your local router to allow easy access over a VPN tunnel from your local machine.

**Listing 3:** Foundation models available from AWS Bedrock.

```
$ aws bedrock list-foundation-models --region us-east-1 --no-paginate | grep modelId | sort

…
    "modelId": "anthropic.claude-3-5-haiku-20241022-v1:0",
    "modelId": "anthropic.claude-3-haiku-20240307-v1:0",
    "modelId": "anthropic.claude-3-haiku-20240307-v1:0:200k",
    "modelId": "anthropic.claude-3-haiku-20240307-v1:0:48k",
    "modelId": "anthropic.claude-3-sonnet-20240229-v1:0",
    "modelId": "anthropic.claude-3-sonnet-20240229-v1:0:200k",
    "modelId": "anthropic.claude-3-sonnet-20240229-v1:0:28k",
    "modelId": "anthropic.claude-haiku-4-5-20251001-v1:0",
    "modelId": "anthropic.claude-opus-4-1-20250805-v1:0",
    "modelId": "anthropic.claude-opus-4-20250514-v1:0",
    "modelId": "anthropic.claude-opus-4-5-20251101-v1:0",
    "modelId": "anthropic.claude-opus-4-6-v1",
    "modelId": "anthropic.claude-opus-4-7",
    "modelId": "anthropic.claude-sonnet-4-20250514-v1:0",
    "modelId": "anthropic.claude-sonnet-4-5-20250929-v1:0",
    "modelId": "anthropic.claude-sonnet-4-6",
…

Editor's note: This listing was too long to print in full for the value
it provides to readers. See the article online for this long list of
available models.
```

**Listing 4:** AWS IAM role permissions for accessing AWS Bedrock.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowBedrockStreamingInference",
      "Effect": "Allow",
      "Action": [
        "bedrock:InvokeModel",
        "bedrock:InvokeModelWithResponseStream"
      ],
      "Resource": [
        "arn:aws:bedrock:*::foundation-model/anthropic.claude-
         sonnet-4-5*",
        "arn:aws:bedrock:*:*:inference-profile/*"
      ]
    },
    {
      "Sid": "AllowBedrockDiscovery",
      "Effect": "Allow",
      "Action": [
        "bedrock:GetInferenceProfile",
        "bedrock:ListInferenceProfiles",
        "bedrock:ListFoundationModels",
        "bedrock:GetFoundationModel"
      ],
      "Resource": "*"
    }
  ]
}
```

### *A Full Array of Models and Enterprise Security*

As shown in **Listing 3**, AWS Bedrock supports a full array of foundation models from many of the major vendors. Some models can be significantly more costly than others and it is often important for enterprises to be able to put controls around which models their users are able to access. Because we are using an EC2 instance in AWS, we can take advantage of the tight controls enabled by AWS's role-based permissions system. Our EC2 IAM role profile, for example, contains permissions for just a single model.

If we wanted, we could allow newer, more expensive Claude models, or models from any of the vendors shown in **Listing 3**. **Listing 4** shows the permissions needed to allow just one model. Any of the model IDs could be added by an authorized AWS user to allow more options.

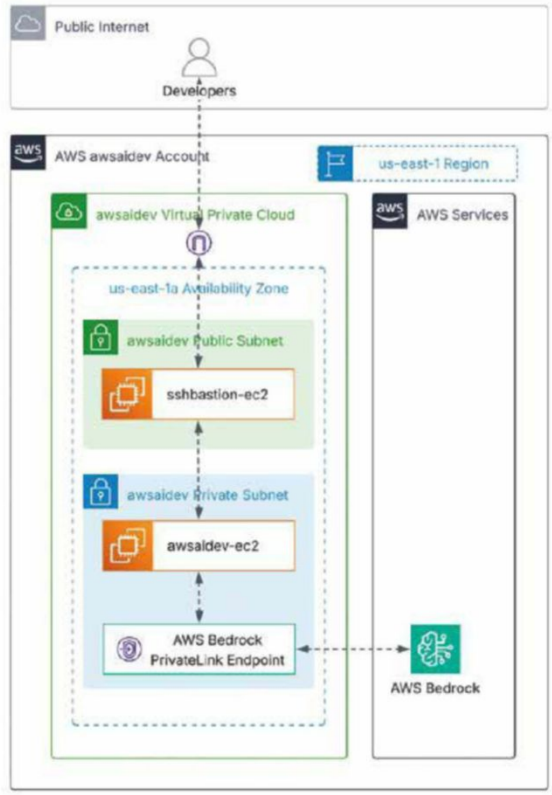



**Figure 7:** An air-gapped build using AWS Bedrock with PrivateLink.

When this role is assigned to a group, anyone in the groups would obtain the same permissions.

In short, AWS Bedrock provides a very robust and secure architectural framework for your data—both inputs and generated outputs. When combined with a private architecture like the one shown above using PrivateLink endpoints, where data is air-gapped from ever transiting the public internet, tight controls for compliance requirements or other purposes can be achieved. As shown, model access can be controlled by centralized permission policies, and accidental data sharing is not susceptible to human error when setting up a local machine.

## Closing Remarks

I hope that this article helped to pull back the curtain on some of the mystery surrounding the tools and techniques that you can apply when generating AI-assisted code. Like everything else, AI-assisted coding can be done well, or poorly. By leveraging a good framework, your knowledge can be amplified and your productivity will multiply. By focusing on being very intentional in managing the context sessions you generate each step of the way, the resulting output will be more predictable and better reflect your expectations. Also, by using a structured approach to engineering context and preserving provenance during all stages, the entirety of every choice and outcome can be reviewed, reused, tested, and refined by others later, thereby empowering your team and even others beyond your organization. And ***that*** is professional-grade.

Bill Catlan

# Roll for Initiative: Building an Offline D&D Character Sheet in a Single HTML File

Picture this: It's 7 p.m. on a Friday night. You and the nerds are gathered around the table. Snacks have been dispensed. Miniatures are laid out. Maps are drawn. Maybe someone is wearing a cloak. It's about to get weird. But these are modern times with modern problems. Gaming isn't the same as it was in decades past. It's not just paper, pencils, and dice. Sometimes it's virtual tabletops, digital libraries, and expensive subscription systems. Those dollars can add up. And so can the minutes spent waiting for the buffer screens. Yes, I know. I'm offending some old school sensibilities here. You old school D&D players are gnashing your teeth, saying all those modern technical impurities pollute the game. It goes against tradition! You're supposed to churn your own butter!

Fine. You're welcome to throw yourself upon the loom of progress, John Henry. Some of us don't like wearing a character sheet ragged with our pencil and erasers. Some of us don't like math. Let's let a formula do it. That's why God invented code.

Snark aside, digital applications in the tabletop role playing game space are a tremendous time saver. The rules can be dense, especially to new players, and when they run into calculations, it can bring the game to a screeching halt. (Is there such a thing as a quick round of combat in Dungeons and Dragons 5E? Is that even possible?)

What if we built a character sheet web app that works offline, lives in a single html file, and runs on any phone without installation? The official SRD–the system reference document–is Dungeon and Dragons' open-license rulebook, released under Creative Commons. We'll encode these rules as pure JavaScript functions, build a reactive UI that derives everything from a single state object, and save everything to the browser's local storage. No server. No framework. No build pipeline. One file.

## What We're Building

First, we have to nail down the scope. This is a weekend project, but from the jump, the bells and whistles will start piling up. Stay on target! Our character sheet needs to do just five things:

- Display and edit the six core ability scores with live modifier calculation
- Auto-calculate every skill bonus, saving throw, initiative, and passive perception derived from those scores
- Track hit points with damage and healing buttons, plus death saving throw pips
- Handle weapons and attack rolls—attack bonus, damage dice, damage type
- Save everything automatically, with an export/import system so characters survive a phone switch or cache wipe

The deployment is a neat and clean part of our story. When we're done, we'll have a single .html file. To share it, just email it. To install it on an iPhone, open it in Safari and tap "Add to Home Screen." On Android, open it in Chrome and do the same from the three-dot menu. On both platforms it appears on the home screen, launches full-screen without browser details, and works with airplane mode on. That's it. That's the entire deployment pipeline. Check out what we're gunning for in **Figure 1**.

## The Runes Behind the Magic: Encoding the D&D Math

Like any good project, we want to keep the code separated from the UI. Before we dive into the HTML or CSS, we'll focus on the JavaScript functions that encode the math for the SRD. These functions will take numerical inputs and return numerical outputs. Easy peasy. No side effects, no DOM access, no knowledge of what a button is. Easy to test, easy to reason about, completely reusable.

One of foundations of your character's mathematical progression is the ability modifier. Every ability score produces a modifier that gets added to dice rolls. The formula is pretty straightforward.

```
function getMod(score) {
  return Math.floor((score - 10) / 2);
}
```

Ability modifiers break down very simply. A score of 10 gives the player +0. A score of 16 provides +3, but a

**Jason Murphy**
Thestrangerous.substack.com
@jason-murphy.bsky.social

Jason is a writer, AI enthusiast, and media creator. As the producer and co-host of **Hacking the System** on the National Geographic Channel, he stole cars in Hollywood, made improvised smoke bombs, and prepared for the apocalypse. On YouTube, he co-created and hosted **the Modern Rogue**, where he explored hacking, lock-picking, and tradecraft. After publishing multiple speculative fiction novels and writing a produced screenplay, Jason is currently exiled in the desert, where he stares into the future with awe and terror. And he still wants to make a video game.

low score of 8 is a –1. That Math.floor is load-bearing. Without it, a score of 11 would return +0.5. I'm sure some uber-dorks would love that, but that's a hard pass from me, thank you very much.

The next big one to consider is how we integrate the calculation of the Proficiency Bonus. As characters gain levels, they get better at things they're trained in. It's kind of the whole point. The bonus starts at +2 at level 1 and increases every four levels, capping at +6. In the SRD this is available as a single table, but it can be compressed into a single function:

```
function profBonus(level) {
  return Math.floor((level - 1) / 4) + 2;
}
```

Let's test it:

- Level 1 Math.floor(0/4) + 2 = 2
- Level 5 Math.floor(4/4) + 2 = 3
- Level 17 Math.floor(16/4) + 2 = 6

We've built a simple formula that calculates that one fundamental table for us.

The ability and proficiency modifier are the two fundamental building blocks. With those, everything else falls into place. If you're unfamiliar, here's a quick lesson that glosses over the high points:

- A **skill bonus** is the relevant ability modifier plus the Proficiency Bonus if the character is trained in that skill.
- **Saving throws** work the same way.
- **Initiative** is just the Dexterity modifier added to a D20 roll.
- **Passive Perception** is 10 plus the Perception skill bonus.

For dice rolling, JavaScript's built-in Math.random() works just fine:

```
function d20() {
  return Math.floor(Math.random() * 20) + 1;
}

function rollDice(count, sides) {
  const results = [];
  for (let i = 0; i < count; i++) {
    results.push(Math.floor(Math.random() * sides)
      + 1);
```

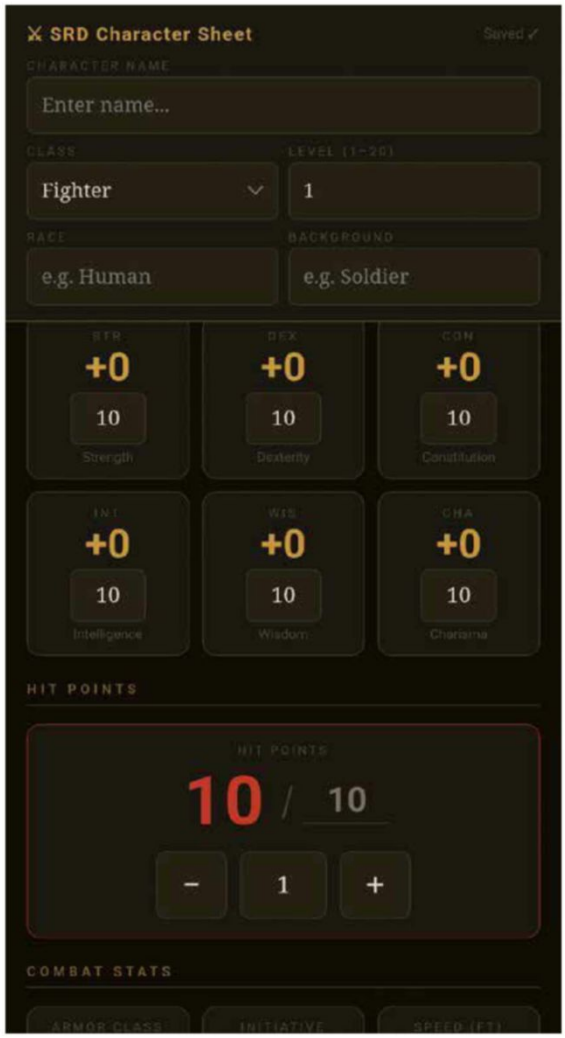



**Figure 1:** The Stats tab shows ability scores and hit points and (roughly) what we're trying to create.

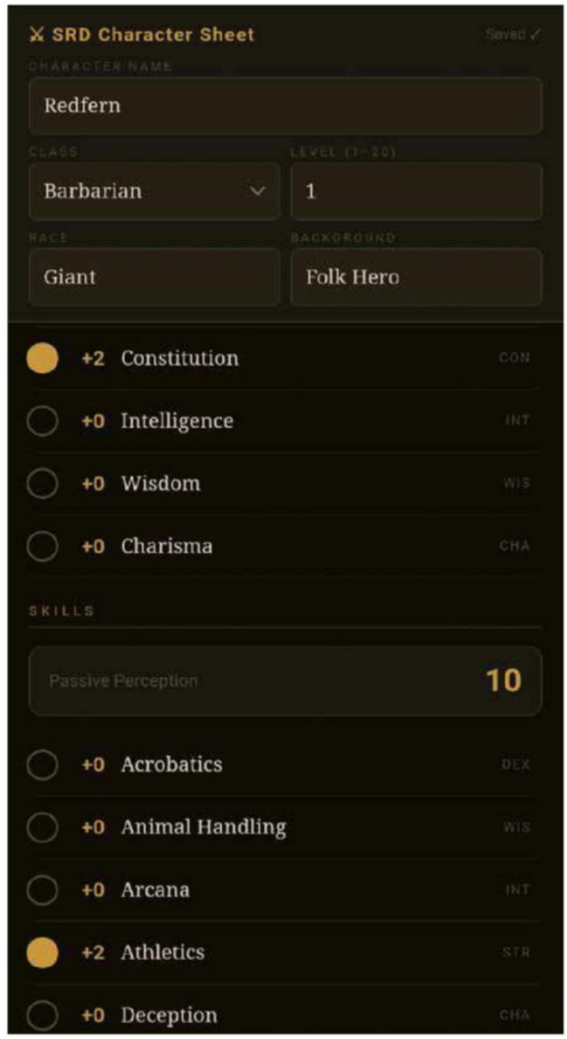



**Figure 2:** The Skills tab shows saving throws and skill proficiencies are calculated from the ability scores in Figure 1.

```
  }
  return results;
}
```

Take a look at our Skills tab in action in **Figure 2.**

## The Data Model: Defining a Character

Now that we have our rules engine built, we need to decide what a character looks like on the inside. Not when they're hurt in battle. We're not there yet. I mean as data. Everything displayed in the UI comes from this object.

```
let S = {
  name: '', cls: 'Fighter', level: 1, race: '', bg: '',
  ab:  { str:10, dex:10, con:10, int:10, wis:10,
    cha:10 },
  sp:  [],              // skill proficiencies
  svp: ['str', 'con'], // save proficiencies
  hp:  { cur: 10, max: 10 },
  ac: 10, spd: 30,
  ds: { s: [0,0,0], f: [0,0,0] }, // death saves
  wpn: [], notes: '', rolls: []
};
```

Let's take note as to what's included here. We have the six ability scores—Strength (str), Dexterity (dex), Constitution (con), Intelligence (int), Wisdom (wis), and Charisma (cha). But you'll see that items like the modifiers, saving throw totals, and initiative are not included. These are not stored but calculated on the fly whenever the UI needs to display them.

## User Interface: Putting the Math on the Screen

Our character sheet is built around five core tabs. These tabs collect the primary domains of knowledge players must keep track of—Stats, Skills, Combat, Rolls, and Notes. There's a lot more we could tack onto that, but that's the foundation.

The UI is structured around five tabs with those five domains, mapping to how players actually use a character sheet at the table. Throughout this, the render function is key. Instead of updating individual DOM elements when pieces of data change, it re-renders entire sections from the state object.

```
function renderSkills() {
  const list = el('skillsList');
  list.innerHTML = '';
  SKILLS.forEach(sk => {
    const bonus = getMod(S.ab[sk.a])
      + (S.sp.includes(sk.n) ? profBonus(S.level) : 0);
    const row = mkSkillRow(
      sk.n, sk.a.toUpperCase(), bonus,
      S.sp.includes(sk.n),
      () => toggleSP(sk.n), // pip tap toggles
                            proficiency
      () => rollSkill(sk.n) // row tap triggers a roll
    );
    list.appendChild(row);
  });
}
```

When a player switches a proficiency toggle in our interface, the state array is updated and renderSkills() is called again. The list of 18 items is rapidly rebuilt. If an ability score changes, its modifier updates immediately. The re-render cascades down across everything else.

```
function onAb(key, val) {
  S.ab[key] = Math.max(1, Math.min(30,
    parseInt(val) || 10));
  const m = el('m-' + key);
  if (m) m.textContent = fmt(getMod(S.ab[key]));
  renderDerived();
  renderSkills();
  renderSaves();
  renderWpn();
  save();
}
```

## Roll to Hit: The Combat Tab

Attack rolls follow the same pattern as skill checks—roll a d20, add a modifier to increase your chances to hit your enemy. There's a second stage, though—a hit triggers a damage roll, depending on the specific weapon's damage dice. Each weapon is treated as a simple object. For example:

```
{ n: 'Longsword', dmg: '1d8', dt: 'slashing',
  a: 'str', prof: true }
```

Below we have our attack roll function. It parses the damage string for the weapon, generates the rolls, and adds the ability modifier to the total.

```
function rollAttack(i) {
  const w  = S.wpn[i];
  const am = getMod(S.ab[w.a || 'str']);
  const pb = w.prof ? profBonus(S.level) : 0;
  const ar = d20();
  let extra = '';

  const m = (w.dmg || '1d6').match(/(\d+)d(\d+)/i);
  if (m) {
    const rolls = rollDice(parseInt(m[1]), parseInt(m[2]));
    const dmg   = rolls.reduce((a,b) => a+b, 0) + am;
    extra = 'Dmg: ' + dmg +
      ' (' + rolls.join('+') + fmt(am) + ') ' + w.dt;
  }
  addRoll((w.n || 'Weapon') + ' Attack', ar, am + pb, extra);
}
```

Each roll pops up in the Rolls tab. We'll make the outcomes color-coded. A natural 20 (a good, good thing, for those of you who don't know) is highlighted in gold. A natural 1 (a bad, bad thing) is greyed out. We break up everything else into categories based on the modified total:

- Strong (15+)
- Moderate (10–14)
- Low (below 10)

The Roll History is clean and elegant, as you can see in **Figure 3**.

## Persistence: Setting Up Efficient Local Storage

Saving to local storage is simple. It's a key-value store built into every browser, persistent across sessions, accessible with two methods:

### 6 Stats/18 Skills

Every Dungeons and Dragons character is built around six core ability scores—Strength, Dexterity, Constitution, Intelligence, Wisdom, and Charisma. Each of these typically fall within the range of 1 to 20, with 10 representing the average human capability. Each of these scores produces a modifier, the number added to the dice roll. A score of 16 gives a +3 modifier. A score of 8 gives a -1. When a player attempts something difficult, they roll a 20-sided die and add the relevant modifier. The Dungeon Master sets a Difficulty Class (DC) for the player's roll. If their roll plus modifier total meets or beats the DC, the player succeeds.

These 6 ability scores also cover 18 more specific skills. Constitution has no associated skills, but Strength covers Athletics. Dexterity is tied to Acrobatics, Sleight of Hand, and Stealth. Intelligence covers skills like Arcana, History, Investigation, Nature, and Religion. Wisdom is for Animal Handling, Insight, Medicine, Perception, and Survival. And Charisma covers Deception, Intimidation, Performance, and Persuasion. Characters are proficient in a subset of these skills based on what class and background they choose. The Proficiency Bonus gets added to those specific roles. A Ranger is usually good at Animal Handling. A Rogue is good at Sleight of Hand, and so on.

```
function save() {
  collectDOM();
  try {
    localStorage.setItem('srd3', JSON.stringify(S));
    el('saveIndicator').textContent = 'Saved   ';
  } catch(e) {
    el('saveIndicator').textContent = 'Storage full';
  }
}

function load() {
  const raw = localStorage.getItem('srd3');
  if (raw) S = JSON.parse(raw);
}
```

Call save() at the end of every event handler. Call load() once on page init before the first render. The character is always there when the player reopens the app, even when the phone is rebooted. For you neophytes and non-coders, this is only on that specific device and browser. The character you save on Chrome isn't available on Brave. The character you save on your Android isn't available on your iPhone, and so on. When a player switches devices, they'll need to export the system.

## Cross-Platform Mobile: iOS and Android Tweaks

This is where I ran into the most amount of trouble. It took a surprising amount of elbow grease to get it working on both of the big two platforms. (I **think** I got everything, but maybe you'll find some improvements we can make.)

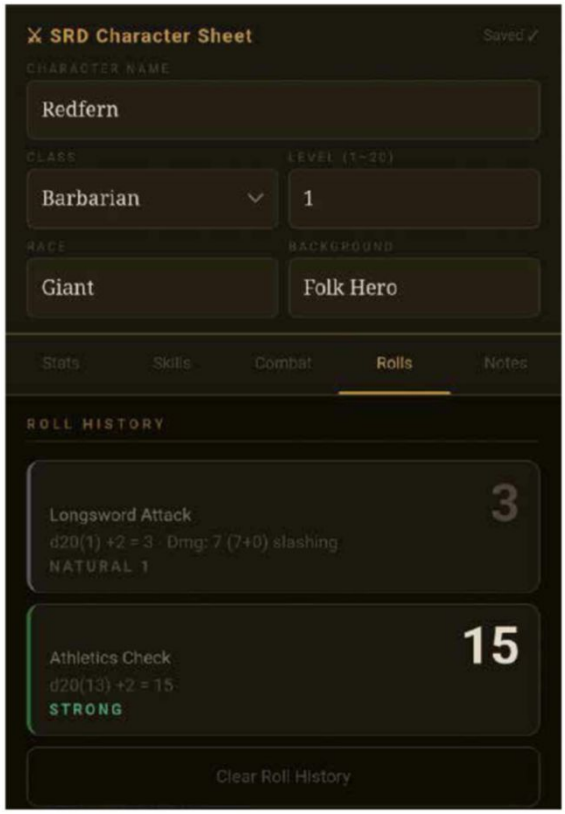



**Figure 3:** On the Rolls tabs, we can see our history of dice rolls.

### *Offline Capability and Home Screen Installation*

Both iOS Safari and Android Chrome support adding a page to the home screen as a standalone app. When launched this way, the address bar and browser navigation buttons are hidden. The app fills the screen like any native application would. iOS requires specific meta tags. Android Chrome reads a PWA manifest—a small JSON file that tells the browser the app's name, icon, and display preferences, and signals that it's installable as a standalone home screen app.

```
<!-- iOS Safari -->
<meta name="apple-mobile-web-app-capable"
      content="yes">
<meta name="apple-mobile-web-app-status-bar-style"
      content="black-translucent">
<meta name="apple-mobile-web-app-title"
      content="SRD Sheet">

<!-- Android Chrome -->
<meta name="mobile-web-app-capable" content="yes">
<meta name="theme-color" content="#0f0e0c">

<!-- Inline manifest keeps the file self-contained -->
<link rel="manifest" href="data:application/json,...">
```

### *System Fonts—Eliminating Dependencies*

Using Google Fonts could break things at first. It's an external font service, so the app could look broken the first time it opens offline. The device's own font stack has the fix, however.

```
:root {
  --font-ui: -apple-system, BlinkMacSystemFont,
             'Segoe UI', Roboto, sans-serif;
}
```

- **-apple-system** resolves to San Francisco on Apple devices.
- **BlinkMacSystemFont** handles older macOS Chrome.
- **Segoe UI** covers Windows.
- **Roboto** covers Android.

### *Touch Targets and Tap Delay*

It just keeps getting messier and fiddlier! This is the kind of minute UI work that turns my brain into lumpy stew. Apple's Human Interface Guidelines and Google's Material Design both specify a minimum tap target of 44×44 pixels. Every interactive element in our character sheet meets this minimum. It's also worth noting that Android Chrome added a 300ms delay before registering taps. One CSS property eliminates it:

```
.nav-btn {
  touch-action: manipulation;
}
```

**touch-action: manipulation** tells the browser this element responds to taps and drags but not zoom, so the delay can be skipped.

### *Safe Area Insets for Notched Devices*

iPhones from the series X and beyond, along with many Androids, have notches or home indicator bars that can obscure content on the page or app. CSS environment variables handle this elegantly:

## Resources

- **D&D 5e System Reference Document:** dnd.wizards.com/resources/systems-reference-document
- **Creative Commons Attribution 4.0:** creativecommons.org/licenses/by/4.0
- **Web Share API (MDN):** developer.mozilla.org/en-US/docs/Web/API/Web_Share_API
- **Complete source code:** Available on the Code Magazine website with this article.

```
:root {
  --sat: env(safe-area-inset-top,    0px);
  --sab: env(safe-area-inset-bottom, 0px);
}

.header { padding-top: calc(12px + var(--sat)); }
.app    { padding-bottom: calc(80px + var(--sab)); }
```

On devices without notches, this just zeroes out. On devices with notches, the content is pushed into the visible area. The viewport meta tag must also opt in, as seen below.

```
<meta name="viewport"
  content="width=device-width, initial-scale=1.0,
    viewport-fit=cover">
```

### *Dynamic Viewport Height*

One last minor tweak to our UI is a subtle iOS issue. The "browser chrome"—address bar, tab bar—is counted in the standard 100vh measurement, making a full-height element taller than the visible screen. A newer unit fixes this:

```
body { min-height: 100dvh; }
```

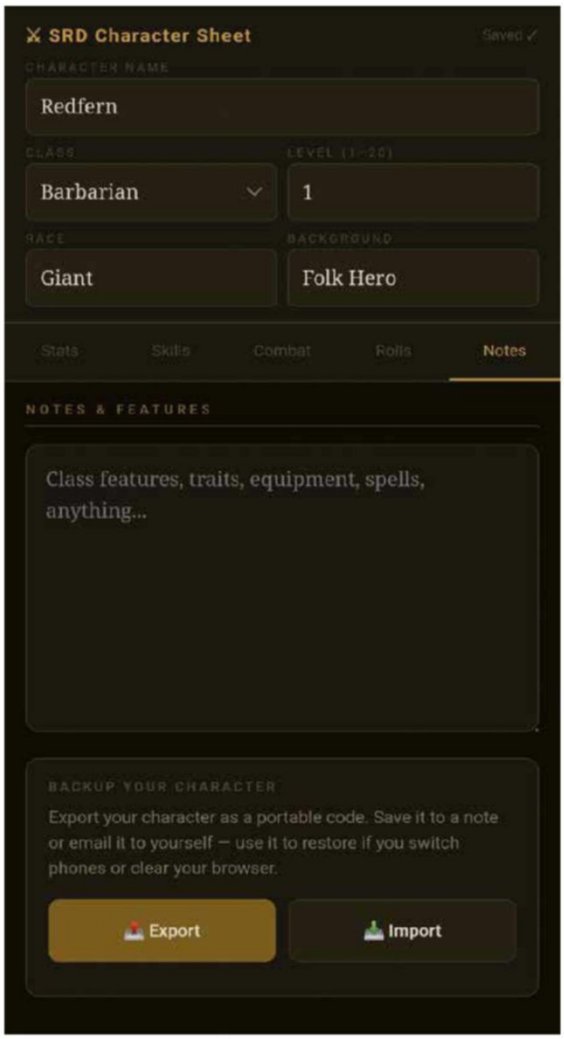



**Figure 4:** On the Notes tab we can store notes or export our character details as a backup.

**100dvh** adjusts dynamically and will hide that unnecessary browser information. This is supported in iOS Safari 16+ and Android Chrome 108+, so it will cover most modern mobile devices.

## Export and Import: Surviving a Cache Wipe

**localStorage** is persistent but vulnerable. A browser wipe can destroy your character or maybe you got a new phone. Naturally, we must be able to make our character sheet transferable. An easy way to do that is to encode the entire state as a portable text string.

```
function exportChar() {
  collectDOM();
  // JSON -> UTF-8 -> base64
  const code = btoa(unescape(
    encodeURIComponent(JSON.stringify(S))));

  if (navigator.share) {
    // Opens native share sheet on iOS and Android
    navigator.share({ title: S.name + ' - SRD Backup',
      text: code });
  } else if (navigator.clipboard) {
      navigator.clipboard.writeText(code);
  }
}
```

**navigator.share** is the Web Share API. On iOS it opens the native share sheet: AirDrop, Messages, Notes, Mail. When using Android it opens the system share dialog. If neither is available, the clipboard is the fallback. Import runs the process in reverse: paste the code, decode the base64, parse the JSON, rebuild state.

We've placed the Export/Import buttons on the Notes tab, right below Notes and Features. You can see its simplicity in **Figure 4**.

## The Journey Ahead

Here's where the real fun starts. As we play with our character sheets, we'll start to see all sorts of things we want to expand upon. The options are nearly infinite:

**Spell slot tracker—**This one is a must. We'll definitely need a section showing spell slots by level (1st through 9th) with toggles for each. The SRD provides the slot table for each character class. The rules engine is unchanged, of course. This feature will require a UI and state addition.

**Multiple characters—**Who just has one character? New players, that's who. To rectify that limitation for after you've played a few weeks or months, we can replace the single state object with an array stored under different **localStorage** keys. We'll add a character select screen at launch. The same **getMod()** and **profBonus()** functions work for every character unchanged.

**Export to PDF—**The browser's print dialog renders a clean version of the sheet to PDF if you add a print stylesheet. The **jsPDF** library can generate properly formatted PDFs entirely in the browser, still with no central server, all on the device.

### The SRD

The System Reference Document—the SRD—is the official and open-licensed version of the Dungeons and Dragons 5th Edition rules. It includes all of the core mechanics, species, classes, spells, etc., along with the mechanical aspects of the game, that underlies Dungeons and Dragons. In 2023, Wizards of the Coast, the publisher of the game, released version 5.1 under the Creative Commons Attribution 4.0 International License. That means anyone can build on it, publish with it, or even build a commercial product using the SRD as long as they credit the source.

**AI narration—**This one is the flash. This turns our practical little app into something sassy. Imagine—a player rolls a skill check. That result, along with the character's pertinent information, is passed over to an LLM's API. A one sentence narration for the action is returned. A level 3 Fighter with a +5 Athletics bonus rolling a 17 might see:

> Your boots find purchase on the slick stone. You haul yourself up over the wall with finesse.

Our simple rules engine determines the math. The AI will provide some vague story flavor. Those are two different types of architectural problems in the same result, so it will certainly result in some interesting obstacles.

**Homebrew rules—**Every table has its own house rules, little nudges to the SRD that makes it more fun or convenient for you or your players. And one of the best things about building your own tool is that you can modify it to your heart's content. If you change getMod(), every derived stat in the app updates instantly. There's no expensive subscription required.

Jason Murphy

CODE

Jul/Aug 2026
Volume 27 Issue 4

Group Publisher
Markus Egger

Editor-in-Chief
Rod Paddock

Managing Editor
Ellen Whitney

Content Editor
Erik Ruthruff

Writers in This Issue
Bill Catlan
Joydip Kanjilal
Wei-Meng Lee
Jason Murphy
Otto Dobretsberger
Sonu Kapoor
Sahil Malik

Technical Reviewers
Markus Egger
Rod Paddock

Production
Friedl Raffeiner Grafik Studio
www.frigraf.it

Graphic Layout
Friedl Raffeiner Grafik Studio in collaboration with onsight (www.onsightdesign.info)

Printing
Fry Communications, Inc.
800 West Church Rd.
Mechanicsburg, PA 17055

Advertising Sales
Erik Ruthruff
eruthruff@codemag.com

Circulation & Distribution
General Circulation: EPS Software Corp.
Newsstand: Ingram Periodicals, Inc.
International Bonded Couriers (IBC)
Media Solutions
Source Interlink International

Circulation Manager
Colleen Cade
832-717-4445 ext. 28
ccade@codemag.com

Subscriptions

US subscriptions are $29.99 USD for one year.
Subscriptions outside the US are $50.99 USD.
Payments should be made in US dollars drawn on a US bank. American Express, MasterCard, Visa and Discover credit cards accepted.
Back issues are available. For subscription information, email subscriptions@code-magazine.com or contact customer service at 832-717-4445 ext. 9.

Subscribe online at
www.code-magazine.com

CODE Developer Magazine
EPS Software Corporation / Publishing Division
6605 Cypresswood Drive, Ste 425, Spring, Texas 77379 USA
Phone: 832-717-4445

# YOUR PARTNER FOR CUSTOM SOFTWARE SOLUTIONS


REAL BUSINESS VALUE FOR AI

TRAINING / MENTORING

CUSTOM APPLICATION DEVELOPMENT

CONTINGENT IT STAFFING

Is your development team struggling to complete business-critical projects on time?
Are you looking to harness cutting-edge technologies, including AI, for maximum impact?

## CODE EXCELS IN:

- **AI integration for enhanced functionality**
- **.NET web and desktop development**
- **Azure cloud migration and transformation**
- **Blazor development**
- **Mobile app creation**
- **Staffing, training and mentoring**

Let CODE transform your software challenges into competitive advantages.

**CONTACT US TODAY. NO STRINGS. NO COMMITMENT.**

**codemag.com/code**
832-717-4445 ext. 9 • info@codemag.com

CODE
STAFFING

# UNLOCK STAFFING EXCELLENCE

**Top-Notch IT Talent, Contract Flexibility, Happy Teams, and a Commitment to Customer Success Converge with CODE Staffing**

Our IT staffing solutions are engineered to drive your business forward while saving you time and money. Say goodbye to excessive overhead costs and lengthy recruitment efforts. With CODE Staffing, you'll benefit from contract flexibility that caters to both project-based and permanent placements. We optimize your workforce strategy, ensuring a perfect fit for every role and helping you achieve continued operational excellence.

***Ready to Discuss Your IT Staffing Needs?***

Visit our website to find out more about how we are changing the staffing industry.

Website: codestaffing.com
Direct: +1 (832) 717-4445
Email: info@codestaffing.com