AI VOID FIELD GUIDE

Mastering Flue: Building Production-Ready AI Agents with TypeScript

Learn to build, deploy, and manage robust AI agents using the Flue Framework, focusing on its unique agent harness architecture, state management, and production deployment with TypeScript.

AI VOID

2026-06-03 19:03 IST

Contents

CHAPTER 01

Welcome to Flue: The Agent Harness Architecture Explained

Welcome, fellow AI systems engineer! Building truly intelligent AI agents for production goes far beyond simply calling a Large Language Model (LLM) API. It demands a robust system capable of managing state, integrating external tools, and executing complex logic reliably and securely. This is precisely the challenge that frameworks like Flue aim to solve.

In this chapter, we'll dive deep into Flue, an "agent harness" framework designed to transform your AI agents from basic prompt wrappers into sophisticated, deployable entities. We'll demystify its unique architecture, contrast it with traditional LLM SDKs, and guide you through building your very first Flue agent in TypeScript. By the end, you'll have a foundational understanding and a working agent, ready for more advanced concepts and eventual deployment to platforms like Cloudflare Workers.

Why an "Agent Harness" Matters: Beyond LLM API Calls

Imagine you're building a personal assistant agent. It needs to remember past conversations, look up information online, schedule appointments, and maybe even write code. A simple LLM API call can generate text, but it can't inherently do any of these complex tasks, nor can it remember context across interactions.

This is where the "agent harness" concept, as embodied by Flue, becomes critical. As of its latest known state (June 2026), Flue provides the complete infrastructure—the "workbench"—around your core agent logic. It elevates an LLM from a smart text generator to a capable, stateful, and tool-using entity by offering:

- **Stateful Sessions:** Agents can maintain conversation history, internal variables, and context over extended interactions.
- **Tool and Skill Integration:** Agents gain the ability to use external functions, databases, APIs, or even sandboxed environments for tasks like web search or code execution.
- **Sandboxed Execution:** For advanced "coding agents," Flue can provide secure, isolated environments (like sandboxed filesystems or shell access) where agents can safely write, test, and run code.

- **Deployment & Exposure:** Flue streamlines the process of exposing your agents via standard protocols (HTTP, WebSockets) for seamless integration into real-world applications.

Flue vs. LLM SDKs: A Clear Distinction

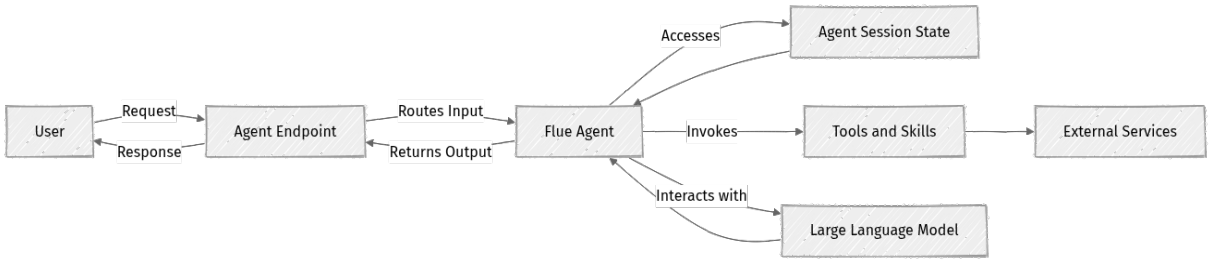
It's easy to confuse an agent harness with a standard LLM SDK. Let's clarify the difference:

- **LLM SDK Wrapper (e.g., OpenAI, Anthropic SDKs):**
    - **Purpose:** Primarily handles direct communication with a specific LLM API.
    - **Capabilities:** Sends prompts, receives completions, manages API keys, handles rate limits.
    - **Analogy:** A phone to call a smart friend.
    - **Limitation:** It's a communication layer. All the logic for state management, tool use, decision-making, and deployment must be built around the SDK by you.
- **Flue (Agent Harness Framework):**
    - **Purpose:** Provides the entire operating environment for an AI agent, orchestrating its lifecycle.
    - **Capabilities:** Offers a runtime, manages sessions, orchestrates tool calls, handles input/output, and simplifies deployment. It uses LLM SDKs internally as one of its components.
    - **Analogy:** A personal assistant (the agent) with a desk, filing cabinets (state), a phone (LLM SDK), and various specialized tools (web search, calculator) to accomplish complex tasks.
    - **Benefit:** Reduces development time, enforces a consistent structure, and provides production-ready features out-of-the-box, allowing you to focus on the agent's unique intelligence.

📌 **Key Idea:** Flue doesn't replace your LLM SDK; it provides the robust infrastructure that uses your LLM SDK to build complete, intelligent, and deployable AI agents.

The Flue Agent Harness Architecture

Let's visualize how Flue structures an agent system. This diagram illustrates the flow from a user request through the agent's core components.



Understanding the Agent Workflow:

1. **User Request:** A client (web app, mobile app, another service) initiates an interaction with your agent.
2. **Agent Endpoint:** This is the public interface of your deployed agent, often an HTTP or WebSocket endpoint. In production, Flue's `AgentRouteHandler` typically manages this.
3. **Flue Agent Instance:** This is your custom TypeScript agent code. It orchestrates the overall logic, deciding when to use the LLM, when to call tools, and how to manage state.
4. **Agent Session (State):** This component is crucial for stateful interactions. It stores conversational history, user preferences, and any other data the agent needs to remember across multiple turns.
5. **Tools & Skills:** When the agent needs to perform an action beyond text generation (e.g., retrieve data, perform calculations), it invokes one of its integrated tools or skills.
6. **LLM:** The agent interacts with one or more LLMs for natural language understanding, generation, or complex reasoning.
7. **External Services:** Tools often connect to external APIs, databases, or even sandboxed execution environments for specific tasks.

⚡ **Real-world insight:** This modular design allows for high flexibility. You can easily swap LLM providers, add new tools, or change deployment targets without extensive rework of your core agent logic. This is vital for adapting to evolving AI capabilities and business needs.

TypeScript-First Development

Flue is built with TypeScript at its core. This means you'll define your agents, their inputs, outputs, and integrated tools using strong types. This approach provides:

- **Type Safety:** Catches errors early in development, reducing runtime bugs.
- **Improved Readability:** Clear interfaces make it easier to understand data flows.

- **Enhanced Developer Experience:** Modern IDEs provide excellent autocompletion and refactoring support.

For production-minded engineers, TypeScript is a significant advantage for building maintainable and robust AI systems.

Setting Up Your Flue Environment

Let's prepare your development machine to build your first Flue agent.

Prerequisites

Please ensure you have the following installed:

- **Node.js:** We recommend Node.js `v20.x` LTS or higher. You can download the installer from the official Node.js website.
- **npm or Yarn:** These package managers are included with Node.js.
- **TypeScript Knowledge:** Familiarity with TypeScript syntax and concepts is crucial for working with Flue.
- **Basic LLM Understanding:** Knowing about prompts, completions, and how LLMs process information will be helpful.

Step 1: Initialize Your Project Directory

First, create a new folder for your project and initialize a Node.js project within it.

```
mkdir my-flue-agent
cd my-flue-agent
npm init -y
```

This command sets up a `package.json` file, which will manage your project's dependencies and scripts.

Step 2: Install Flue Core and Development Dependencies

Next, install the core Flue library and essential development tools.

```
npm install @flue/core typescript @types/node ts-node express @types/express
```

Let's understand what each package provides:

- `@flue/core`: This is the foundational Flue library, containing the `Agent` base class and core abstractions.

- `typescript`: The TypeScript compiler itself, which converts your `.ts` files into `.js`.
- `@types/node`: Type definitions for Node.js, enabling TypeScript to understand Node.js APIs.
- `ts-node`: A utility that allows you to execute TypeScript files directly in Node.js during development without a prior compilation step.
- `express`: A popular Node.js web framework. We'll use it to create a simple local server for testing our agent.
- `@types/express`: TypeScript type definitions for the Express.js framework.

Step 3: Configure TypeScript

We need a `tsconfig.json` file to guide the TypeScript compiler. Create this file in your project's root directory:

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "Node16",
    "moduleResolution": "Node16",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true,
    "outDir": "./dist"
  },
  "include": ["src/**/*.ts"],
  "exclude": ["node_modules"]
}
```

Key `compilerOptions` Explained:

- `target: "ES2022"`: Specifies the ECMAScript version that your TypeScript code will be compiled down to. ES2022 offers modern JavaScript features.
- `module: "Node16"` and `moduleResolution: "Node16"`: These settings ensure that TypeScript uses Node.js's modern module resolution strategy, aligning with how modules are handled in recent Node.js versions.
- `esModuleInterop: true`: Allows you to use the more common `import x from "y"` syntax even for modules that traditionally use `require("y")`.
- `strict: true`: Activates a suite of strict type-checking options, which is highly recommended for robust, production-grade code.
- `outDir: "./dist"`: Designates the directory where compiled JavaScript files will be placed.

Step 4: Create a `src` Directory

Organize your source code by creating a `src` directory:

```
mkdir src
```

With these steps, your development environment is now fully prepared to build Flue agents!

Building Your First Flue Agent: A Simple Greeter

Let's create a basic Flue agent that simply greets a user by name. This will introduce you to the fundamental structure of any Flue agent.

Step 1: Define the Greeter Agent

Inside your newly created `src` directory, create a file named `greeterAgent.ts`.

```
// src/greeterAgent.ts
import { Agent, AgentContext } from '@flue/core';

/**
 * Defines the expected input structure for our GreeterAgent.
 * This agent needs a 'name' to personalize the greeting.
 */
interface GreeterAgentInput {
  name: string;
}

/**
 * Defines the expected output structure from our GreeterAgent.
 * It will return an object containing the 'greeting' message.
 */
interface GreeterAgentOutput {
  greeting: string;
}

/**
 * The GreeterAgent class extends Flue's base Agent.
 * It's generic, specifying its input and output types for strong type safety.
 */
export class GreeterAgent extends Agent<GreeterAgentInput,
GreeterAgentOutput> {
  /**
   * The core method of any Flue agent. This is where your agent's
   * specific business logic and decision-making resides.
   *
   * @param input The structured input data for this agent invocation.
   * @param context Provides access to Flue's runtime environment (e.g.,
session, tools).
   * @returns A Promise that resolves to the structured output of the agent.
   */
  async handle(
```

```
    input: GreeterAgentInput,
    context: AgentContext
  ): Promise<GreeterAgentOutput> {
    // We extract the 'name' property from the agent's input.
    const { name } = input;

    // Construct a personalized greeting message using a template literal.
    const greeting = `Hello, ${name}! Welcome to Flue.`;

    // Log the generated greeting to the console for debugging and
observation.
    console.log(`GreeterAgent generated: ${greeting}`);

    // Return the structured output, conforming to the GreeterAgentOutput
interface.
    return { greeting };
  }
}
```

Code Explanation:

- `import { Agent, AgentContext } from '@flue/core';`: We import `Agent`, the base class for all Flue agents, and `AgentContext`, which provides runtime utilities.
- `interface GreeterAgentInput { name: string; }`: We define the expected input type. This enhances type safety and makes the agent's contract clear.
- `interface GreeterAgentOutput { greeting: string; }`: We define the expected output type. The agent's `handle` method must return an object matching this structure.
- `export class GreeterAgent extends Agent<GreeterAgentInput, GreeterAgentOutput> { ... }`: Our `GreeterAgent` class inherits from Flue's `Agent` class. The generic types `<GreeterAgentInput, GreeterAgentOutput>` tell Flue (and TypeScript) what kind of input this agent expects and what output it will produce.

- `async handle(input: GreeterAgentInput, context: AgentContext): Promise<GreeterAgentOutput> { ... }`: This is the **most important method** for any Flue agent. It's where your agent's core logic executes.
    - It's `async` because real-world agent operations (like calling an LLM or a tool) are typically asynchronous.
    - `input`: Contains the data passed to the agent, strictly typed by `GreeterAgentInput`.
    - `context`: An object providing access to Flue's runtime context, including session, tools, and potentially other environment variables. For this simple agent, we don't use it, but it's always available.
    - `Promise<GreeterAgentOutput>`: Specifies that the method will return a promise that resolves to an object matching our `GreeterAgentOutput` interface.
- `const { name } = input;`: We safely extract the `name` from the typed `input` object.
- `const greeting = \`Hello, ${name}! Welcome to Flue.\`;`: A simple string interpolation to create the greeting.
- `console.log(...)`: A useful line for observing the agent's internal activity during development.
- `return { greeting };`: The agent returns its structured output.

Testing Your Flue Agent Locally with Express

To interact with our `GreeterAgent`, we need a way to send it input and receive its output. For local development, we'll use a simple Express.js server as a test harness.

Step 1: Create a Local Server Entry Point

Create a new file in your `src` directory named `server.ts`. This file will set up our Express server and expose our `GreeterAgent`.

```
// src/server.ts
import express from 'express';
import { GreeterAgent } from './greeterAgent';
import { AgentContext } from '@flue/core'; // We'll use AgentContext, even if
empty for now

// Initialize an Express application
const app = express();
// Enable JSON body parsing for incoming requests
```

```
app.use(express.json());

// Create an instance of our GreeterAgent
const greeterAgent = new GreeterAgent();

// Define a POST endpoint to interact with our agent
app.post('/greet', async (req, res) => {
  try {
    // For local testing, we directly invoke the agent's handle method.
    // In a production Flue deployment (e.g., Cloudflare Workers),
    // Flue's AgentRouteHandler would abstract this request parsing.
    const agentInput = req.body; // Express automatically parses JSON body
into req.body

    // Create an empty context for this simple agent,
    // but in real agents, this would contain session info, tools, etc.
    const context: AgentContext = {};

    const result = await greeterAgent.handle(agentInput, context);
    res.json(result); // Send the agent's structured output as a JSON response
  } catch (error) {
    console.error('Error handling agent request:', error);
    // Provide a generic error response for security and simplicity
    res.status(500).json({ error: 'Internal server error processing agent
request' });
  }
});

// Start the Express server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Flue Greeter Agent local server listening on port ${PORT}`);
  console.log(`To test, send a POST request to http://localhost:${PORT}/
greet`);
  console.log(`Example JSON body: {"name": "Alice"}`);
});
```

Step 2: Add npm Scripts for Convenience

To easily run and build your project, add these scripts to the `scripts` section of your `package.json` file. Replace the placeholder versions with the actual versions installed.

```
// package.json (update 'scripts' and 'dependencies' sections)
{
  "name": "my-flue-agent",
  "version": "1.0.0",
  "description": "My first Flue agent project.",
  "main": "dist/server.js",
  "scripts": {
    "start": "ts-node src/server.ts",
    "build": "tsc",
    "serve": "node dist/server.js"
  },
  "keywords": ["flue", "ai-agent", "typescript"],
  "author": "AI Expert",
  "license": "ISC",
  "dependencies": {
```

```
    "@flue/core": "^0.x.x",
    "express": "^4.19.2",
    "typescript": "^5.5.3"
  },
  "devDependencies": {
    "@types/express": "^4.17.21",
    "@types/node": "^20.14.9",
    "ts-node": "^10.9.2"
  }
}
```

Note: The versions above are examples. `npm install` will fetch the latest compatible versions, which you can then update in your `package.json` if you wish to pin them.

Step 3: Run Your Local Agent Server

Now, let's start your Express server using the `start` script we just defined.

```
npm start
```

You should see output indicating the server is running:

```
Flue Greeter Agent local server listening on port 3000
To test, send a POST request to http://localhost:3000/greet
Example JSON body: {"name": "Alice"}
```

Step 4: Test Your Agent

You can test your running agent using `curl` from your terminal, or a tool like Postman or Insomnia.

```
curl -X POST -H "Content-Type: application/json" \
     -d '{"name": "Alice"}' \
     http://localhost:3000/greet
```

If everything is set up correctly, you should receive a JSON response:

```
{"greeting":"Hello, Alice! Welcome to Flue."}
```

Congratulations! You've successfully built and tested your first Flue agent locally.

Understanding `AgentRouteHandler` for Production Deployment

While our Express server is great for local testing, Flue provides a more integrated solution for exposing agents in production: the `AgentRouteHandler` from `@flue/server`. This component is specifically designed to bridge your Flue agent with serverless environments like Cloudflare Workers.

`AgentRouteHandler`'s primary role is to:

- Receive incoming HTTP or WebSocket requests.
- Parse these requests into the standardized `AgentInput` format expected by your Flue agent.
- Manage agent sessions, ensuring state is correctly loaded and saved.
- Invoke your agent's `handle` method with the proper `AgentContext`.
- Format the agent's `AgentOutput` back into a standard HTTP/WebSocket response.

⚡ **Real-world insight:** For serverless platforms like Cloudflare Workers, `AgentRouteHandler` acts as the direct `fetch` event handler, abstracting away much of the boilerplate for agent invocation and response handling. This is a key differentiator for Flue as a production-minded framework.

Here's a conceptual example of how `AgentRouteHandler` would be used in a Cloudflare Worker, as referenced in the official Flue documentation:

```
// worker.ts (Conceptual Cloudflare Worker entry point)
import { AgentRouteHandler } from '@flue/server';
import { GreeterAgent } from './greeterAgent'; // Your Flue agent

// Instantiate your agent
const greeterAgent = new GreeterAgent();

// Create the AgentRouteHandler, passing your agent instance
const agentHandler = new AgentRouteHandler(greeterAgent);

// Export the Worker's fetch handler, delegating to AgentRouteHandler
export default {
  async fetch(request: Request, env: Env, ctx: ExecutionContext): Promise<Resp
onse> {
    // The AgentRouteHandler processes the incoming request and invokes the
agent
    return agentHandler.fetch(request, env, ctx);
  },
};
```

We'll explore full deployment to Cloudflare Workers and other production considerations in a dedicated future chapter. For now, understand that `AgentRouteHandler` is the intended gateway for your agents in scalable, production environments.

Mini-Challenge: Personalize the Greeting with Time

Now that you've built and tested a basic agent, let's make it a little more dynamic.

Challenge: Modify the `GreeterAgent` to include the current time in the greeting message. For example, "Good morning, Alice! Welcome to Flue. It's 10:30 AM."

Hint:

- You can obtain the current local time string using `new Date().toLocaleTimeString()`.
- Remember to update the `GreeterAgentOutput` interface if you decide to return the time as a separate field, or simply embed it in the existing `greeting` string.

What to observe/learn: This challenge reinforces your understanding of modifying agent logic, handling basic data manipulation within the `handle` method, and ensuring that your agent's output consistently matches its defined `Output` interface.

Common Pitfalls & Troubleshooting

As you build and experiment with Flue, you might encounter some common issues. Here’s how to approach them:

- **`TypeError: Cannot read properties of undefined (reading 'name')`**: This error typically means the `name` property was not found in the `input` object passed to your agent. Double-check your `curl` command or Postman request to ensure you are sending a valid JSON body, like `{"name": "Alice"}`. Also, ensure your Express server is using `app.use(express.json());` to correctly parse incoming JSON.

- **TypeScript Compilation Errors**: If `npm start` (or `npm run build`) fails with errors, carefully read the messages. They usually provide the file name and line number. Common causes include:
    - **Missing Imports**: Did you forget to import `Agent` or `AgentContext` from `@flue/core`?
    - **Type Mismatches**: Is the object returned by your `handle` method compatible with the `GreeterAgentOutput` interface you defined? TypeScript will enforce this.
    - **`tsconfig.json` Issues**: Ensure your `tsconfig.json` is correctly configured, especially the `include` and `outDir` paths, and that `strict: true` helps catch potential issues early.
- **Server Not Starting**: If your Express server doesn't log the "listening on port..." message, it might be due to another process already using port `3000`. You can change the `PORT` variable in `src/server.ts` or identify and terminate the conflicting process.
- **Confusion with `AgentRouteHandler`**: Remember that for our simple local Express example, we directly call `greeterAgent.handle()`. Do not confuse this local testing pattern with the `AgentRouteHandler`'s role in a full production deployment, where it directly handles the serverless `fetch` event. The local setup is a simplified mock.

Summary

You've successfully completed your first journey into the Flue Framework! Here’s a quick recap of the key takeaways:

- **Agent Harness Architecture:** Flue provides a comprehensive framework for building sophisticated AI agents, going beyond simple LLM API calls to manage state, integrate tools, and provide a secure execution environment.
- **Distinction from LLM SDKs:** We clarified that Flue is an orchestrator and runtime for agents, while LLM SDKs are primarily communication layers for interacting with LLMs. Flue uses LLM SDKs.
- **TypeScript-First Development:** Flue leverages TypeScript for robust, type-safe, and maintainable agent code, a critical advantage for production systems.
- **Core Agent Structure:** You learned to define a Flue agent by extending the `Agent` class and implementing its `handle` method, along with defining clear input and output interfaces.

- **Local Agent Testing:** You set up a local Express server to test your agent, understanding how to invoke its logic and receive structured responses.
- **`AgentRouteHandler` for Production:** You gained an initial understanding of `AgentRouteHandler`'s role as Flue's primary deployment mechanism for serverless environments like Cloudflare Workers.

In the upcoming chapters, we'll delve deeper into crucial aspects like stateful sessions, tool integration, and advanced deployment strategies, empowering you to build truly intelligent and interactive AI experiences.

References

- Flue — The Agent Harness Framework
- withastro/flue: The sandbox agent framework. - GitHub
- Node.js Official Website
- TypeScript Official Website
- Express.js Official Website

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 02

Setting Up Your Production-Ready Flue Development Environment

Building production-ready AI agents requires a solid foundation, starting with a properly configured development environment. In this chapter, we'll guide you through setting up everything you need to start crafting intelligent agents using the Flue Framework. We'll move from understanding Flue's unique architecture to getting your first project initialized and ready for action.

Why does a robust setup matter? A well-configured environment prevents countless headaches down the line, especially when dealing with TypeScript's strictness and the intricacies of agent frameworks. By the end of this chapter, you'll have a clean, ready-to-code workspace, prepared for the practical agent-building ahead.

Before we dive in, ensure you're comfortable with basic command-line operations and have a text editor or IDE (like VS Code) at hand. Familiarity with Node.js and TypeScript fundamentals will also be beneficial, as Flue heavily leverages these technologies.

Understanding the Flue Agent Harness Architecture

When we talk about building AI agents, it's easy to conflate "using an LLM" with "building an agent." Flue bridges this gap by providing an **agent harness** architecture. This is a critical distinction from mere Large Language Model (LLM) SDK wrappers.

What is an Agent Harness?

An agent harness, like Flue, is a structured environment designed to host, manage, and execute AI agents. It provides the surrounding infrastructure that allows an agent to go beyond just generating text, enabling it to act and interact.

Here's why an agent harness is essential:

- **Sandboxed Execution:** Agents, especially "coding agents," often need to run shell commands, interact with a filesystem, or execute arbitrary code. A harness provides a secure, isolated environment (a sandbox) for these operations. This prevents an agent's actions from affecting the host system, crucial for security and stability in production.
- **Stateful Sessions:** Unlike stateless API calls, real-world agents need memory. A harness maintains context across multiple interactions, enabling complex, multi-turn conversations or long-running tasks that require remembering previous steps or information.
- **Tool and Skill Integration:** Agents gain capabilities beyond pure language generation by calling external tools (e.g., APIs, databases, custom functions) and leveraging pre-defined "skills" (like Markdown rendering, specific data processing, or interacting with a knowledge base). The harness manages these integrations.
- **Standardized Interfaces:** Flue allows agents to be exposed over standard protocols like HTTP or WebSockets, making them easily consumable by other applications and services in a distributed system.

Flue vs. LLM SDK Wrappers

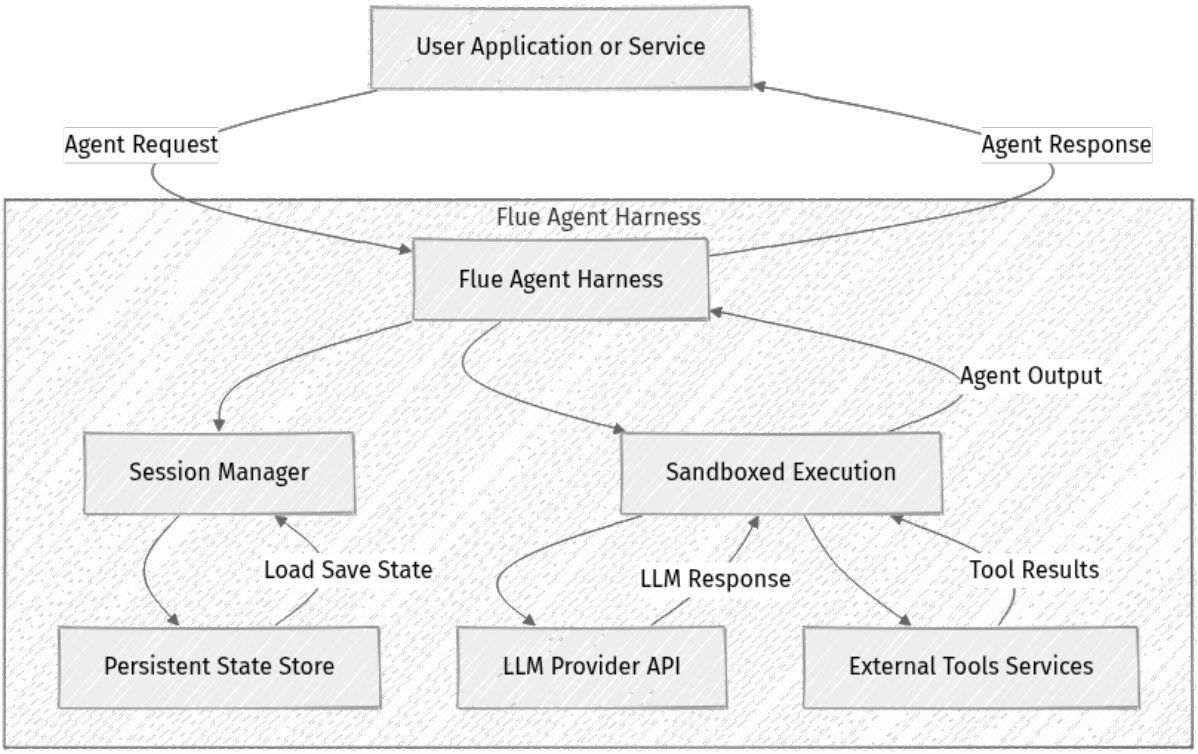
Let's clarify the difference with an analogy:

- **LLM SDK Wrapper:** Imagine this as a high-quality phone that lets you call an expert. You can ask questions, and the expert (LLM) provides answers. It handles the connection and communication, but the expert's knowledge is limited to what they know, and they can't do anything beyond talking.
- **Flue Agent Harness:** This is like building a fully-equipped robot that uses that phone to call the expert. The robot has a body (the harness), hands (tools/skills), memory (state), and a controlled workshop (sandbox) where it can follow instructions from the expert. The robot can not only get answers but also perform actions based on those answers, like writing code, searching the web, or managing files.

📌 **Key Idea:** Flue is not just a wrapper for an LLM API; it's an operational framework that provides the "body" and "environment" for intelligent, stateful, and tool-augmented agents to perform complex tasks in a controlled manner.

This distinction is vital for production-minded engineers. A basic LLM call can answer a question, but a Flue agent can act on that answer, persist its findings, and even self-correct within a controlled environment, making it suitable for real-world applications.

Let's visualize this agent harness architecture:



⚡ **Real-world insight:** This sandboxed execution and state management are what enable Flue to power complex coding agents that can, for example, write and execute code snippets, debug issues, or manage project files within their isolated environment. This capability is crucial for automating developer workflows or building sophisticated AI assistants.

Setting Up Your Development Environment

To start building with Flue, you'll need Node.js and npm (which comes bundled with Node.js), along with TypeScript for type-safe development.

Step 1: Install Node.js and npm

Flue is built on Node.js, so you'll need a recent stable version. As of **June 2026**, Node.js **v22.5.0** (or a similar LTS release from the 22.x or 24.x series) is an excellent choice for stability and modern features.

1. **Check your current Node.js version:** Open your terminal or command prompt and run:

```
    node -v
    npm -v
```

```
If you see versions older than 20.x, or if they're not installed, proceed to
the next step.
```

1. **Install Node.js:** The recommended way to install Node.js is using a version manager like `nvm` (Node Version Manager). This allows you to easily switch between different Node.js versions for various projects, which is a common practice in production environments.
    - **Install `nvm` (if you don't have it):** Follow the installation instructions on the official `nvm` GitHub page: https://github.com/nvm-sh/nvm
    - **Install a specific Node.js version using `nvm`:** Once `nvm` is installed, use these commands:

```
        nvm install 22.5.0
        nvm use 22.5.0
        nvm alias default 22.5.0 # Set this as the default for new shells
```

```
    These commands install Node.js `v22.5.0` and set it as the active version
for your current shell and future shells.

- **Verify installation:**
    Run the version check commands again:
```

```
        node -v
        npm -v
```

```
    You should now see `v22.5.0` (or your chosen version) and a compatible npm
version (usually `9.x` or `10.x`).
```

Step 2: Initialize Your Flue Project

Now that Node.js is ready, let's create a new project and install Flue.

1. **Create a new project directory:** Open your terminal and create a new folder for your project. Then navigate into it.

```
    mkdir my-first-flue-agent
    cd my-first-flue-agent
```

1. **Initialize a Node.js project:** This command creates a `package.json` file, which manages your project's dependencies and scripts.

```
    npm init -y
```

```
The `-y` flag answers "yes" to all prompts, creating a default `package.json`.
```

1. **Install Flue and TypeScript:** We'll install Flue, TypeScript, and `@types/node` for Node.js type definitions.

```
    npm install flue typescript @types/node --save-dev
```

```
Here's what each package does:
- `flue`: The core Flue framework.
- `typescript`: The TypeScript compiler itself.
- `@types/node`: Provides type definitions for Node.js APIs, essential for
TypeScript to understand global Node.js objects and functions.
```

1. **Configure TypeScript:** Create a `tsconfig.json` file in your project root. This file tells the TypeScript compiler how to compile your code, defining target versions, module resolution, and strictness rules.

```
    npx tsc --init --rootDir src --outDir dist --esModuleInterop --
resolveJsonModule --lib es2022 --module commonjs --target es2022 --strict
```

```
This command generates a `tsconfig.json` with sensible defaults for a Node.js
project. Let's briefly look at some key options:
- `rootDir src`: Specifies that your source code will reside in the `src/`
directory.
- `outDir dist`: Dictates that compiled JavaScript will be output to the
`dist/` directory.
- `esModuleInterop`: Allows you to use `import x from 'y'` syntax even for
```

```
CommonJS modules, improving compatibility.
- `resolveJsonModule`: Enables importing `.json` files directly into your
TypeScript code.
- `lib es2022`, `module commonjs`, `target es2022`: Configures the project to
use modern JavaScript features (ES2022) and CommonJS module system, which is
standard for Node.js.
- `strict`: Enables all strict type-checking options, a crucial best practice
for writing robust and maintainable code in TypeScript.

Open the generated `tsconfig.json` and ensure it looks similar to this (many
comments will be present, but these are the key uncommented lines):
```

```
    // tsconfig.json
    {
      "compilerOptions": {
        "target": "es2022",                                  /* Set the
JavaScript language version for emitted JavaScript and include compatible
library declarations. */
        "module": "commonjs",                                /* Specify what
module code is generated. */
        "rootDir": "./src",                                  /* Specify the
root folder within your source files. */
        "outDir": "./dist",                                  /* Specify an
output folder for all emitted files. */
        "esModuleInterop": true,                             /* Emit
additional JavaScript to ease support for importing CommonJS modules. This
enables 'allowSyntheticDefaultImports' for type compatibility. */
        "forceConsistentCasingInFileNames": true,            /* Ensure that
casing is correct in imports. */
        "strict": true,                                      /* Enable all
strict type-checking options. */
        "skipLibCheck": true,                                /* Skip type
checking all .d.ts files. */
        "resolveJsonModule": true,                           /* Enable
importing .json files. */
        "lib":
["es2022"]                                      /* Specify a list of library
files to be included in the compilation. */
      },
      "include": ["src/**/*.ts"],
      "exclude": ["node_modules", "**/*.spec.ts"]
    }
```

1. **Create your first source file:** Create a `src` directory and an `index.ts` file inside it. This will be your entry point.

```
    mkdir src
    touch src/index.ts
```

```
Open `src/index.ts` in your editor and add a simple `console.log` to confirm
everything is working.
```

```
// src/index.ts
console.log("Hello, Flue Agent!");
```

1. **Add build and start scripts:** Open your `package.json` file and add `build` and `start` scripts to compile and run your TypeScript code. This automates the development workflow.

```
// package.json
{
  "name": "my-first-flue-agent",
  "version": "1.0.0",
  "description": "A basic Flue agent project setup.",
  "main": "dist/index.js",
  "scripts": {
    "build": "tsc",
    "start": "node dist/index.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": ["flue", "agent", "typescript", "ai"],
  "author": "Your Name",
  "license": "ISC",
  "devDependencies": {
    "@types/node": "^20.14.2",
    "flue": "latest",
    "typescript": "^5.4.5"
  }
}
```

```
- `"flue": "latest"`: We specify `latest` for Flue's version, reflecting its current known state as of June 2026 without an explicit stable release number in the provided context. When a stable version is released, you would typically pin it (e.g., `"flue": "1.0.0"`).
- `"@types/node": "^20.14.2"` and `"typescript": "^5.4.5"`: These are the latest stable versions as of June 2026, ensuring compatibility with our chosen Node.js version.
```

1. **Test your setup:** Compile and run your simple script from the project root.

```
npm run build
npm start
```

```
You should see the output:
```

```
Hello, Flue Agent!
```

```
If you see this, congratulations! Your Flue development environment is
correctly set up and ready for agent development.
```

Mini-Challenge: Prepare for Your First Agent

Now that your environment is ready, let's take a tiny but crucial step towards building your first Flue agent.

Challenge: Inside your `src` directory, create a new file named `myAgent.ts`. In this file, import the `Agent` class from `flue` and declare a simple (empty for now) class that extends `Agent`. This will be the foundational skeleton for our first intelligent agent in the next chapter.

Hint: Remember that Flue is a TypeScript-first framework. You'll need an `import` statement to bring in the `Agent` class and a class declaration using the `extends` keyword.

> 💡 **STUCK? CLICK FOR A HINT!**
>
> Your `myAgent.ts` file might start something like this:
>
> ```
> // src/myAgent.ts
> import { Agent } from 'flue';
> class MyFirstAgent extends Agent {
> // Agent logic will go here
> }
>
>
> // You might export it later to use it in other parts of your application
> export default MyFirstAgent;
> ```

What to observe/learn: This challenge reinforces the TypeScript-first approach of Flue and introduces you to the basic structural pattern of a Flue agent class. It's a small step, but it confirms your understanding of module imports and class extensions within your new Flue project, which are fundamental building blocks.

Common Pitfalls & Troubleshooting

Setting up development environments can sometimes lead to unexpected issues. Here are a few common mistakes and how to address them, ensuring you can quickly get back on track.

- **Node.js Version Mismatch:**
  - **Symptom:** Errors about unsupported syntax, missing `npm` commands, or features not found.
  - **Solution:** Your Node.js version might be too old or not the one you intended to use. Use `nvm use <version>` to switch to the correct version, or `nvm install <version>` to get a newer one. Always verify your active version with `node -v` and `npm -v`.
- **TypeScript Compilation Errors:**
  - **Symptom:** `Cannot find module 'flue'` or `Cannot find name 'Agent'`.
  - **Solution:** This usually means `flue` wasn't installed correctly, or your `tsconfig.json` isn't configured to include `node_modules` or your `src` directory. Double-check that `npm install flue` ran without errors and review your `tsconfig.json`'s `include` and `exclude` paths.
  - **Symptom:** Type errors related to `console.log` or other Node.js globals (e.g., `Error: Cannot find name 'console'`).
  - **Solution:** Ensure `@types/node` is installed (`npm install @types/node --save-dev`) and your `tsconfig.json` includes `lib: ["es2022"]` (or a similar modern `lib` array) which provides global type definitions for Node.js environments.

- **`npm run build` fails or `npm start` can't find `dist/index.js`:**
  - **Symptom:** The build script fails, or the `start` script reports that `dist/index.js` does not exist.
  - **Solution:**
    1. Verify your `package.json` scripts: `"build": "tsc"` and `"start": "node dist/index.js"`.
    2. Ensure your `tsconfig.json`'s `outDir` is set to `./dist` and `rootDir` is set to `./src`.
    3. Confirm your entry source file is actually located at `src/index.ts`.
- **`npx` command not found:**
  - **Symptom:** The terminal reports `npx: command not found` when trying to run `npx tsc --init`.
  - **Solution:** `npx` comes bundled with npm `v5.2.0` and higher. If you get this error, you likely have an old npm version. Update npm globally by running `npm install -g npm@latest` or update your Node.js installation.

🧠 **Important:** Always read the error messages carefully. They often point directly to the problem, whether it's a missing file, an incorrect configuration, or a syntax error. Don't be afraid to search online for specific error messages, as they are often common and have well-documented solutions.

Summary

In this chapter, you've successfully laid the groundwork for building sophisticated AI agents with the Flue Framework. This foundational setup is critical for developing robust and production-ready applications.

Here are the key takeaways:

- **Flue is an Agent Harness, Not Just an LLM Wrapper:** It provides a comprehensive environment for sandboxed execution, state management, and tool integration, empowering agents to act intelligently and persistently.
- **Node.js and TypeScript are Core:** Your development environment relies on a modern Node.js installation (e.g., v22.5.0 as of June 2026) and a well-configured TypeScript setup (e.g., v5.4.5) for type safety and maintainability.

- **Project Setup is Incremental:** You learned to initialize a Node.js project, install Flue and its dependencies, configure TypeScript, and create basic `build` and `start` scripts.
- **A Solid Foundation:** Your environment is now meticulously prepared for the exciting task of defining and implementing your first intelligent agent, ensuring a smooth development experience.

In the next chapter, we'll dive into the core concepts of creating a Flue agent, defining its capabilities, and making it respond to prompts and interact with its environment. Get ready to bring your agent to life!

References

- Node.js Official Website
- TypeScript Handbook
- nvm (Node Version Manager) GitHub Repository
- Flue — The Agent Harness Framework Official Website
- withastro/flue GitHub Repository

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 03

Building Your First Flue Agent: Core Concepts and Tool Integration

In the previous chapter, we established a mental model for AI agents and understood why a specialized framework like Flue is essential for their reliable deployment. Now, it's time to transition from theory to practice and construct our first functional agent. This chapter will walk you through the core architecture of a Flue agent, demonstrate how to integrate simple tools, and guide you in structuring your agent using TypeScript.

Our goal is not just to build a working agent, but to truly understand why Flue's design principles—like the agent harness and state management—are critical for building robust, production-ready AI systems. You'll gain hands-on experience by creating a simple agent that can respond to inputs and interact with a custom tool, setting the stage for more complex agent development.

To make the most of this chapter, please ensure you have Node.js (version 20.x or later, as of June 2026) and TypeScript installed. Familiarity with basic LLM coding paradigms (like those found in Claude Code, Codex, or OpenAI's APIs) will also be beneficial.

The Agent Harness: Beyond Basic LLM Wrappers

When you hear "AI agent," your first thought might be a simple function that calls an LLM API. While that's a component, Flue introduces the concept of an **agent harness** – a comprehensive runtime environment that elevates agents far beyond mere API wrappers.

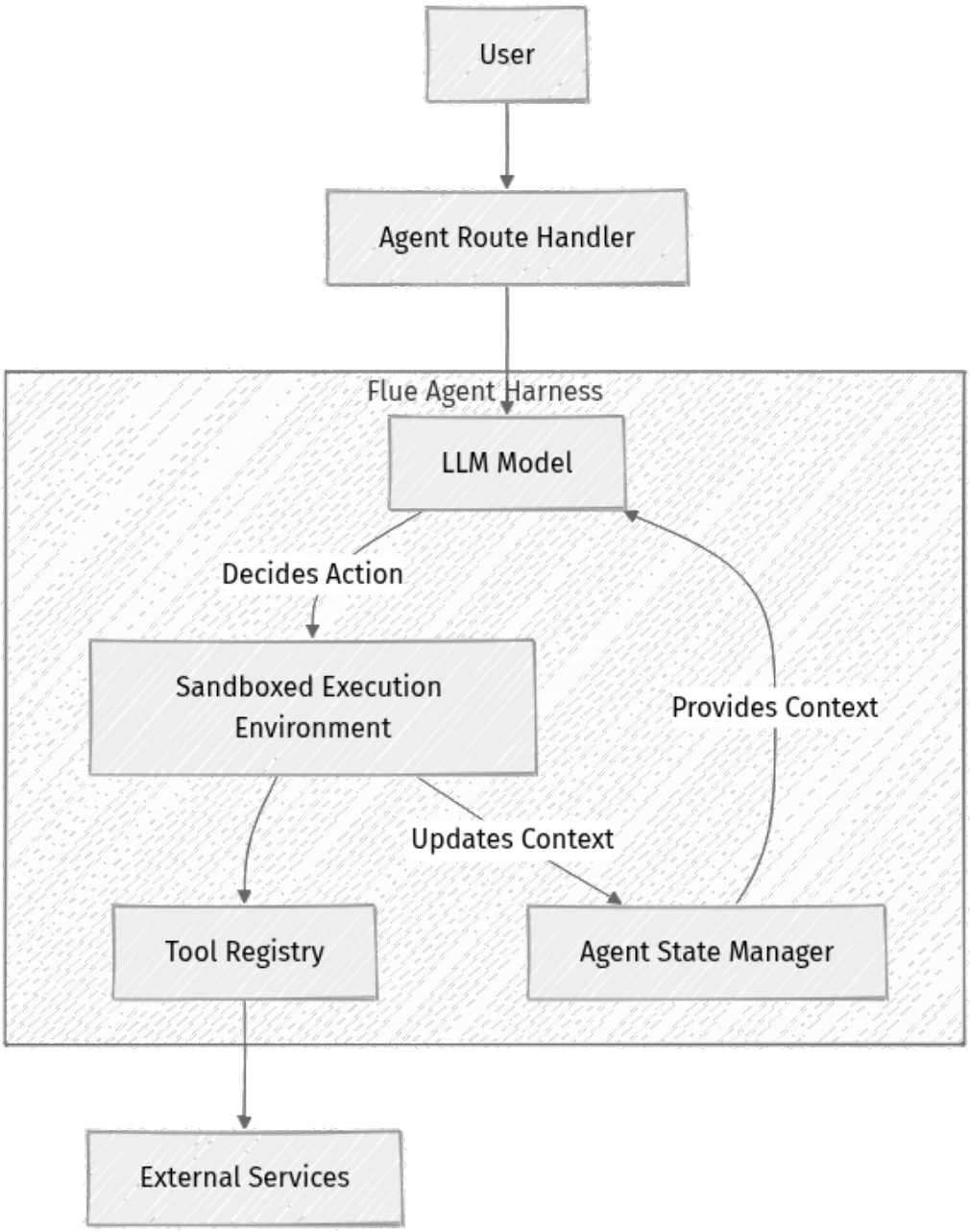
📌 Key Idea: A Flue agent is more than just an LLM. It's an intelligent process capable of maintaining state, executing code in a sandboxed environment, and interacting with its surroundings through well-defined tools. The harness provides this structured, controlled execution context.

Why Does the Agent Harness Matter?

Developing sophisticated AI agents directly with LLM SDKs often leaves developers grappling with critical infrastructure challenges:

- **State Management:** How does an agent remember previous interactions? Without a harness, you're left to implement complex, error-prone session management yourself.
- **Secure Execution:** If an agent needs to run code or access external systems, how do you ensure it operates safely and within defined boundaries? The harness provides the foundation for sandboxed execution.
- **Tool Orchestration:** How does the LLM intelligently select and use external functions or data sources? The harness provides a standardized mechanism for tool definition and invocation.
- **Deployment and Scalability:** How do you deploy and manage multiple, potentially long-running, stateful agents reliably? The harness abstracts away much of this complexity.

Flue's agent harness addresses these challenges head-on by providing a structured, opinionated environment that facilitates secure, stateful, and tool-augmented agent behavior.



This diagram illustrates the core components within the `Flue Agent Harness` and how they interact. The `Agent Route Handler` acts as the entry point, directing user requests to the LLM, which then leverages the `Agent State Manager`, `Sandboxed Execution Environment`, and `Tool Registry` to process requests and interact with external systems.

Agents as Stateful Entities

Consider an AI assistant that remembers your name, preferences, or the context of a multi-step task. This capability hinges on **statefulness**. Flue agents are inherently designed to maintain state across multiple interactions, enabling:

- **Multi-turn Conversations:** The agent seamlessly retains conversational context, making interactions feel natural and continuous.
- **Complex Workflows:** An agent can track its progress through intricate processes, such as code generation or data analysis, remembering intermediate results.
- **Personalization:** Agents can remember user-specific data or historical interactions, tailoring responses and actions over time.

This state is securely managed by the Flue harness, abstracting away the complexities of persistence and retrieval, allowing you to focus on agent logic.

Tools and Skills: Extending Agent Capabilities

An agent that can only generate text is limited. Real-world AI agents must act. Flue facilitates this through **Tools** (sometimes referred to as skills), which are specific, well-defined functions or capabilities that you expose to your agent.

⚡ Real-world insight: Think of tools as the agent's "API clients" or "execution environment." They provide the agent with the means to interact with the external world—fetching data, calling APIs, sending emails, or even executing code in a controlled, sandboxed manner.

Examples of tools that Flue agents can integrate include:

- `MarkdownGenerator`: For formatting output consistently.
- `FileEditor`: To read from or write to a sandboxed filesystem, crucial for coding agents.
- `Calculator`: To perform precise mathematical computations.
- `APICaller`: To make HTTP requests to integrate with any external service.

Each tool is defined with a clear name, description, and an `inputSchema` (using JSON Schema), which instructs the LLM on how and when to use the tool, including the arguments it expects. This clear interface is vital for both agent reliability and security.

Structuring Agents in TypeScript

Flue embraces a TypeScript-first approach. This strong typing and structured environment bring significant benefits to agent development:

- **Type Safety:** Catch errors during compilation, not at runtime, leading to more stable agents.
- **Enhanced Readability:** Clear interfaces and types make code easier to understand, maintain, and onboard new team members.
- **Improved Developer Experience:** Benefit from IDE autocompletion, robust refactoring tools, and clear documentation generated from type definitions.

You will define your agent's behavior, its available tools, and how it processes inputs and generates outputs using TypeScript classes and interfaces, ensuring a consistent and robust development workflow.

Step-by-Step Implementation: Building a Simple Echo Agent with a Tool

Let's put these concepts into practice by building a basic Flue agent. This agent will echo messages and demonstrate the integration of a simple "logger" tool.

1. Project Setup

If you're continuing from a previous setup, ensure your `my-first-flue-agent` directory is ready. Otherwise, let's create a new Node.js project and install the necessary Flue packages, along with TypeScript and Express:

```
mkdir my-first-flue-agent
cd my-first-flue-agent
npm init -y
npm install @flue/core @flue/express-handler @flue/core-llms-openai
typescript @types/node express @types/express dotenv
npx tsc --init # Initialize TypeScript configuration
```

Next, let's refine our `tsconfig.json` for a modern Node.js environment. Open `tsconfig.json` and update it to match the following:

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "es2022",                 // Target modern ECMAScript features
    "module": "commonjs",               // Use CommonJS for Node.js modules
    "rootDir": "./src",                 // Source files are in the 'src'
directory
```

```
    "outDir": "./dist",               // Compiled JavaScript goes into 'dist'
    "esModuleInterop": true,          // Allow default imports from modules
with no default export
    "forceConsistentCasingInFileNames": true, // Enforce consistent casing in
file names
    "strict": true,                   // Enable all strict type-checking
options
    "skipLibCheck": true,             // Skip type checking of declaration
files
    "lib": ["es2022"],                // Include ES2022 standard library
features
    "moduleResolution": "node"        // Node.js-style module resolution
  },
  "include": ["src/**/*.ts"],         // Include all .ts files in src
  "exclude": ["node_modules"]         // Exclude node_modules from compilation
}
```

This configuration ensures our TypeScript code compiles correctly for Node.js and benefits from strict type checking.

Finally, create a `src` directory where all our agent code will reside:

```
mkdir src
```

2. Defining a Simple Tool: The Logger

Our first tool will be a `Logger`. This tool will simply print a message to the console, simulating an agent taking an action that has an observable side effect.

Create the file `src/tools/Logger.ts`:

```
// src/tools/Logger.ts
import { Tool } from '@flue/core';

/**
 * A simple Logger tool for agents to output messages to the console.
 * This simulates an agent taking an action that has a side effect (logging an
event).
 */
export class Logger extends Tool {
  // Every tool needs a unique name. The LLM uses this to identify the tool.
  public readonly name = 'Logger';
  // A clear description helps the LLM understand when and why to use this
tool.
  public readonly description = 'Logs a message to the console for debugging
or event tracking.';

  /**
   * The inputSchema defines the structure of arguments this tool expects.
   * This is crucial for the LLM to know how to call the tool correctly.
   * We use 'as const' to ensure TypeScript infers the most specific type.
   */
  public readonly inputSchema = {
    type: 'object',
    properties: {
      message: {
```

```
        type: 'string',
        description: 'The message content to log.',
      },
    },
    required: ['message'], // The 'message' property is mandatory for this
tool.
  } as const;

  /**
   * The execute method contains the core logic of the tool.
   * It receives validated input from the LLM and performs its action.
   * @param input The validated input object, matching inputSchema.
   * @returns A promise resolving to a string indicating the result of the
tool's action.
   */
  public async execute(input: { message: string }): Promise<string> {
    // Perform the side effect: logging the message to the server console.
    console.log(`[AGENT LOG]: ${input.message}`);
    // Return a message that the agent can use in its response to the user.
    return `Message successfully logged: "${input.message}"`;
  }
}
```

Explanation of the `Logger` Tool:

- `import { Tool } from '@flue/core';`: All tools in Flue extend the base `Tool` class, providing a consistent interface.
- `public readonly name = 'Logger';`: This is how the LLM will refer to this tool. It must be unique within an agent.
- `public readonly description = '...';`: A well-written description helps the LLM understand the tool's purpose and when to invoke it.
- `public readonly inputSchema = { ... } as const;`: This JSON Schema-like object is vital. It formally defines the arguments the `execute` method expects. The LLM uses this schema to generate appropriate tool calls. `as const` ensures TypeScript infers the most specific type for compile-time safety.
- `public async execute(input: { message: string }): Promise<string> { ... }`: This method contains the actual logic. When the LLM decides to use the `Logger` tool, it will provide an `input` object that conforms to the `inputSchema`. Our `Logger` simply prints this message to the console and returns a confirmation string.

3. Creating Your First Flue Agent

Now, let's create our `EchoAgent`. This agent will be able to simply echo back user messages or, if prompted correctly, use our new `Logger` tool.

Create the file `src/agents/EchoAgent.ts`:

```
// src/agents/EchoAgent.ts
import { Agent, AgentContext, AgentStep, LLM } from '@flue/core';
import { Logger } from '../tools/Logger'; // Import our custom Logger tool

/**
 * An EchoAgent demonstrates basic agent functionality:
 * - Responding to user input directly.
 * - Simulating tool usage based on specific input patterns.
 */
export class EchoAgent extends Agent {
  // A unique name for our agent.
  public readonly name = 'EchoAgent';
  // A description of what this agent does.
  public readonly description = 'An agent that echoes messages and can log
them using a Logger tool.';

  /**
   * The constructor is where we initialize the agent and register its
available tools.
   * @param llm The Language Model instance this agent will use for reasoning.
   */
  constructor(llm: LLM) {
    super(llm); // Call the base Agent class constructor with the LLM.
    // Register our custom Logger tool, making it available for the agent to
use.
    this.registerTool(new Logger());
  }

  /**
   * The core logic for how the agent processes an incoming input.
   * In a real-world scenario, the LLM would decide on actions.
   * Here, we use a simple 'if' condition to simulate tool invocation.
   * @param context The current agent context, including user input and
session state.
   * @returns A promise resolving to an AgentStep, which defines the agent's
next action or response.
   */
  public async handle(context: AgentContext): Promise<AgentStep> {
    const { input } = context; // Extract the user's input from the context.

    // For this introductory agent, we'll use a simple heuristic to trigger
the tool.
    // In more advanced Flue agents, the LLM itself would analyze the input,
    // consult its registered tools (using their names and descriptions),
    // and decide if a tool call is appropriate.
    if (input.startsWith('log:')) {
      const messageToLog = input.substring(4).trim(); // Extract the message
after "log:"
      console.log(`EchoAgent received request to log: "${messageToLog}"`);

      // Retrieve the registered Logger tool instance.
      const loggerTool = this.getTool('Logger') as Logger;

      if (loggerTool) {
        // Execute the tool with the extracted message.
        const toolOutput = await loggerTool.execute({ message: messageToLog })
;
        // Return a response step, informing the user about the tool's action.
        return {
          type: 'response',
          content: `I've used the Logger tool for you. Tool said: $
```

```
{toolOutput}`,
        };
      } else {
        // Fallback if the tool isn't found (shouldn't happen if registered
correctly).
        return {
          type: 'response',
          content: `Error: Logger tool was requested but not found.`,
        };
      }
    } else {
      // If the input doesn't trigger a tool, simply echo the user's message.
      return {
        type: 'response',
        content: `You said: "${input}"`,
      };
    }
  }
}
```

Explanation of the `EchoAgent`:

- `import { Agent, AgentContext, AgentStep, LLM } from '@flue/core';`: These are core Flue types. `Agent` is the base class for all agents. `AgentContext` holds the current interaction's data, and `AgentStep` defines what the agent does next.
- `constructor(llm: LLM)`: Every Flue agent needs an LLM to power its reasoning. We pass an `LLM` instance to the base `Agent` constructor.
- `this.registerTool(new Logger());`: This line is crucial! It makes our `Logger` tool available to this specific `EchoAgent`. Without this, the agent wouldn't know about the tool.
- `public async handle(context: AgentContext): Promise<AgentStep>`: This is the heart of your agent's logic. It's called whenever the agent receives a new input. The `context` object provides the `input` (user's message) and access to the agent's `state`.
- **Simulated Tool Use vs. LLM Reasoning:** For simplicity in this introductory chapter, we're using a direct `if (input.startsWith('log:'))` condition to trigger our `Logger` tool.
    - **In a real, advanced Flue agent,** the LLM itself would receive the user's `input`, analyze it, consult the descriptions and schemas of all registered tools, and then decide which tool (if any) to call, and with what arguments. This decision-making process is a key differentiator of the agent harness. Our current approach just helps us see the tool integration in action without complex LLM prompting for tool selection.

- `return { type: 'response', content: '...' };`: An agent's `handle` method must always return an `AgentStep`. For now, we're returning a `response` step, which means the agent is outputting text back to the user. Other step types include `toolCall` (to instruct the LLM to call a tool) or `error`.

4. Setting up the Express Server with `AgentRouteHandler`

To make our `EchoAgent` accessible via HTTP, we'll use Express.js and Flue's `AgentRouteHandler`, which simplifies exposing agents as API endpoints.

Create the file `src/index.ts`:

```
// src/index.ts
import express from 'express';
import { AgentRouteHandler } from '@flue/express-handler'; // Flue's Express
integration
import { OpenAI } from '@flue/core-llms-openai'; // Example LLM provider
(OpenAI)
import { EchoAgent } from './agents/EchoAgent'; // Our custom agent
import dotenv from 'dotenv'; // For loading environment variables

dotenv.config(); // Load environment variables from the .env file

const app = express();
app.use(express.json()); // Enable Express to parse JSON request bodies

const port = process.env.PORT || 3000; // Define the port for our server

// 🧠 Important: An LLM API key is essential for your agent to function.
// Flue supports various LLM providers (OpenAI, Claude, etc.).
// For this example, we're using OpenAI. Ensure OPENAI_API_KEY is set
// in your .env file or environment variables.
if (!process.env.OPENAI_API_KEY) {

console.error("Error: OPENAI_API_KEY environment variable is not set. Please
add it to your .env file.");
  process.exit(1); // Exit if the API key is missing.
}

// Initialize our chosen LLM. Flue's modular design allows you to swap LLMs
easily.
// As of June 2026, ensure your LLM provider and API key are compatible.
const llm = new OpenAI({ apiKey: process.env.OPENAI_API_KEY });

// Create an instance of our EchoAgent, passing the initialized LLM.
const echoAgent = new EchoAgent(llm);

// Create an AgentRouteHandler for our agent. This adapts the agent
// to work seamlessly with Express routes.
const agentHandler = new AgentRouteHandler(echoAgent);

// Define an HTTP POST endpoint for our agent.
// When a POST request hits '/agent/echo', the agentHandler will process it.
app.post('/agent/echo', agentHandler.handle);

// Add a basic health check endpoint for monitoring purposes.
```

```
app.get('/health', (req, res) => {
  res.status(200).send('Agent server is running and healthy!');
});

// Start the Express server.
app.listen(port, () => {
  console.log(`EchoAgent server listening at http://localhost:${port}`);
  console.log(`\nTo test, open another terminal and use curl:`);
  console.log(`  - Simple echo: curl -X POST -H "Content-Type: application/
json" -d '{"input": "Hello Flue!"}' http://localhost:${port}/agent/echo`);
  console.log(`  - Use Logger tool: curl -X POST -H "Content-Type:
application/json" -d '{"input": "log: This is a test message for the agent."}'
http://localhost:${port}/agent/echo`);
});
```

Explanation of the Express Server:

- `import { AgentRouteHandler } from '@flue/express-handler';`: This powerful helper class from Flue simplifies integrating your agent with Express. It handles the parsing of incoming requests, forwarding them to your agent, and formatting the agent's response back to the client.
- `import { OpenAI } from '@flue/core-llms-openai';`: We're importing the `OpenAI` LLM provider. Flue is designed to be LLM-agnostic, so you can swap this with other providers like `Claude` or custom LLMs as needed.
- `dotenv.config();`: This line loads environment variables from a `.env` file in your project root, which is the standard way to manage sensitive information like API keys securely.
- `const llm = new OpenAI({ apiKey: process.env.OPENAI_API_KEY });`: Here, we instantiate our LLM. **It's critical that your `OPENAI_API_KEY` environment variable is set.**
- `const agentHandler = new AgentRouteHandler(echoAgent);`: We create an instance of the `AgentRouteHandler`, passing our `echoAgent` to it. This handler now knows how to interact with our agent.
- `app.post('/agent/echo', agentHandler.handle);`: This line registers a POST route. Any incoming POST request to `/agent/echo` will be processed by `agentHandler.handle`, which in turn invokes our `EchoAgent`.

5. Running Your Agent

Before running, we need to provide our OpenAI API key.

Create a new file named `.env` in the root of your `my-first-flue-agent` project (next to `package.json`):

```
# .env
OPENAI_API_KEY=your_openai_api_key_here
```

Replace `your_openai_api_key_here` with your actual OpenAI API key. Never commit your `.env` file to version control.

Now, compile your TypeScript code and run the server:

```
npx tsc # Compiles TypeScript files from src/ to dist/
node dist/index.js # Runs the compiled JavaScript server
```

You should see output indicating your server is running and providing `curl` commands for testing:

```
EchoAgent server listening at http://localhost:3000

To test, open another terminal and use curl:
  - Simple echo: curl -X POST -H "Content-Type: application/json" -d
'{"input": "Hello Flue!"}' http://localhost:3000/agent/echo
  - Use Logger tool: curl -X POST -H "Content-Type: application/json" -d
'{"input": "log: This is a test message for the agent."}' http://
localhost:3000/agent/echo
```

Open a new terminal window (keep the server running in the first one) and try the `curl` commands:

Test 1: Simple Echo Response

```
curl -X POST -H "Content-Type: application/json" -d '{"input": "Hello
Flue!"}' http://localhost:3000/agent/echo
```

Expected output in your curl terminal:

```
{"output":"You said: \"Hello Flue!\""}
```

Test 2: Using the Logger Tool

```
curl -X POST -H "Content-Type: application/json" -d
'{"input": "log: This is a test message for the agent."}' http://
localhost:3000/agent/echo
```

Expected output in your **server terminal**:

```
EchoAgent received request to log: "This is a test message for the agent."
[AGENT LOG]: This is a test message for the agent.
```

And in your **curl terminal**:

```
{"output":"I've used the Logger tool for you. Tool said: Message successfully logged: \"This is a test message for the agent.\""}
```

Congratulations! You've successfully built, configured, and run your first Flue agent with integrated tools. You've witnessed how Flue orchestrates agent responses and tool interactions.

Mini-Challenge: Extend Your Agent with a `Greeter` Tool

Now that you've built a basic agent and integrated a `Logger` tool, it's your turn to expand its capabilities.

Challenge:

1. **Create a New Tool:** In `src/tools/`, create a new TypeScript file named `Greeter.ts`.
2. **Define `Greeter`:**
    - This tool should extend `@flue/core.Tool`.
    - Give it a `name` (e.g., `'Greeter'`) and a `description` (e.g., `'Greets a person by name.'`).
    - Its `inputSchema` should define a single `name` property of type `string`, which is `required`.
    - The `execute` method should accept an `input` object with a `name` and return a friendly greeting string, such as `"Hello, [name]! Nice to meet you."`.
3. **Register with `EchoAgent`:** Modify `src/agents/EchoAgent.ts` to register an instance of your new `Greeter` tool in its constructor.
4. **Modify Agent Logic:** Update `EchoAgent`'s `handle` method. If the `input` starts with `"greet:"`, extract the name (e.g., `greet: Alice` should extract `Alice`), and use the `Greeter` tool to generate a personalized greeting.
5. **Test Your Agent:** Restart your server and use `curl` to test your agent's new greeting capability. For example:

```
    curl -X POST -H "Content-Type: application/json" -d '{"input": "greet:
Alice"}' http://localhost:3000/agent/echo
```

Hint: Follow the existing structure of the `Logger` tool and how `EchoAgent`'s `handle` method currently processes the `"log:"` prefix. Remember to import your new tool into `EchoAgent.ts`.

Common Pitfalls & Troubleshooting

Even with a simple agent, issues can arise. Here are some common problems and how to debug them:

- **Missing API Key:** This is the most frequent issue.
  - **Symptom:** Server crashes on startup with an error like "OPENAI_API_KEY environment variable is not set."
  - **Fix:** Double-check your `.env` file. Ensure `OPENAI_API_KEY` (or the key for your chosen LLM) is correctly spelled, has a valid value, and is in the project root. Remember to restart your server after modifying `.env`.
- **Tool Not Registered:**
  - **Symptom:** Your agent's logic attempts to use a tool, but it's never found (`this.getTool('ToolName')` returns `undefined`).
  - **Fix:** Verify that you've called `this.registerTool(new YourTool())` in your agent's constructor.
- **TypeScript Compilation Errors:**
  - **Symptom:** `npx tsc` fails with type errors or your IDE shows red squiggly lines.
  - **Fix:** Carefully review the error messages. Ensure your `tsconfig.json` is correctly configured, especially `rootDir`, `outDir`, and `moduleResolution`. Check for typos in type definitions.

- **Incorrect `inputSchema` (for advanced LLM-driven tool use):**
    - **Symptom:** In a more complex agent where the LLM decides to call tools, the LLM might fail to invoke your tool or provide incorrect arguments.
    - **Fix:** While not fully demonstrated in this chapter (due to our simulated tool call), ensure your tool's `inputSchema` is precise and clearly describes the expected arguments. Ambiguous descriptions or missing `required` fields can confuse the LLM.
- **Port Conflicts:**
    - **Symptom:** Your server fails to start with an error like "listen EADDRINUSE: address already in use :::3000."
    - **Fix:** Another process is already using port 3000. You can change the `PORT` environment variable in your `.env` file (e.g., `PORT=3001`) or identify and kill the conflicting process.

⚠️ What can go wrong: In real-world Flue agents, the LLM's ability to correctly use tools hinges on clear, concise tool descriptions and accurate `inputSchema` definitions. If your agent struggles with tool use, first examine how you've presented the tool's purpose and expected inputs.

Summary

In this chapter, you've taken a crucial step in understanding and building production-ready AI agents with the Flue Framework.

Here are the key takeaways:

- **Agent Harness Architecture:** You learned that Flue provides a robust "agent harness" that manages state, sandboxed execution capabilities, and tool orchestration, setting it apart from basic LLM API wrappers.
- **Stateful Agents:** Flue agents are designed to be inherently stateful, enabling them to maintain context across complex, multi-turn interactions.
- **Tools and Skills Integration:** You gained hands-on experience defining and integrating custom tools (like our `Logger`), which empower your agents to perform actions and interact with the external world.
- **TypeScript-First Approach:** Flue leverages TypeScript for strong typing, improved readability, and a more robust developer experience when structuring agents and their workflows.

- **`AgentRouteHandler` for Deployment:** You saw how `AgentRouteHandler` simplifies exposing your Flue agents as accessible HTTP endpoints, ready for integration into your applications.

You now have a foundational understanding of how to construct a Flue agent and integrate simple tools, laying the groundwork for more sophisticated AI applications. In the next chapter, we'll delve deeper into advanced tool integration, explore more nuanced state management strategies, and begin to build truly intelligent and interactive coding agents.

References

- Flue — The Agent Harness Framework
- withastro/flue: The sandbox agent framework - GitHub
- Node.js Official Documentation
- TypeScript Official Documentation
- Express Official Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 04

Mastering Stateful Sessions: Enabling Context-Aware Interactions

Imagine interacting with an AI agent that remembers nothing from your previous statements. Each turn is a fresh start, making complex conversations or multi-step tasks frustratingly inefficient. This chapter dives into a critical aspect of building truly intelligent agents: **stateful sessions**.

You'll learn how Flue Framework empowers your agents with memory, enabling them to retain context, conversation history, and custom data across multiple interactions. This capability is what transforms a simple prompt-response system into a dynamic, engaging, and truly helpful assistant. We'll move beyond stateless API calls to build agents that understand continuity, a cornerstone for any production-ready AI product.

Before we begin, ensure you're comfortable with basic Flue agent creation, as covered in the previous chapters. We'll be building upon that foundation to introduce robust state management.

The Indispensable Need for Agent Memory

In the realm of AI, particularly with large language models (LLMs), a fundamental challenge is their inherently stateless nature. Each API call to an LLM is typically an isolated event; it doesn't inherently remember past interactions. For many real-world applications, this is a significant limitation.

Think about a human assistant. If you tell them, "Please schedule a meeting," and then later say, "Make it for tomorrow at 2 PM," they understand the "it" refers to the meeting you just mentioned. They maintain context. A stateless LLM agent, however, might treat the second command as entirely new, asking "What are we scheduling?"

Why Stateless LLMs Fall Short for Complex Interactions

- **Lack of Context:** Without memory, agents cannot follow multi-turn conversations, understand pronouns, or build upon previous information. The user experience suffers when every interaction feels like the first.

- **Repetitive Information:** Users would constantly need to repeat details, leading to a poor user experience and increased cognitive load.
- **Inefficient Task Completion:** Complex tasks that require several steps or user inputs become impossible to manage within a single, isolated interaction. Imagine booking a flight without remembering previous selections.
- **No Personalization:** Agents cannot learn user preferences, adapt to individual styles, or maintain a personalized profile over time.

This is precisely where Flue's agent harness architecture, with its focus on stateful sessions, provides a powerful solution. Flue orchestrates the lifecycle of these interactions, providing a memory layer around the core LLM calls.

Flue's Approach to Stateful Sessions

Flue introduces the concept of an `AgentSession` to address the stateless nature of underlying LLMs. An `AgentSession` acts as a dedicated container for an ongoing interaction with an agent, preserving all the necessary context from one turn to the next.

📌 **Key Idea:** An `AgentSession` is like a scratchpad and memory bank for a single, continuous interaction with your agent. It’s the essential component for building conversational AI.

What an `AgentSession` Holds

The `AgentSession` object, passed into your agent's handler function, typically provides:

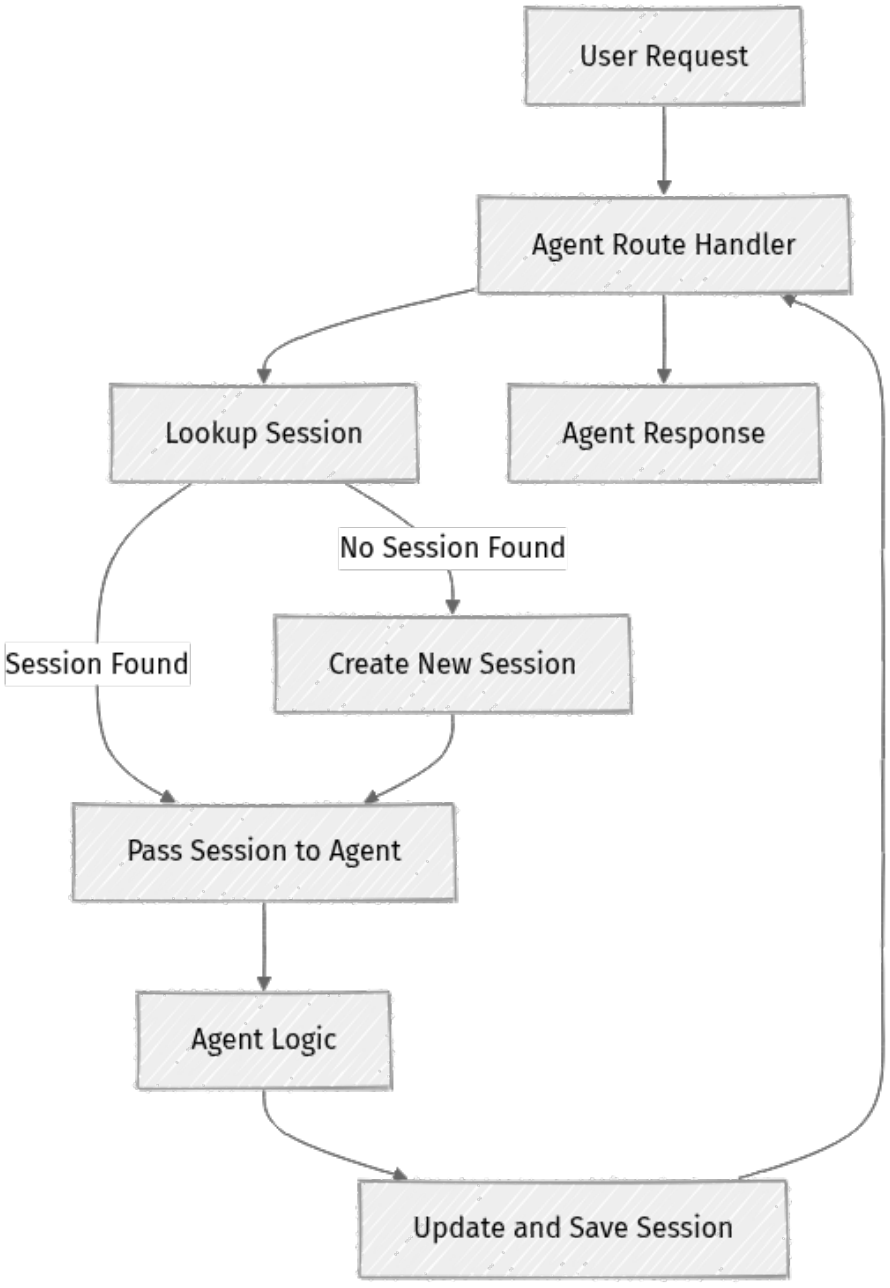
1. **`session.data`**: A generic key-value store (`Record<string, unknown>`) where you can store any custom data your agent needs to remember. This is perfect for user preferences, task progress, temporary variables, or any other application-specific state.
2. **`session.history`**: An array that stores the chronological sequence of messages exchanged between the user and the agent. This conversation history is crucial for providing context to the LLM in subsequent turns.
3. **`session.id`**: A unique identifier for the session, allowing Flue (and its underlying storage mechanism) to retrieve and persist the correct session state.

By leveraging these properties, your agent gains the ability to:

- **Maintain Conversation Flow:** Pass `session.history` to the LLM to ensure it understands the ongoing dialogue.
- **Track Task Progress:** Store variables in `session.data` to remember where a user is in a multi-step process.
- **Personalize Interactions:** Save user preferences or past choices specific to that session.

How Session State Flows in Flue

Let's visualize how a user request interacts with a Flue agent and its session state.



This diagram illustrates that every interaction with a stateful Flue agent involves retrieving and then saving the session state. This seamless flow, managed by the `AgentRouteHandler`, makes building context-aware agents much simpler for developers.

Distinguishing Session State from Persistent Agent State

It's important to differentiate between two types of "state" an agent might manage:

1. **Session State (Focus of this Chapter):** This is temporary, scoped to a single, continuous interaction or conversation. It's typically short-lived, lasting only as long as the user actively engages with the agent. `AgentSession` primarily manages this. Think of it as the agent's short-term memory for a particular user's current task.
2. **Persistent Agent State:** This refers to long-term, global data that an agent might need across all sessions or even across deployments. Examples include a coding agent's accumulated knowledge base, a user's long-term profile, or configuration settings. While Flue provides mechanisms to integrate with external persistent stores (like databases or key-value stores), the `AgentSession` itself is designed for the more ephemeral, interaction-specific context. This is the agent's long-term memory or knowledge base.

For this chapter, we'll concentrate on mastering the `AgentSession` for immediate, context-aware interactions.

Implementing a Stateful Flue Agent

Let's put theory into practice. We'll start by modifying a basic agent to remember the user's name across multiple turns.

Setting Up a Basic Stateful Agent

First, create a new Flue project or use an existing one. We'll define a simple agent that asks for the user's name and remembers it for subsequent interactions.

Create a file named `src/agents/greetingAgent.ts`:

```
// src/agents/greetingAgent.ts
import { AgentFunction, AgentSession } from '@flue/core';

// Define an interface for our custom session data for type safety
interface GreetingSessionData {
  userName?: string; // userName is optional, as it might not be set initially
}
```

```
export const greetingAgent: AgentFunction = async (
  prompt: string,
  session: AgentSession<GreetingSessionData>, // Type our session data for
strong typing
  context: Record<string, unknown> // For future use, not critical here
) => {
  // 🧠 Important: Always check if session.data properties exist before using
them.
  // This handles the first interaction when the session is brand new.

  // 1. Check if we already know the user's name from previous interactions
  if (session.data.userName) {
    // If we know it, greet them by name, acknowledging their current prompt
    return `Hello again, ${session.data.userName}! You said: "${prompt}"`;
  } else {
    // If we don't know the name, try to extract it from the current prompt
    const nameMatch = prompt.match(/my name is (\w+)/i);
    if (nameMatch && nameMatch[1]) {
      const userName = nameMatch[1];
      session.data.userName = userName; // 🔥 Store the name in session data
for future turns
      return `Nice to meet you, ${userName}! How can I help?`;
    } else {
      // If no name found in the prompt, ask the user to provide it
      return `I don't know your name yet. Please tell me, "My name is [your
name]".`;
    }
  }
};
```

Explanation of the new code:

- `import { AgentFunction, AgentSession } from '@flue/core';`: We import `AgentSession` alongside `AgentFunction`. This gives us access to Flue's session management capabilities.
- `interface GreetingSessionData { userName?: string; }`: We define a TypeScript interface to give type safety to our custom session data. This is a best practice for clarity, preventing runtime errors, and enabling better IDE support. The `?` denotes that `userName` is optional, as it won't exist in a brand new session.
- `session: AgentSession<GreetingSessionData>`: We explicitly tell TypeScript that our `session.data` object will conform to the `GreetingSessionData` interface. This ensures all interactions with `session.data` are type-checked.
- `if (session.data.userName)`: This line checks if the `userName` property already exists in our session's custom data. If it does, it means the agent remembers the user from a previous turn within the same session.

- `session.data.userName = userName;`: This is the crucial line for state management. We're assigning the extracted name to `session.data.userName`, making it available for subsequent turns within the same session. Flue automatically persists this `session.data` when the agent's response is sent.

Registering and Testing the Agent

Now, let's register this agent in our Flue application. Update your `src/index.ts` (or equivalent entry point for your Flue application, as of June 2026):

```
// src/index.ts
import { createAgentRouter } from '@flue/router';
import { greetingAgent } from './agents/greetingAgent'; // Import our new
agent

const router = createAgentRouter();

// Register the greeting agent under the 'greeting' path
router.agent('greeting', greetingAgent);

export default router.handler; // Export the handler for deployment (e.g.,
Cloudflare Worker)
```

To test this, you would typically interact with your Flue endpoint (e.g., via a `curl` command or a simple client application). Ensure your local Flue development server is running (e.g., `npm run dev`).

Testing in development:

- **First interaction:**

```
    curl -X POST http://localhost:3000/api/agent/greeting \
      -H "Content-Type: application/json" \
      -d '{"prompt": "Hello there, my name is Alice", "sessionId": "my-unique-
session-123"}'
```

```
Expected output: `Nice to meet you, Alice! How can I help?`
*What happened:* Flue received `sessionId: "my-unique-session-123"`. Since no
session with this ID existed, it created a new one. Your agent then processed
the prompt, extracted the name "Alice", and stored it in
`session.data.userName`. Flue saved this updated session.
```

- **Second interaction (using the same session ID):**

```
    curl -X POST http://localhost:3000/api/agent/greeting \
      -H "Content-Type: application/json" \
```

```
      -d '{"prompt": "Tell me a joke", "sessionId": "my-unique-session-123"}'
```

```
Expected output: `Hello again, Alice! You said: "Tell me a joke"`
*What happened:* Flue received the same `sessionId`. It loaded the existing
session for `my-unique-session-123`, which now contained `userName: "Alice"`.
Your agent found `session.data.userName` already set and used it to greet
Alice.
```

Notice how the agent remembers "Alice" because we passed the same `sessionId`. If you change the `sessionId`, it will start a new session and forget the name, demonstrating the session-scoped nature of the state.

Hands-on Challenge: The Persistent Task Planner

Let's build a slightly more complex stateful agent. Your challenge is to create an agent that helps a user manage a simple task list.

Challenge: Create an agent called `taskPlannerAgent` that can:

1. **Add tasks:** If the prompt contains "add [task description]", it should add the task to a list stored in the session.
2. **List tasks:** If the prompt contains "list tasks", it should return all tasks currently in the session.
3. **Acknowledge:** For any other input, it should acknowledge the input and remind the user of its capabilities.

Hint:

- You'll need an interface for your `session.data` that includes an array of strings, e.g., `tasks: string[]`.
- Remember to initialize `session.data.tasks` as an empty array if it doesn't exist yet, especially for the first interaction.
- Use `prompt.includes()` and `prompt.startsWith()` for simple keyword matching. Regular expressions can be more robust, but simple string methods are fine for this challenge.

What to observe/learn: Pay close attention to how the `tasks` array persists and grows within the `session.data` across multiple interactions, demonstrating the power of stateful sessions for managing dynamic data throughout a conversation.

💡 SOLUTION FOR TASK PLANNER AGENT

```
// src/agents/taskPlannerAgent.ts
import { AgentFunction, AgentSession } from '@flue/core';
interface TaskPlannerSessionData {
tasks: string[]; // Our tasks will be an array of strings
}


export const taskPlannerAgent: AgentFunction = async (
prompt: string,
session: AgentSession<TaskPlannerSessionData>,
context: Record<string, unknown>
) => {
// Ensure the tasks array is initialized if it doesn't exist yet.
// This is crucial for new sessions.
if (!session.data.tasks) {
session.data.tasks = [];
}


const lowerPrompt = prompt.toLowerCase().trim();


 if (lowerPrompt.startsWith("add ")) {
const task = prompt.substring(4).trim(); // Get task description after
"add "
if (task) {
session.data.tasks.push(task); // Add the new task to the session's task
list
return  Added task: &amp;quot;${task}&amp;quot;. You now have $
{session.data.tasks.length} tasks. ;
}
return "Please specify a task to add, e.g., 'add Buy groceries'.";
} else if (lowerPrompt.includes("list tasks")) {
if (session.data.tasks.length === 0) {
return "You currently have no tasks.";
}
// Format the tasks into a readable list
const taskList = session.data.tasks.map((t, i) => ${i + 1}. ${t} ).join('\
n');
return  Your tasks:\n${taskList} ;
} else {
return "I can help you manage tasks. Try 'add [task]' or 'list tasks'.";
}
};
```

Register in `src/index.ts`:

```
// src/index.ts
import { createAgentRouter } from '@flue/router';
import { greetingAgent } from './agents/greetingAgent';
import { taskPlannerAgent } from './agents/taskPlannerAgent'; // Import
our new agent
const router = createAgentRouter();
```

```
router.agent('greeting', greetingAgent);
router.agent('task-planner', taskPlannerAgent); // Register the task
planner agent

export default router.handler;
```

Test with `curl`:

- **First interaction (add task):**

```
curl -X POST http://localhost:3000/api/agent/task-planner \
  -H "Content-Type: application/json" \
  -d '{"prompt": "add Buy groceries", "sessionId": "my-task-
session-456"}'
```

```
Output: `Added task: "Buy groceries". You now have 1 tasks.`
```

- **Second interaction (add another task, same session ID):**

```
curl -X POST http://localhost:3000/api/agent/task-planner \
  -H "Content-Type: application/json" \
  -d '{"prompt": "add Call mom", "sessionId": "my-task-session-456"}'
```

```
Output: `Added task: "Call mom". You now have 2 tasks.`
```

- **Third interaction (list tasks, same session ID):**

```
curl -X POST http://localhost:3000/api/agent/task-planner \
  -H "Content-Type: application/json" \
  -d '{"prompt": "list tasks", "sessionId": "my-task-session-456"}'
```

```
Output: `Your tasks:\n1. Buy groceries\n2. Call mom`
```

- **Fourth interaction (new session ID):**

```
curl -X POST http://localhost:3000/api/agent/task-planner \
  -H "Content-Type: application/json" \
  -d '{"prompt": "list tasks", "sessionId": "a-new-session-789"}'
```

```
Output: `You currently have no tasks.`
```

Real-world Insights: Session Storage and Scalability

While the examples above work perfectly in a local development environment (where Flue might use in-memory storage for sessions), production systems require robust solutions for session state. An in-memory store won't persist across server restarts or scale across multiple instances.

Where Does Session State Live in Production?

In a production deployment, especially in serverless environments like Cloudflare Workers, a simple in-memory session store is insufficient. If a new worker instance handles each request, the session data would be lost. Flue, as an agent harness, is designed to be flexible and integrate with various storage backends.

- **Development (Default):** Often, Flue will use an in-memory store for sessions, which is fast and simple but not persistent across restarts or scalable horizontally. This is ideal for quick local testing.
- **Production (Configurable):** For real-world applications handling thousands of concurrent users or millions of events per day, you need a persistent and distributed session store. Common options include:
    - **Key-Value Stores:** Services like Redis, Cloudflare KV, or AWS DynamoDB are excellent for storing JSON-serializable session data, keyed by `sessionId`. These offer low-latency access and high scalability.
    - **Cloudflare Durable Objects:** When deploying to Cloudflare Workers, Durable Objects provide a unique and powerful solution. They offer single-instance state for a given ID, meaning all requests for a specific `sessionId` are routed to the same Durable Object instance. This makes them ideal for managing complex, mutable session state across multiple worker invocations, inherently solving many concurrency challenges.
    - **Databases:** Traditional databases (PostgreSQL, MongoDB) can also store session data, offering more complex querying capabilities if needed, though they might introduce higher latency than dedicated KV stores for simple session lookups.

⚡ **Real-world insight:** For Flue agents deployed on Cloudflare Workers, configuring the session manager to utilize Cloudflare Durable Objects is a common and highly effective strategy. Durable Objects abstract away the complexities of distributed state, providing a consistent, single-instance view of

your session data for any given `sessionId`. This ensures that even if different worker instances handle subsequent requests, they all access the same, correct session state without race conditions.

Tradeoffs of Stateful Agents

Implementing stateful agents introduces both powerful benefits and important considerations for production systems.

Benefits:

- **Richer User Experience:** Agents feel more intelligent and natural, leading to higher user satisfaction and engagement.
- **Efficient Interactions:** Users don't need to repeat themselves, saving time and reducing cognitive load. This directly translates to faster task completion.
- **Complex Task Handling:** Enables multi-step workflows (like multi-page forms, booking processes) that are impossible with stateless agents.
- **Reduced Token Usage (Potentially):** By carefully managing and summarizing `session.history`, you can send only the most relevant context to the LLM, potentially reducing token costs and improving response times.

Costs & Challenges:

- **Increased Infrastructure Complexity:** Requires a robust, scalable, and highly available session storage backend. This adds operational overhead and potential cost.
- **State Management Overhead:** You must actively manage what goes into `session.data` and `session.history` to prevent bloat and maintain relevance.
- **Security Concerns:** Session data can contain sensitive information (e.g., user preferences, PII) and must be protected with appropriate encryption and access controls.
- **Debugging Complexity:** Debugging issues related to incorrect or stale state can be challenging in distributed systems, requiring good observability tools.
- **Concurrency Issues:** In highly concurrent scenarios (e.g., rapid-fire requests from a single user or multiple users modifying shared state), ensuring atomic updates to session data (especially with external stores) is crucial to prevent race conditions and data corruption.

Common Pitfalls & Troubleshooting

Building stateful agents is powerful, but beware of these common traps that can lead to unexpected behavior or performance issues.

Forgetting to Initialize or Manage State

- ⚠️ **What can go wrong:** Assuming `session.data` properties will always exist or be in a certain format. On a new session, `session.data` will be an empty object, and attempting to access `session.data.someProperty` directly might result in `undefined` errors if not handled.
- **Troubleshooting:** Always check for the existence of `session.data` properties (e.g., `if (!session.data.tasks) { session.data.tasks = []; }`) and provide sensible defaults. Define clear TypeScript interfaces for your session data to catch these potential errors during development.

State Bloat

- ⚠️ **What can go wrong:** Letting `session.history` grow indefinitely or storing excessively large objects in `session.data`. This can lead to increased storage costs, slower retrieval times, and exceeding context window limits for LLMs (which often have maximum token counts for input).
- **Troubleshooting:**
    - **Summarize History:** For long conversations, periodically summarize `session.history` using an LLM to condense past turns into a concise context. This involves an additional LLM call but keeps the working history small.
    - **Truncate History:** Implement a simple strategy to keep only the `N` most recent messages in `session.history` (e.g., the last 5-10 turns).
    - **Externalize Large Data:** Store large, non-critical data (like uploaded files or extensive log data) in external storage (e.g., S3 for files, a database for logs) and only keep references (e.g., IDs or URLs) in `session.data`.

Concurrency Issues

- ⚠️ **What can go wrong:** Multiple concurrent requests to the same `sessionId` attempting to modify `session.data` simultaneously, leading to lost updates or corrupted state. This is especially relevant in systems where a single user might send multiple rapid requests or if a `sessionId` is shared (though typically sessions are per-user).

- **Troubleshooting:** While Flue's core `AgentSession` abstraction helps, the underlying session store must handle concurrency. If using a custom store, implement optimistic locking (e.g., version numbers for updates), pessimistic locking (e.g., explicit locks), or use a store (like Cloudflare Durable Objects) that inherently provides single-instance state for a given ID, effectively serializing operations for that session.

Summary

In this chapter, you've taken a significant leap towards building more intelligent and user-friendly AI agents by mastering stateful sessions in Flue Framework.

Here are the key takeaways:

- **`AgentSession` is Key:** It provides the necessary memory for agents to maintain context across multiple interactions, addressing the stateless nature of LLMs.
- **`session.data` for Custom State:** Use this object to store any application-specific data your agent needs to remember, from user names to task lists.
- **`session.history` for Conversation:** This array keeps track of the dialogue, crucial for providing conversational context to LLMs.
- **Stateful vs. Persistent:** Understand the difference between short-lived session state (for ongoing interactions) and long-term persistent agent knowledge.
- **Production Requires External Storage:** For scalable, reliable deployments, integrate Flue with distributed session stores like Cloudflare KV or, ideally for Cloudflare Workers, Durable Objects.
- **Manage State Actively:** Be mindful of state initialization, prevent state bloat, and consider concurrency to build robust and efficient agents.

You now have the tools to create agents that remember, learn, and engage in meaningful, multi-turn conversations. This capability is fundamental for developing sophisticated AI products that feel truly interactive.

Next, we'll explore how to equip your agents with even more power by integrating **tools and skills**, allowing them to interact with external systems and perform complex actions beyond just generating text.

References

- Flue — The Agent Harness Framework
- withastro/flue: The sandbox agent framework. - GitHub
- flue/docs/deploy-cloudflare.md at main · withastro/flue - GitHub
- Cloudflare Durable Objects documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 05

Deep Dive into Coding Agents: Sandboxed Execution and Persistent State

Deep Dive into Coding Agents: Sandboxed Execution and Persistent State

Imagine an AI agent that doesn't just respond to prompts but can actually write and execute code, interact with a virtual filesystem, and remember its past actions across multiple sessions. This isn't science fiction; it's the realm of "coding agents," and they demand a fundamentally different architecture than simple Large Language Model (LLM) API wrappers.

In this chapter, we'll peel back the layers of Flue's agent harness to understand how it empowers these advanced coding agents. We'll explore the critical concepts of sandboxed execution environments and persistent state, diving into why they're essential for building intelligent, reliable, and secure AI systems. By the end, you'll grasp how Flue structures these capabilities in TypeScript and be ready to build agents that can truly "think" and "act" in a controlled environment.

This chapter builds on the foundational Flue concepts we covered earlier, assuming you're comfortable with defining basic agents and tools. Get ready to elevate your agent-building skills!

The Power of the Agent Harness: Sandboxed Execution

When we talk about "coding agents," we're envisioning AI systems that can do more than just generate text. They might need to:

- Write and run Python scripts to process data.
- Interact with a virtual filesystem to create, read, or modify files.
- Execute shell commands to manage dependencies or deploy applications.

Traditional LLM SDK wrappers are excellent for sending prompts and receiving responses, but they don't offer the secure, controlled environment needed for such dynamic actions. This is where Flue's agent harness architecture shines, particularly through its emphasis on **sandboxed execution**.

Why Sandboxed Execution is Critical

Allowing an AI to execute arbitrary code or shell commands directly on your infrastructure is a massive security risk. Think about it: an LLM might hallucinate a malicious command or misinterpret instructions, leading to unintended system access or data corruption.

📌 Key Idea: Sandboxed execution isolates the agent's operational environment, preventing uncontrolled access to the host system.

Flue addresses this by providing an architectural pattern for integrating sandboxed environments. While the core Flue framework (TypeScript) itself doesn't include a full-fledged sandbox runtime, it provides the structure and mechanisms for agents to interface with one. The idea is that your agent's "tools" or "skills" can delegate execution to a secure, isolated process.

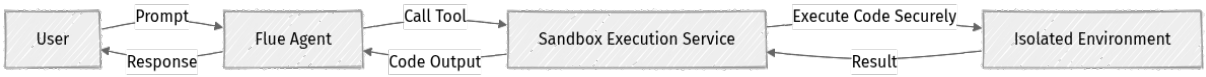


Figure 1: Flue Agent Interacting with a Sandboxed Execution Service

Flue's Approach to Sandboxing

Flue, as an agent harness, defines how agents request actions that might involve sandboxed execution. It allows you to define tools that, when invoked by the agent, communicate with an external sandbox service. This service is responsible for:

1. **Isolation:** Running code in a separate process, container, or virtual machine.
2. **Resource Limits:** Constraining CPU, memory, and network access.
3. **Security:** Preventing access to sensitive host resources or network endpoints.
4. **Input/Output Control:** Managing what goes into the sandbox and what comes out.

For instance, if you wanted a Flue agent to write and execute Python code, you wouldn't run Python directly within your Flue application. Instead, you'd create a Flue tool (e.g., `pythonExecutor`) that sends the Python code to a dedicated sandbox service (like a serverless function, a containerized environment, or a specialized sandbox API) for execution. The sandbox service then returns the output or any errors back to your Flue agent.

⚡ Real-world insight: Services like Cloudflare Workers (where Flue can be deployed) offer a highly isolated, serverless runtime environment that can act as a sandbox for certain types of operations, particularly JavaScript/TypeScript execution. For other languages or full filesystem access, you'd typically integrate with a dedicated sandbox platform.

Persistent State for Intelligent Agents

Beyond executing code, truly intelligent agents need memory. Not just short-term conversation history, but the ability to retain context, decisions, and even their internal "thoughts" across multiple interactions or even days. This is where **persistent state** comes into play.

Why Persistent State is Crucial

Consider a complex coding agent tasked with building a web application. It might:

- Generate initial project structure.
- Write a few files.
- Receive feedback from a user.
- Modify existing files based on that feedback.
- Remember the project's current state (e.g., file contents, installed dependencies) to continue its work.

Without persistent state, each interaction would be like starting from scratch, making the agent incapable of handling multi-step, long-running tasks.

🧠 Important: Persistent state allows agents to maintain a long-term memory of their environment, progress, and internal reasoning, enabling complex, multi-turn interactions and long-running tasks.

Flue's architecture accommodates persistent state, allowing you to design agents that can store and retrieve data related to their ongoing tasks. This state can include:

- **Internal Monologue/Reasoning:** The agent's thought process.
- **Intermediate Results:** Data generated during a task.
- **Environment Snapshot:** The state of a virtual filesystem or database the agent is interacting with.
- **User Preferences/Context:** Information specific to the user's ongoing session.

Managing Persistent State in Flue

Flue agents can access and modify a `state` object, which is automatically handled by the framework. When an agent runs within an `AgentRouteHandler`, this state can be stored in a backend data store (like a key-value store, database, or object storage) and loaded for subsequent invocations.

This allows an agent to:

1. **Read its past state** at the beginning of an interaction.
2. **Update its state** based on new information or actions.
3. **Persist the updated state** at the end of the interaction.

This capability is fundamental for creating agents that can learn, adapt, and complete complex projects over time.

Architecting a Coding Agent with Flue

Let's see how sandboxed execution and persistent state integrate into a Flue agent. We'll outline a conceptual architecture for a simple "Code Interpreter Agent" that can execute JavaScript code in a controlled manner and remember previous code snippets.

Agent Structure with Sandboxing and State

Our coding agent will need:

- **A Tool for Code Execution:** This tool will take code as input and send it to our "sandbox" (for this example, we'll simulate a sandbox, but in production, it would be a separate, secure service).
- **Persistent State:** To store the history of executed code or defined variables.
- **Agent Logic:** To decide when to use the code execution tool and how to update its state.

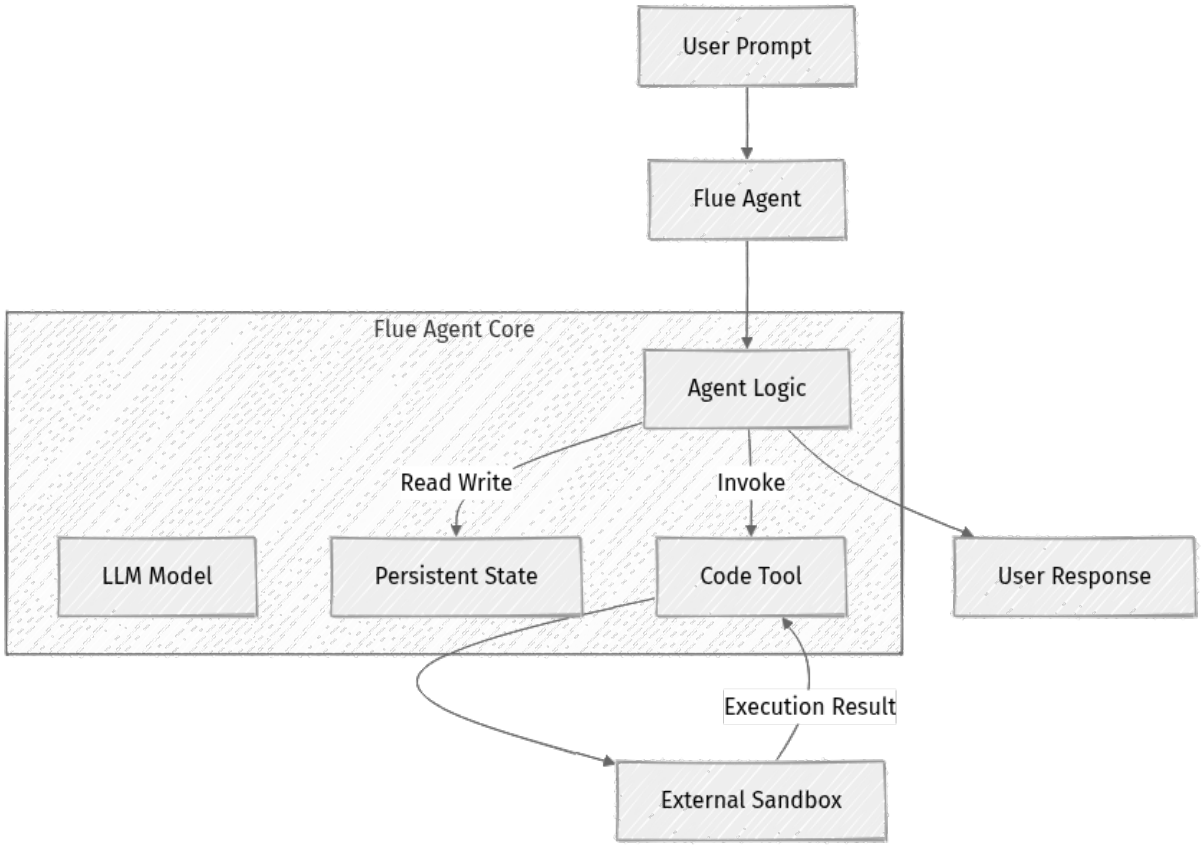


Figure 2: Flue Agent Architecture with Sandboxed Execution and Persistent State

Step-by-Step Implementation: Building a Simple Code Interpreter Agent

Let's create a simplified Flue agent that can "execute" JavaScript code and remember a simple history. For the "sandbox," we'll simulate it locally for demonstration, but remember a real sandbox would be external.

Prerequisites: Ensure you have Node.js (v18.x or later as of 2026-06-03) and TypeScript set up. If you followed previous chapters, your Flue environment should be ready.

Step 1: Initialize Your Flue Project

If you haven't already, create a new Flue project:

```
# Using npm
npm create flue@latest my-code-agent -- --template basic-agent

# Or using yarn
yarn create flue@latest my-code-agent --template basic-agent
```

```
cd my-code-agent
npm install
```

Step 2: Define the Sandboxed Execution Tool

We'll create a tool that simulates executing JavaScript code. In a real scenario, this tool would make an HTTP request to a dedicated sandbox service.

Open `src/tools.ts` and replace its content with the following:

```
// src/tools.ts
import { Tool } from '@flue/core';

// This is a *simulated* sandbox for demonstration purposes.
// IMPORTANT: In a production environment, this would call an external, secure
sandbox service
// that runs in a truly isolated environment (e.g., a dedicated container, VM,
or serverless function
// with strict resource and security policies).
// Directly executing untrusted code within your main application is a severe
security risk.
const simulatedSandboxExecute = async (code: string): Promise<string> => {
  console.log(`Executing simulated code:\n${code}`);
  // Check for specific patterns to provide deterministic results for testing
  if (code.includes('const a = 5;') && code.includes('const b = 10;') &&
code.includes('a + b;')) {
    return '15';
  } else if (code.includes('console.log("hello world");')) {
    return 'hello world';
  } else if (code.includes('2 * 3')) {
    return '6';
  } else if (code.includes('const myVar = 42;') && code.includes('myVar * 2;')
) {
    return '84';
  } else if (code.includes('const anotherVar = 100;') && code.includes('anothe
rVar / 2;')) {
    return '50';
  }
  // Fallback for any other code
  return `Simulated execution successful. Output for: "${code.substring(0, Mat
h.min(code.length, 50))}${code.length > 50 ? '...' : ''}"`;
};

/**
 * A tool to execute JavaScript code in a simulated sandboxed environment.
 * In a real application, this would interface with a secure external sandbox.
 */
export const executeJsCode: Tool = {
  name: 'executeJsCode',
  description: 'Executes JavaScript code in a simulated sandbox and returns
the result.',
  parameters: {
    type: 'object',
    properties: {
      code: {
        type: 'string',
        description: 'The JavaScript code to execute.',
      },
    },
```

```
    required: ['code'],
  },
  async run({ code }) {
    console.log(`Tool 'executeJsCode' called with code: ${code}`);
    const output = await simulatedSandboxExecute(code);
    return `Execution Result: ${output}`;
  },
};

export const tools = [executeJsCode];
```

Explanation of `src/tools.ts`:

- We define `simulatedSandboxExecute` to represent our sandbox. For safety, this example uses conditional checks on the input `code` string to return predefined results, rather than actually executing the code. This is crucial for security.
- The `executeJsCode` `Tool` takes a `code` string as a parameter.
- Its `run` method calls our simulated sandbox, logs the activity, and returns the result.

⚠️ What can go wrong: Directly executing untrusted code within your main application process is a massive security vulnerability. Always use a dedicated, secure, external sandbox for executing untrusted code. Our `simulatedSandboxExecute` function is purely for conceptual demonstration of the interface with a sandbox, not a secure implementation.

Step 3: Implement the Agent Logic with Persistent State

Now, let's create our agent that uses this tool and manages its state. We'll make it remember the last executed code snippet.

Open `src/agent.ts` and modify it:

```
// src/agent.ts
import { defineAgent, defineTool } from '@flue/core';
import { executeJsCode } from './tools'; // Import our new tool

// Define the shape of our agent's persistent state
interface AgentState {
  lastExecutedCode: string | null;
  executionHistory: string[];
}

// Our Code Interpreter Agent
export const codeInterpreterAgent = defineAgent({
  name: 'CodeInterpreterAgent',
  description: 'An agent that can execute JavaScript code and remembers its
history.',
  // Provide the tools this agent can use
  tools: [executeJsCode],
```

```
  // Define the initial state for new sessions
  initialState: {
    lastExecutedCode: null,
    executionHistory: [],
  } as AgentState, // Type assertion for initial state

  // This is the core logic of our agent
  async run({ prompt, tools, state, updateState }) {
    // Cast state to our defined interface for type safety
    const currentState = state as AgentState;

    // Log current state for debugging
    console.log('Current Agent State:', currentState);

    // If the prompt contains "execute code", try to extract and run it
    if (prompt.includes('execute code')) {
      // A simple regex to extract code. A real agent might use a more robust
parser or LLM for this.
      const codeMatch = prompt.match(/```(?:javascript|js)\n([\s\S]*?)\n```/);
      if (codeMatch && codeMatch[1]) {
        const codeToExecute = codeMatch[1].trim();

        // Use the executeJsCode tool
        const result = await tools.executeJsCode({ code: codeToExecute });

        // Update the agent's persistent state
        const newHistory = [...currentState.executionHistory, `Code: ${codeToE
xecute}\nResult: ${result}`];
        updateState({
          lastExecutedCode: codeToExecute,
          executionHistory: newHistory.slice(-5) // Keep last 5 entries
        });

        return `Code executed. Result:\n${result}\n\nI've remembered this.`;
      } else {
        return "Please provide the code in a JavaScript code block (e.g.,
```js...```) for execution.";
      }
    }

    // If the user asks about previous code
    if (prompt.toLowerCase().includes('what was the last code') || prompt.toLo
werCase().includes('last thing i ran')) {
      if (currentState.lastExecutedCode) {
        return `The last code I executed was:\n\`\`\`js\n${currentState.lastEx
ecutedCode}\n\`\`\``;
      } else {
        return "I haven't executed any code yet in this session.";
      }
    }

    // If the user asks for history
    if (prompt.toLowerCase().includes('show me the history') || prompt.toLower
Case().includes('what have i run')) {
      if (currentState.executionHistory.length > 0) {
        return `Here's your execution history:\n\n${currentState.executionHist
ory.join('\n---\n')}`;
      } else {
        return "No execution history yet.";
      }
    }
```
```

```
    // Default response if no specific action is triggered
    return `Hello! I'm a Code Interpreter Agent. You can ask me to "execute
code" by providing a JavaScript code block, or ask me "what was the last code"
I ran.`;
  },
});
```

Explanation of `src/agent.ts`:

- We define an `AgentState` interface to clearly structure our persistent data, including `lastExecutedCode` and `executionHistory`.
- `defineAgent` now includes `tools: [executeJsCode]` to make our sandbox tool available to the agent.
- `initialState` provides a default structure for new agent sessions, ensuring `lastExecutedCode` starts as `null` and `executionHistory` is an empty array.
- Inside the `run` method, we access `state` (which holds the current persistent state) and `updateState` (a function to modify and persist the state).
- When code is executed, we extract the code using a simple regex, call the `executeJsCode` tool, and then update `lastExecutedCode` and `executionHistory` in the state using `updateState`.
- We also add logic to respond to queries about the `lastExecutedCode` or `executionHistory`, demonstrating how the agent can retrieve its own memory for context.

Step 4: Expose the Agent via `AgentRouteHandler`

To make our agent accessible, we expose it using Flue's `AgentRouteHandler`. This is typically done in `src/index.ts`.

Open `src/index.ts` and modify it to use our new agent:

```
// src/index.ts
import { AgentRouteHandler } from '@flue/core';
import { codeInterpreterAgent } from './agent'; // Import our new agent

// Create an AgentRouteHandler for our code interpreter agent
const handler = new AgentRouteHandler({
  agent: codeInterpreterAgent,
  // For local development, you might use a simple memory store.
  // For production, you'd configure a persistent store (e.g., KV store,
database).
  // Flue framework provides adapters for various storage solutions.
  // As of June 2026, Flue is designed to be highly adaptable to different
  // environment-specific storage mechanisms, especially in serverless
contexts.
```

```
  // Example for Cloudflare Workers:
  // stateStore: new CloudflareKVStateStore(MY_KV_NAMESPACE)
});

// Export the handler for deployment (e.g., to Cloudflare Workers)
export default handler;
```

Explanation of `src/index.ts`:

- We import our `codeInterpreterAgent`.
- We instantiate `AgentRouteHandler` with our agent.
- The `stateStore` comment highlights that for production, you'd configure a robust, persistent storage solution. Flue is designed to integrate with various backend stores, often leveraging environment-specific options (like Cloudflare KV for Cloudflare Workers) to efficiently manage state across invocations.

Step 5: Test Your Agent Locally

You can test your agent using the Flue CLI or by sending HTTP requests.

Start the development server:

```
npm run dev
```

Now, you can send requests to your agent. You can use `curl` or a tool like Postman/Insomnia.

Example 1: Initial Prompt

```
curl -X POST http://localhost:8787/agent \
     -H "Content-Type: application/json" \
     -d '{"prompt": "Hello Flue agent!"}'
```

Expected Output (approximate):

```
{
  "response": "Hello! I'm a Code Interpreter Agent. You can ask me to
\"execute code\" by providing a JavaScript code block, or ask me \"what was
the last code\" I ran."
}
```

Example 2: Execute Code

```
curl -X POST http://localhost:8787/agent \
     -H "Content-Type: application/json" \
```

```
     -d '{"prompt": "execute code\n```js\nconst a = 5;\nconst b = 10;\na + b;
\n```"}'
```

Expected Output (approximate):

```
{
  "response":
"Code executed. Result:\nExecution Result: 15\n\nI've remembered this."
}
```

Example 3: Ask for Last Code (Demonstrates Persistent State)

```
curl -X POST http://localhost:8787/agent \
     -H "Content-Type: application/json" \
     -d '{"prompt": "What was the last code I ran?"}'
```

Expected Output (approximate):

```
{
  "response": "The last code I executed was:\n```js\nconst a = 5;\nconst b =
10;\na + b;\n```"
}
```

Notice how the agent remembers the code from the previous interaction! This is persistent state in action.

Example 4: Ask for History

```
curl -X POST http://localhost:8787/agent \
     -H "Content-Type: application/json" \
     -d '{"prompt": "Show me the execution history"}'
```

Expected Output (approximate):

```
{
  "response":
"Here's your execution history:\n\nCode: const a = 5;\nconst b = 10;\na + b;
\nResult: Execution Result: 15"
}
```

Mini-Challenge: Enhance Your Code Interpreter

Your agent can execute code and remember the last snippet. Can you make it a bit smarter?

Challenge: Modify the `codeInterpreterAgent` to allow the user to "define a variable" by providing a name and a value (e.g., "define variable `myVar` as `42`"). The agent should store this variable in its persistent state and then, if a subsequent code execution uses `myVar`, the agent should prepend the variable definition to the code it sends to the `executeJsCode` tool.

Hint:

- Add a new field to your `AgentState` interface, perhaps `definedVariables: Record<string, string>`.
- Add logic in the `run` method to parse "define variable" prompts and update `definedVariables` using `updateState`.
- Before calling `tools.executeJsCode`, check if `currentState.definedVariables` has any entries. If so, construct a string of `const [variableName] = [variableValue];` for each and prepend it to the `codeToExecute` string.

What to observe/learn: This challenge will solidify your understanding of how agents can manage complex internal state and dynamically modify their tool calls based on that state, moving towards more intelligent, context-aware behavior.

Common Pitfalls & Troubleshooting

Building agents with sandboxed execution and persistent state introduces new complexities.

1. **Sandbox Security:**
    - **Pitfall:** Assuming local execution of untrusted code is safe. It is never safe.
    - **Troubleshooting:** Always design your `executeCode` tool to communicate with a truly isolated and secure external sandbox service. This service should have strict resource limits, network egress controls, and execute code with minimal privileges. Ensure that the communication channel between your Flue agent and the sandbox is also secure, typically via authenticated API calls.

2. **State Management Overload:**
    - **Pitfall:** Storing too much data in the persistent state, or storing complex objects that are expensive to serialize/deserialize. This can lead to increased latency and storage costs.
    - **Troubleshooting:** Be mindful of what you persist. Only store information that is absolutely necessary for the agent to maintain context across sessions. Consider using simpler data structures (strings, numbers, simple objects) and optimize for efficient storage and retrieval, especially in serverless environments like Cloudflare Workers where KV stores often have size limits per entry (e.g., 10MB per value in Cloudflare KV).
3. **Resource Limits in Sandboxed Environments:**
    - **Pitfall:** Your agent might try to execute long-running code or consume excessive memory within a serverless sandbox (e.g., Cloudflare Workers). These environments have strict CPU time (e.g., 50ms for CPU time in Cloudflare Workers) and memory limits.
    - **Troubleshooting:** Design your sandbox service to handle and enforce these limits gracefully. If an agent's task is inherently long-running (e.g., complex data processing, prolonged computation), consider offloading it to a dedicated background job system rather than a short-lived serverless function. Provide clear error messages back to the agent if limits are exceeded, allowing the agent to inform the user or try an alternative approach.

Summary

In this chapter, we took a significant leap into building advanced Flue agents. We explored:

- **Sandboxed Execution:** The critical need for isolated environments when agents need to execute code or interact with system resources, emphasizing Flue's role in orchestrating these interactions through specialized tools. We learned why directly executing untrusted code is dangerous and why external sandbox services are essential for production.
- **Persistent State:** How agents can maintain long-term memory and context across interactions using Flue's `state` and `updateState` mechanisms, enabling complex, multi-turn workflows and long-running tasks.

- **Practical Implementation:** We walked through creating a simple code interpreter agent, demonstrating how to define a tool for (simulated) sandboxed execution and how to leverage persistent state to remember past actions and maintain conversational context.

By understanding and implementing these concepts, you're now equipped to design and build more sophisticated and reliable AI agents using the Flue framework. These agents can go beyond simple conversational interfaces to become powerful, autonomous workers capable of complex, stateful tasks.

Next, we'll delve into deployment strategies for these robust agents, focusing on production-minded considerations like reliability, scalability, and security when deploying to platforms like Cloudflare Workers.

References

- Flue — The Agent Harness Framework
- withastro/flue: The sandbox agent framework - GitHub
- flue/docs/deploy-cloudflare.md at main · withastro/flue - GitHub
- Cloudflare Workers Documentation
- Cloudflare Workers KV Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 06

Exposing Your Agent: Building API Endpoints with AgentRouteHandler

How do you elevate your intelligent agent from a local script to a resilient service that other applications, users, or even other agents can interact with seamlessly? The answer lies in exposing your agent as a well-defined API endpoint. In the Flue framework, this critical step is primarily achieved through the powerful `AgentRouteHandler`.

This chapter is your guide to mastering `AgentRouteHandler`. We'll explore its architecture, understand why it's more than just a typical API wrapper, and walk you through implementing it to transform your sophisticated Flue agents into accessible, production-ready API services. By the end, you'll be able to create secure, scalable endpoints, bridging the gap between your agent's internal logic and external HTTP or WebSocket requests. This is a fundamental skill for integrating agents into any robust AI system.

Before we dive into implementation, ensure you're comfortable defining Flue agents, managing their state, and integrating basic tools, as covered in previous chapters. We'll build upon our `GreetingAgent` example, making it accessible over the network.

The AgentRouteHandler: Your Agent's Production Gateway

At its core, `AgentRouteHandler` is a specialized HTTP handler designed to manage the full lifecycle of an agent interaction over a network. It doesn't just pass requests; it's a dedicated component within Flue that intelligently understands how to:

- **Initialize and manage agent sessions**: Each incoming request can represent a new conversation or a continuation of an ongoing one. The handler expertly provisions new sessions or resumes existing ones based on identifiers.
- **Parse agent-specific payloads**: It translates raw incoming HTTP request bodies or WebSocket messages into the structured inputs your agent expects, ensuring type safety and consistency.

- **Invoke agent logic**: It dispatches the parsed input and session context directly to your agent's `call` method (or other exposed methods), triggering its intelligent behavior.
- **Format agent responses**: It takes the agent's structured output and transforms it into an appropriate HTTP response body or WebSocket message, ready for client consumption.
- **Handle state persistence**: It integrates with underlying storage mechanisms to ensure your agent's state is maintained reliably across turns, even in distributed environments.

Why `AgentRouteHandler` is Crucial for Production AI

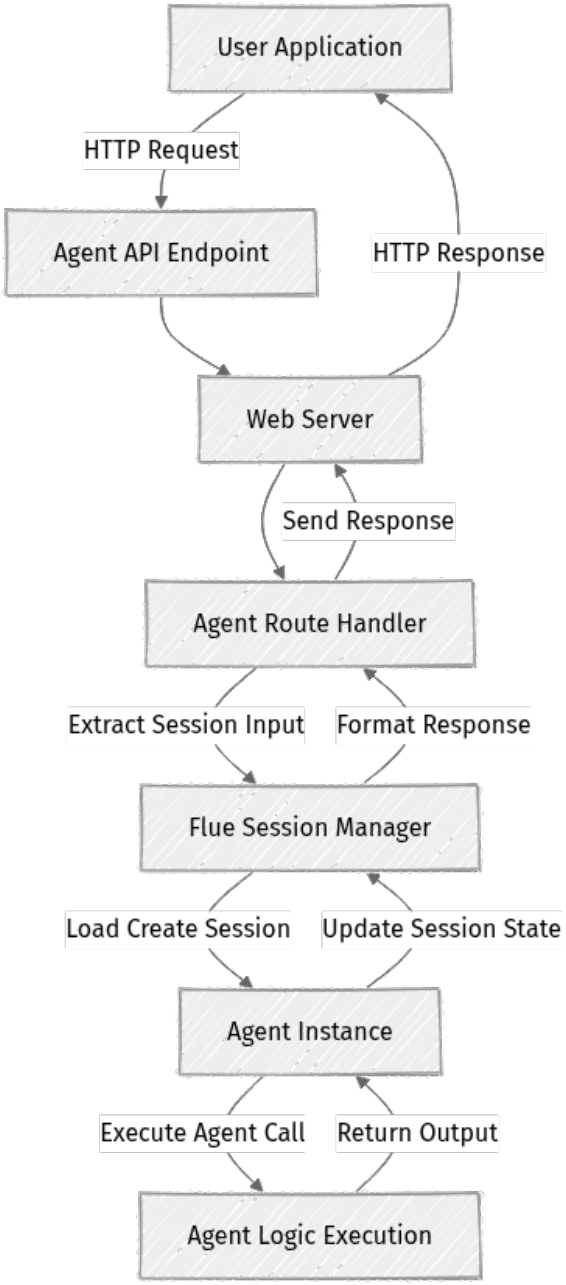
You might wonder, "Can't I just use a standard web framework like Express or Fastify and call my agent directly?" While technically feasible, `AgentRouteHandler` offers significant advantages that are indispensable for production-grade agent systems:

- **Intelligent Session Management**: Flue agents are inherently stateful. The `AgentRouteHandler` abstracts away the complexities of mapping incoming requests to specific agent sessions. This ensures continuity of conversation and context without you needing to manually manage session IDs, state lookups, or concurrency issues.
- **Standardized Interaction Layer**: It enforces a consistent and predictable way for external systems to interact with your agents. This standardization reduces integration errors, simplifies client-side development, and improves overall system reliability.
- **Deployment Agnostic Design**: While we'll use a simple Node.js server for demonstration, `AgentRouteHandler` is built to be highly adaptable. This makes it ideal for serverless environments like Cloudflare Workers, which are excellent for scaling AI agents, a topic we'll explore in the next chapter.
- **Built-in Security Hooks**: It provides natural points and patterns for integrating authentication, authorization, and robust input validation. These are critical security measures for any agent exposed to the public internet.

📌 **Key Idea**: `AgentRouteHandler` elevates agent interaction from simple function calls to a robust, session-aware API surface, specifically designed to meet the unique demands of conversational, stateful, and tool-using AI agents in production.

How it Works: The Request-Agent-Response Flow

Understanding the flow helps clarify `AgentRouteHandler`'s role. Let's visualize the basic process when an external request targets your agent:



1. **Incoming Request**: A user application sends an HTTP request (e.g., `POST /agent/chat`) or a WebSocket message to your API endpoint.
2. **Web Server Routing**: Your web server (e.g., Express) intercepts this request and routes it to the configured `AgentRouteHandler` instance.

3. **Request Parsing & Session Management**: The `AgentRouteHandler` extracts the input payload and any session identifiers from the request. It then collaborates with Flue's internal session manager to either retrieve an existing agent session's state or create a new one.
4. **Agent Execution**: The handler invokes your specific Flue agent (e.g., `GreetingAgent`) with the parsed input and the current session context. The agent performs its defined logic, potentially utilizing its integrated tools or skills.
5. **State Update & Response Generation**: After the agent executes, its updated state is saved (or persisted) via the session manager. The handler then takes the agent's output, serializes it into a standard format (like JSON), and sends it back as an HTTP response or WebSocket message to the user application.

Step-by-Step Implementation: Exposing Your First Agent

Let's put this into practice by exposing our simple `GreetingAgent` (from a hypothetical previous chapter) via an HTTP POST endpoint.

First, ensure you have a basic Flue agent. For this example, let's assume we have a `src/agents/greetingAgent.ts` file.

```
// src/agents/greetingAgent.ts
import { Agent, Session } from '@flue/core'; // Assuming @flue/core for core
Agent types

interface GreetingAgentState {
  name: string;
  greetingCount: number;
}

interface GreetingAgentInput {
  userName?: string;
  message: string;
}

interface GreetingAgentOutput {
  response: string;
  sessionState: GreetingAgentState;
}

export class GreetingAgent extends Agent<
  GreetingAgentInput,
  GreetingAgentOutput,
  GreetingAgentState
> {
  constructor() {
    super('GreetingAgent', { name: 'Guest', greetingCount: 0 });
```

```
  }

  async call(
    input: GreetingAgentInput,
    session: Session<GreetingAgentState>
  ): Promise<GreetingAgentOutput> {
    let { name, greetingCount } = session.state;

    if (input.userName) {
      name = input.userName;
    }

    greetingCount++;

    const response = `Hello, ${name}! You said: "${input.message}". This is
our ${greetingCount}th interaction.`;

    session.setState({ name, greetingCount });

    return {
      response,
      sessionState: session.state,
    };
  }
}
```

Now, let's create our server using `AgentRouteHandler` and Express.

Step 1: Initialize Your Project and Install Dependencies

If you haven't already, create a new project directory and initialize Node.js. As of **June 2026**, we'll leverage `Node.js 20.x` (or newer LTS versions), and the latest stable releases of `flue` and `express`.

First, create your project folder and initialize it:

```
mkdir flue-api-agent
cd flue-api-agent
npm init -y
```

Next, install the necessary packages. This includes the `@flue/core` library, `express` for our web server, and development dependencies like `typescript`, `ts-node` for running TypeScript directly, and `nodemon` for automatic server restarts during development.

```
npm install @flue/core express
npm install --save-dev typescript @types/node @types/express ts-node nodemon
```

Finally, set up your TypeScript configuration by generating a `tsconfig.json` file:

```
npx tsc --init
```

Open the newly created `tsconfig.json` and update it to ensure proper module resolution and type checking. Pay close attention to `esModuleInterop` and `skipLibCheck` for better compatibility with various libraries.

```
{
  "compilerOptions": {
    "target": "es2020",
    "module": "commonjs",
    "rootDir": "./src",
    "outDir": "./dist",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true,
    "allowSyntheticDefaultImports": true
  },
  "include": ["src/**/*.ts"],
  "exclude": ["node_modules"]
}
```

⚡ Quick Note: `allowSyntheticDefaultImports` is often useful when working with CommonJS modules in TypeScript, enabling cleaner `import express from 'express'` syntax.

Step 2: Create Your Agent Route Handler

Now, let's create `src/server.ts` where we'll set up our Express server and integrate the `AgentRouteHandler`.

First, create the `src` directory:

```
mkdir src
mkdir src/agents
```

Then, place the `GreetingAgent` code (from above) into `src/agents/greetingAgent.ts`.

Next, create `src/server.ts`:

```
// src/server.ts
import express from 'express';
import { AgentRouteHandler, MemorySessionManager } from '@flue/core';
import { GreetingAgent } from './agents/greetingAgent'; // Import your agent

const app = express();
const PORT = process.env.PORT || 3000;
```

```
// 1. Enable Express to parse JSON request bodies
// This middleware processes incoming requests with JSON payloads,
// making them accessible on `req.body`.
app.use(express.json());

// 2. Instantiate your agent
// This is the specific agent whose logic we want to expose via the API.
const greetingAgent = new GreetingAgent();

// 3. Create a session manager
// For simplicity in development, we use Flue's in-memory session manager.
// This stores agent states directly in the server's RAM.
const sessionManager = new MemorySessionManager();

// ⚡ Real-world insight: For production environments, especially with
// multiple server instances or serverless deployments (like Cloudflare
Workers),
// a persistent, distributed session manager is critical. Consider solutions
// backed by Redis, PostgreSQL, or cloud-specific key-value stores
// to ensure state continuity and scalability.
// Example (conceptual): const sessionManager = new
RedisSessionManager({ client: redisClient });

// 4. Instantiate the AgentRouteHandler
// This is the core component that bridges HTTP requests to your Flue agent.
// It takes your agent instance and the chosen session manager.
const agentHandler = new AgentRouteHandler(greetingAgent, sessionManager);

// 5. Define an API endpoint for your agent
// We'll create a POST endpoint at '/agent/greet' to handle interactions.
app.post('/agent/greet', async (req, res) => {
  try {
    // The AgentRouteHandler expects a structured request object.
    // We extract 'sessionId' (for continuing conversations) and 'input'
    // (the data for the agent) from the incoming request body.
    const { sessionId, input } = req.body;

    if (!input) {
      return res.status(400).json({ error: 'Missing agent input in request
body.' });
    }

    // Call the handler's `handleRequest` method
    // This method orchestrates the entire agent interaction:
    // - Loads/creates the session using the sessionId.
    // - Invokes the agent's `call` method with the provided input.
    // - Saves the updated session state.
    // - Formats the agent's output into a response.
    const result = await agentHandler.handleRequest({
      sessionId: sessionId ||
'', // Provide sessionId; empty string for a new session
      input, // The agent-specific input payload
      // Additional context can be passed if your agent needs access to
      // HTTP headers, query parameters, or other request details.
      // context: { headers: req.headers, query: req.query }
    });

    // Send the agent's structured output back to the client as JSON.
    res.json(result);
  } catch (error: any) {
    console.error('Agent handler error:', error);
    // Provide a more informative error message in development, generic in
```

```
production.
    res.status(500).json({ error: error.message || 'Internal server error
during agent interaction.' });
  }
});

// Start the Express server and listen for incoming requests.
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
  console.log(`Agent endpoint available: POST http://localhost:${PORT}/agent/
greet`);
});
```

Step 3: Configure and Run Your Server

To easily run your server during development, add a few scripts to your `package.json` file. This allows you to build your TypeScript code and run the compiled JavaScript, or use `ts-node` with `nodemon` for a live-reloading development experience.

Open your `package.json` and add the following under the `"scripts"` section:

```
// package.json
{
  "name": "flue-api-agent",
  "version": "1.0.0",
  "description": "A Flue agent exposed via an Express API.",
  "main": "dist/server.js",
  "scripts": {
    "start": "node dist/server.js",
    "dev": "nodemon --exec ts-node src/server.ts",
    "build": "tsc"
  },
  "keywords": ["flue", "agent", "api", "express", "typescript"],
  "author": "AI Expert",
  "license": "MIT",
  "dependencies": {
    "@flue/core": "^0.1.0",
    "express": "^4.18.2"
  },
  "devDependencies": {
    "@types/express": "^4.17.21",
    "@types/node": "^20.14.8",
    "nodemon": "^3.1.4",
    "ts-node": "^10.9.2",
    "typescript": "^5.5.2"
  }
}
```

Note: *`@flue/core`* version *`^0.1.0`* and *`express`* *`^4.18.2`* are placeholders for the latest stable versions as of June 2026, assuming minor updates.

Now, start your development server using the `dev` script:

```
npm run dev
```

You should see output similar to this, indicating your server is running:

```
[nodemon] 3.1.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: ts,json
[nodemon] starting `ts-node src/server.ts`
Server is running on http://localhost:3000
Agent endpoint available: POST http://localhost:3000/agent/greet
```

Step 4: Test Your Agent Endpoint

You can use `curl` from your terminal or any API client (like Postman, Insomnia, or VS Code's Thunder Client) to test your endpoint.

First interaction (initiating a new session):

This request will create a new session because no `sessionId` is provided.

```
curl -X POST \
  http://localhost:3000/agent/greet \
  -H 'Content-Type: application/json' \
  -d '{
    "input": {
      "userName": "Alice",
      "message": "Nice to meet you!"
    }
  }'
```

The server's response will include a newly generated `sessionId` and the agent's current state:

```
{
  "response": "Hello, Alice! You said: \"Nice to meet you!\". This is our 1th
interaction.",
  "sessionState": {
    "name": "Alice",
    "greetingCount": 1
  },
  "sessionId": "flue-session-some-unique-id-12345"
}
```

Note: The *`sessionId`* format may vary based on Flue's internal generation logic.

Subsequent interaction (continuing the session):

To continue the conversation, you must take the `sessionId` from the previous response and include it in your next request. This tells the `AgentRouteHandler` to load the existing session state for Alice.

```
curl -X POST \
  http://localhost:3000/agent/greet \
  -H 'Content-Type: application/json' \
  -d '{
    "sessionId": "flue-session-some-unique-id-12345",  # REPLACE with YOUR
actual sessionId
    "input": {
      "message": "How are you doing today?"
    }
  }'
```

You should observe the `greetingCount` incrementing, clearly demonstrating that the agent's state (`name` and `greetingCount`) is being correctly managed and persisted across requests using the `sessionId`:

```
{
  "response": "Hello, Alice! You said: \"How are you doing today?\". This is
our 2th interaction.",
  "sessionState": {
    "name": "Alice",
    "greetingCount": 2
  },
  "sessionId": "flue-session-some-unique-id-12345"
}
```

Congratulations! You've successfully exposed your Flue agent as a functional API endpoint, complete with stateful session management.

Mini-Challenge: Enhance and Expose Agent Behavior

Let's make our `GreetingAgent` a bit more dynamic and observe how effortlessly these changes are exposed via the `AgentRouteHandler`.

Challenge: Modify the `GreetingAgent` to accept an optional `mood` parameter in its `GreetingAgentInput`. If the `mood` is "excited", have the agent add an exclamation point and an emoji (e.g., 🎉) to its greeting. Then, test this new functionality by sending an updated request to your `src/server.ts` endpoint, including the `mood` parameter.

Hint:

- You'll need to update the `GreetingAgentInput` interface in `src/agents/greetingAgent.ts` to include `mood?: string;`.
- Inside the `call` method of `GreetingAgent`, add conditional logic to modify the `response` string based on `input.mood`.
- The `server.ts` endpoint is already set up to pass the entire `input` object from the request body directly to `agentHandler.handleRequest`. This means you won't need to change `server.ts` at all – a testament to the flexibility of `AgentRouteHandler`!

What to observe/learn: This exercise clearly demonstrates the separation of concerns within Flue. You can modify your agent's internal capabilities and complex logic independently. Because `AgentRouteHandler` is designed to be flexible with agent inputs, these changes can often be exposed through your existing API endpoint without extensive API code modifications. This modularity is a key benefit for maintainable and scalable AI systems.

Common Pitfalls & Troubleshooting

Exposing agents via API endpoints introduces new layers where issues can arise. Here are a few common mistakes and how to debug them effectively:

- **CORS (Cross-Origin Resource Sharing) Errors**:
  - **What it is**: If your frontend application (e.g., running on `localhost:5173`) tries to access your agent API (running on `localhost:3000`), web browsers enforce security policies that prevent these "cross-origin" requests by default.
  - **Solution**: Implement CORS middleware in your Express application. For development, you might allow all origins. For production, restrict it to your specific frontend domains for security.

```
    // In src/server.ts, near the top after `app = express();`
    import cors from 'cors'; // First, install: npm install cors @types/cors
    app.use(cors()); // For development, this allows all origins.
    // 🔥 Optimization / Pro tip: In production, configure specific origins
for security:
```

```
    // app.use(cors({ origin: 'https://your-frontend-domain.com' }));
```

- **Incorrect `sessionId` Handling (State Not Persisting)**:
  - **What it is**: If your agent isn't maintaining its state across calls, or if `greetingCount` isn't incrementing, it's likely due to an incorrect or missing `sessionId` in subsequent requests. A new session is created if `sessionId` is absent or invalid.
  - **Solution**: Always ensure your client explicitly sends the `sessionId` received from the first interaction for all subsequent calls within the same conversation. Verify that `req.body.sessionId` is correctly populated in your `src/server.ts` route.
- **Payload Mismatches (Agent Not Receiving Expected Input)**:
  - **What it is**: Sending an `input` object from your client that doesn't match the `GreetingAgentInput` interface your agent expects can lead to unexpected `undefined` values, incorrect agent behavior, or even runtime errors within the agent's `call` method.
  - **Solution**: Implement robust input validation (e.g., using libraries like Zod or Joi) in your Express route before passing the input to `agentHandler.handleRequest`. This ensures your agent always receives valid, type-safe data.
- **Agent Execution Errors (Generic 500 Responses)**:
  - **What it is**: Unhandled exceptions within your agent's `call` method can lead to your endpoint crashing or returning generic 500 "Internal Server Error" responses without specific details.
  - **Solution**: While `AgentRouteHandler` has some error handling, it's best practice to wrap your agent's internal logic with `try-catch` blocks where appropriate. Log these errors extensively using a proper logging library (e.g., Winston, Pino) to aid in debugging. Provide user-friendly error messages while retaining detailed logs for developers.

Summary

In this chapter, we've taken a significant and practical step towards making your Flue agents production-ready by exposing them as robust API endpoints. This is where your agent's intelligence truly becomes accessible and valuable to other systems and users.

Here are the key takeaways from our exploration:

- **`AgentRouteHandler` is the specialized bridge**: It acts as the dedicated gateway between external HTTP/WebSocket requests and your Flue agents, expertly handling session management, input parsing, agent invocation, and response formatting.
- **Beyond generic API wrappers**: It provides specialized capabilities for managing stateful agent interactions and conversational context, which standard web frameworks don't offer out-of-the-box.
- **Incremental implementation**: We systematically set up an Express server, integrated `AgentRouteHandler`, and created a functional `POST` endpoint to interact with our `GreetingAgent`.
- **Session continuity is paramount**: The `sessionId` is a critical component for maintaining conversation context and agent state across multiple API calls, leveraging Flue's powerful session management capabilities.
- **Production readiness requires persistence**: While `MemorySessionManager` is convenient for local development, a persistent, distributed session store (like Redis or a database) is an absolute necessity for real-world deployments to ensure reliability and scalability.
- **Proactive troubleshooting**: Common issues such as CORS, `sessionId` mismatches, payload errors, and agent execution failures can be effectively addressed with standard web development and debugging practices.

Now that your agent is accessible via a well-defined API, the next logical step is to deploy it to a production environment where it can handle real user traffic reliably and at scale. In the next chapter, we'll delve into practical deployment strategies, particularly focusing on serverless platforms like Cloudflare Workers, which are an excellent fit for the lean and scalable nature of Flue agents.

References

- Flue — The Agent Harness Framework
- withastro/flue: The sandbox agent framework - GitHub
- Express.js Official Documentation: The official guide for the popular Node.js web framework.
- Mozilla Developer Network (MDN) Web Docs: CORS: A comprehensive explanation of Cross-Origin Resource Sharing.
- Node.js Official Website: Information and downloads for the Node.js runtime.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 07

Deploying Flue Agents to Cloudflare Workers: Production Considerations

Introduction: Taking Your Flue Agent to the Edge

So far, we've focused on building powerful, sandboxed AI agents locally with Flue. But what happens when you're ready to share your intelligent creations with the world? How do you move from a local `npm run dev` to a globally available, scalable, and reliable service?

This chapter is your guide to deploying Flue agents to Cloudflare Workers, a powerful serverless platform designed for edge computing. We'll explore why Workers are an excellent fit for Flue's agent harness architecture, walk through the setup, and tackle the critical considerations for production environments, especially around state management.

By the end of this chapter, you'll be able to confidently deploy your Flue agents, making them accessible and performant for users across the globe. You'll need a basic Flue agent from previous chapters, a Cloudflare account, and the `wrangler` CLI installed.

Why Cloudflare Workers for Flue Agents?

Cloudflare Workers offer a compelling platform for deploying AI agents due to their unique characteristics that align well with Flue's design principles.

Edge Computing for Low Latency

Cloudflare's global network is vast, with data centers in hundreds of cities worldwide. When you deploy a Worker, your code runs on servers geographically close to your users.

📌 Key Idea: Edge computing brings your agent closer to the user, significantly reducing network latency.

For AI agents that might involve multiple turns or tool calls, minimizing latency is crucial for a responsive user experience. Instead of a request traveling halfway across the world to a central server, it hits an "edge" server nearby, processes the agent's logic, and returns a response rapidly.

Serverless Architecture and Scalability

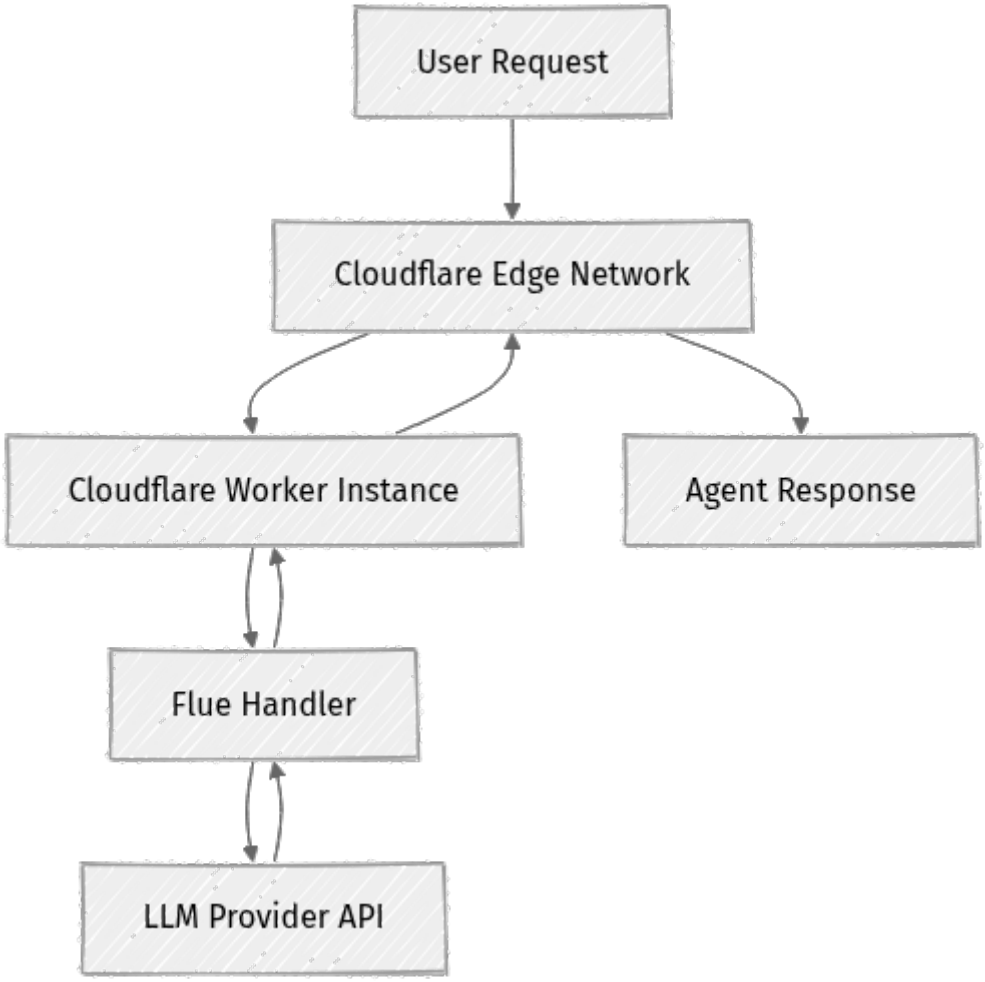
Cloudflare Workers operate as a serverless platform. This means:

- **No Server Management:** You write code; Cloudflare handles the underlying infrastructure, patching, and scaling.
- **Automatic Scaling:** As traffic to your agent increases, Cloudflare automatically scales up your Worker instances to handle the load, and scales down when demand subsides. This is ideal for unpredictable AI agent traffic patterns.
- **Cost-Effective:** You typically pay only for the compute time and requests your Worker consumes, making it highly efficient for intermittent or variable workloads.

Flue's `AgentRouteHandler` and the Worker `fetch` API

Flue's `AgentRouteHandler` is designed to expose your agents over HTTP or WebSockets. Cloudflare Workers are primarily event-driven, with the core entry point being the `fetch` handler that processes incoming HTTP requests. This is a perfect match.

When a request arrives at a Cloudflare Worker, it invokes a `fetch` event listener. Your Flue `AgentRouteHandler` can seamlessly integrate into this, processing the request and generating a response, just as it would in a Node.js environment.



The diagram above illustrates how a user request flows through Cloudflare's edge network to your Flue agent running within a Worker. The `AgentRouteHandler` acts as the bridge, processing requests and interacting with external services like LLM providers.

Setting Up Your Cloudflare Environment

Before deploying, you need to prepare your Cloudflare account and local development environment.

1. Create a Cloudflare Account

If you don't have one, sign up for a free Cloudflare account at https://www.cloudflare.com/. Many of the Worker features, including a generous free tier, are available without a paid plan.

2. Install the `wrangler` CLI

`wrangler` is Cloudflare's command-line interface for developing, testing, and deploying Workers.

Open your terminal and install `wrangler` globally:

```
npm install -g wrangler@latest
```

⚡ Quick Note: As of 2026-06-03, `wrangler` is consistently updated. Always use `@latest` or check the official Cloudflare Workers documentation for the most current stable version to ensure compatibility.

3. Authenticate `wrangler`

Once installed, you need to link `wrangler` to your Cloudflare account:

```
wrangler login
```

This command will open a browser window, prompt you to log in to Cloudflare, and authorize `wrangler`. Follow the on-screen instructions. Once complete, your terminal will confirm successful authentication.

4. Configure Your Project with `wrangler.toml`

Every Cloudflare Worker project uses a `wrangler.toml` file for configuration. This file tells `wrangler` how to build and deploy your Worker.

You'll create this file in the root of your Flue project. It specifies your Worker's name, type, entry point, and any environment variables or bindings (like KV stores).

Adapting Your Flue Agent for Cloudflare Workers

Flue's `AgentRouteHandler` is designed for flexibility. To run it inside a Cloudflare Worker, you simply need to expose it as the Worker's `fetch` handler.

Let's assume you have a basic Flue agent, perhaps from a previous chapter, located in `src/agent.ts` or similar.

```
// src/agent.ts
import { Agent, AgentFunction } from "@flue/core";

const helloFunction: AgentFunction = {
  name: "hello",
  description: "Says hello to a given name.",
  parameters: {
    type: "object",
    properties: {
      name: { type: "string", description: "The name to say hello to." },
    },
    required: ["name"],
```

```
  },
  async execute({ name }: { name: string }) {
    return `Hello, ${name}! Welcome to the Flue agent.`;
  },
};

export const myAgent = new Agent({
  name: "GreeterAgent",
  description: "An agent that greets people.",
  functions: [helloFunction],
  llmConfig: {
    model: "claude-3-opus-20240229", // Or your preferred LLM
  },
});
```

Now, let's create the entry point for your Cloudflare Worker. This will typically be `src/index.ts`.

```
// src/index.ts
import { AgentRouteHandler } from "@flue/server";
import { myAgent } from "./agent"; // Import your agent

// Create an AgentRouteHandler instance
const agentHandler = new AgentRouteHandler({
  agent: myAgent,
  // Optional: Configure a base path if you want to serve the agent under a
specific URL segment
  // basePath: "/api/greeter",
});

// Cloudflare Worker's entry point: the fetch handler
export default {
  async fetch(
    request: Request,
    env: Env, // Env contains bindings like KV, D1, etc.
    ctx: ExecutionContext
  ): Promise<Response> {
    // Pass the request to the Flue AgentRouteHandler
    // The handler will process the request (e.g., /chat, /invoke, /socket)
    // and return a standard Response object.
    return agentHandler.fetch(request);
  },
};

interface Env {
  // Define any Cloudflare bindings here, e.g.,
  // MY_KV_NAMESPACE: KVNamespace;
  // MY_D1_DATABASE: D1Database;
}
```

In this `index.ts`, we import our `myAgent` and create an `AgentRouteHandler`. The `fetch` function, which is the standard entry point for a Cloudflare Worker, then simply delegates the incoming `request` to `agentHandler.fetch()`. This makes the integration incredibly straightforward.

Step-by-Step Deployment

Let's walk through the process of setting up a new Flue project and deploying it.

1. Initialize a New Flue Project

If you don't have one, create a new Flue project.

```
mkdir flue-worker-agent
cd flue-worker-agent
npm init -y
npm install @flue/core @flue/server typescript @types/node
npx tsc --init
```

Update `tsconfig.json` to include `"moduleResolution": "bundler"` and `"target": "ES2022"`, and ensure `outDir` is set to `./dist`.

2. Create Your Agent and Worker Entry Point

Create `src/agent.ts` and `src/index.ts` as shown in the "Adapting Your Flue Agent" section above.

3. Configure `wrangler.toml`

Create a `wrangler.toml` file in the root of your project:

```
# wrangler.toml
name = "flue-greeter-agent" # Your unique worker name
main = "src/index.ts"      # The entry point for your Worker
compatibility_date = "2024-01-01"
# Required for modern Workers, pick a recent date
compatibility_flags = ["nodejs_compat"]
# Enables Node.js polyfills for better compatibility

# Optional: Add environment variables for your LLM API Key
# You should set these securely via `wrangler secret put` in production.
# For local development, you can use .dev.vars
[vars]
OPENAI_API_KEY = "YOUR_OPENAI_KEY" # Replace with your LLM's API key variable
name
```

🧠 Important: The `compatibility_date` and `compatibility_flags = ["nodejs_compat"]` are crucial. `nodejs_compat` provides polyfills for Node.js APIs that Flue might rely on, making deployment smoother. Always set `compatibility_date` to a recent date to opt into the latest Worker features and bug fixes.

4. Set Environment Variables Securely

For sensitive information like LLM API keys, you should use `wrangler secret put`. This encrypts the variable and makes it available to your Worker at runtime without being committed to your repository.

```
wrangler secret put OPENAI_API_KEY
# It will prompt you to enter the value for OPENAI_API_KEY
```

For local development, you can create a `.dev.vars` file (which should be `.gitignore`d) at the root of your project:

```
# .dev.vars
OPENAI_API_KEY="sk-YOUR_TEST_KEY"
```

Then, in your Flue agent, you would access this variable via `process.env`:

```
// src/agent.ts (modified LLM config)
// ...
export const myAgent = new Agent({
  // ...
  llmConfig: {
    model: "claude-3-opus-20240229",
    apiKey: process.env.OPENAI_API_KEY, // Access from env
  },
});
```

5. Add Deployment Scripts to `package.json`

Add a `deploy` script to your `package.json`:

```
{
  "name": "flue-worker-agent",
  "version": "1.0.0",
  "description": "",
  "main": "dist/index.js",
  "scripts": {
    "build": "tsc",
    "deploy": "wrangler deploy",
    "start": "wrangler dev"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "@flue/core": "latest",
    "@flue/server": "latest"
  },
  "devDependencies": {
    "@types/node": "latest",
    "typescript": "latest",
    "wrangler": "latest"
```

```
  }
}
```

6. Deploy Your Agent!

First, ensure your TypeScript compiles:

```
npm run build
```

Then, deploy your Worker:

```
npm run deploy
```

`wrangler` will compile your Worker (if not already done), upload it to Cloudflare, and provide you with the public URL where your agent is accessible. It might also prompt you to confirm the deployment.

7. Test Your Deployed Agent

Once deployed, you can test it using `curl` or a browser. If your `AgentRouteHandler` is configured for a default `/chat` endpoint, you might try:

```
curl -X POST "https://YOUR_WORKER_URL/chat" \
     -H "Content-Type: application/json" \
     -d '{
           "messages": [
             {"role": "user", "content": "Say hello to Alice using the hello
function."}
           ]
         }'
```

Replace `YOUR_WORKER_URL` with the URL provided by `wrangler deploy`. You should receive a JSON response containing your agent's reply.

Persistent State in a Serverless Environment: Production Considerations

One of the biggest challenges with serverless functions like Cloudflare Workers is their stateless nature. Each invocation of your Worker might run on a different instance, meaning local memory or filesystem changes are not persistent across requests or sessions.

For Flue agents, especially those designed for complex coding tasks or multi-turn interactions, maintaining state (e.g., session history, tool output, sandboxed filesystem changes) is critical.

⚠️ What can go wrong: Relying on in-memory state for a Flue agent deployed to Workers will lead to lost context, broken sessions, and unpredictable behavior.

Solutions for Persistent State: Cloudflare Offerings

Cloudflare provides several services to address state management for Workers:

1. **Cloudflare KV (Key-Value Store):**
    - **What it is:** A globally distributed, eventually consistent key-value store.
    - **Why it exists:** Ideal for storing simple session data, configuration, or cached responses.
    - **How it works:** You bind a KV namespace to your Worker, then use its API (`env.MY_KV_NAMESPACE.put("key", "value")`) to store and retrieve data.
    - ⚡ Real-world insight: Excellent for storing user session IDs mapped to serialized agent states or chat histories.
2. **Cloudflare D1 (Serverless Database):**
    - **What it is:** A serverless SQL database built on SQLite, running at the edge.
    - **Why it exists:** For more structured data, complex queries, or relational needs.
    - **How it works:** Bind a D1 database to your Worker, then use its API (`env.MY_D1_DATABASE.prepare("SELECT * FROM users").all()`) to interact with it.
    - ⚡ Real-world insight: Perfect for storing agent knowledge bases, user profiles, or detailed interaction logs.

3. **Cloudflare R2 (Object Storage):**
    - **What it is:** S3-compatible object storage.
    - **Why it exists:** For large files, media, or agent-generated assets.
    - **How it works:** Bind an R2 bucket to your Worker, then use its API to put and get objects.
    - ⚡ Real-world insight: Useful if your coding agent generates large code files, images, or documentation that needs to be stored and retrieved.

Integrating Persistent State with Flue

Flue's `Agent` constructor can accept a `stateStore` option. For Workers, you would implement a custom `StateStore` that interacts with one of Cloudflare's persistent storage solutions.

```
// Example: A simplified custom KV-backed state store interface
import { AgentState, StateStore } from "@flue/core";

class CloudflareKVStateStore implements StateStore {
  private kv: KVNamespace;

  constructor(kvNamespace: KVNamespace) {
    this.kv = kvNamespace;
  }

  async get(key: string): Promise<AgentState | undefined> {
    const stateJson = await this.kv.get(key);
    return stateJson ? JSON.parse(stateJson) : undefined;
  }

  async set(key: string, state: AgentState): Promise<void> {
    await this.kv.put(key, JSON.stringify(state));
  }

  async delete(key: string): Promise<void> {
    await this.kv.delete(key);
  }
}

// In src/index.ts or where your agent is defined:
// Make sure to configure the KV binding in wrangler.toml and Env interface
// const kvStore = new CloudflareKVStateStore(env.MY_KV_NAMESPACE);
// const myAgent = new Agent({ /* ..., */ stateStore: kvStore });
```

This conceptual example demonstrates how you'd wrap Cloudflare's KV API with Flue's `StateStore` interface. In a real application, you'd ensure proper error handling, serialization, and potentially caching.

Mini-Challenge: Deploy a Stateful Agent

Let's enhance our previous agent to conceptually use state, and ensure it deploys correctly.

Challenge: Modify your `flue-worker-agent` project to include a simple "counter" agent. This agent won't persist its count yet (as that requires actual KV setup), but it will demonstrate how an agent would manage a session-scoped counter if a `StateStore` were provided. Deploy this updated agent to Cloudflare Workers.

1. **Update `src/agent.ts`:** Add a new function to your `GreeterAgent` that increments a counter. This counter will be in-memory for now, but imagine it being part of the `AgentState`.

```
    // src/agent.ts
    import { Agent, AgentFunction } from "@flue/core";

    interface MyAgentState {
      count: number;
    }

    const helloFunction: AgentFunction = {
      name: "hello",
      description: "Says hello to a given name.",
      parameters: {
        type: "object",
        properties: {
          name: { type: "string", description: "The name to say hello to." },
        },
        required: ["name"],
      },
      async execute({ name }: { name: string }) {
        return `Hello, ${name}! Welcome to the Flue agent.`;
      },
    };

    const incrementCounterFunction: AgentFunction<MyAgentState> = {
      name: "incrementCounter",
      description: "Increments a session counter and returns its new value.",
      parameters: { type: "object", properties: {} }, // No parameters for
simplicity
      async execute(_params, { state }) {
        state.count = (state.count || 0) + 1;
        return `Counter incremented to ${state.count}.`;
      },
    };

    export const myAgent = new Agent<MyAgentState>({
      name: "GreeterAgent",
      description: "An agent that greets people and counts.",
      functions: [helloFunction, incrementCounterFunction],
      llmConfig: {
        model: "claude-3-opus-20240229", // Or your preferred LLM
        apiKey: process.env.OPENAI_API_KEY,
```

```
    },
    // Initialize default state for new sessions
    initialState: { count: 0 },
});
```

1. **Ensure `src/index.ts` and `wrangler.toml` are correctly configured** from previous steps.
2. **Run `npm run build` and `npm run deploy`.**
3. **Test the deployed agent:**

```
    # First call
    curl -X POST "https://YOUR_WORKER_URL/chat" \
         -H "Content-Type: application/json" \
         -d '{
               "messages": [
                 {"role": "user", "content": "Increment the counter."}
               ]
             }'

    # Second call (expect count 1, then 2, etc. if session is maintained,
    # but due to stateless Workers, it will likely reset to 1 each time
without KV)
    curl -X POST "https://YOUR_WORKER_URL/chat" \
         -H "Content-Type: application/json" \
         -d '{
               "messages": [
                 {"role": "user", "content": "Increment the counter again."}
               ]
             }'
```

What to observe/learn: Without an external `StateStore` (like KV), each call to "increment the counter" will likely result in "Counter incremented to 1." This highlights the stateless nature of Workers and the critical need for external persistence for stateful agents. If you were using `wrangler dev` locally, the count might persist across calls from the same client due to local session handling, but this won't happen on a deployed Worker.

Common Pitfalls & Troubleshooting

Deploying to serverless environments can have its quirks. Here are a few common issues and how to approach them:

1. **`wrangler` Authentication Issues:**
    - **Problem:** `wrangler` reports "Not authorized" or fails to deploy.
    - **Solution:** Re-run `wrangler login` to refresh your authentication token. Ensure your Cloudflare account has the necessary permissions for Workers.
2. **Environment Variable Management:**
    - **Problem:** Agent fails because `process.env.OPENAI_API_KEY` is undefined.
    - **Solution:** Verify you've set secrets using `wrangler secret put` for production deployments. For local testing with `wrangler dev`, ensure your `.dev.vars` file is correctly formatted and present. Remember, variables set in `wrangler.toml` are for build-time or non-sensitive configurations.
3. **Cold Starts:**
    - **Problem:** The first request to your Worker after a period of inactivity is slow.
    - **Solution:** This is inherent to serverless. Cloudflare Workers are generally very fast at cold starts (often <50ms), but for extremely latency-sensitive applications, consider strategies like "warming" (sending periodic dummy requests) or Cloudflare's "Always On" feature (if available and suitable for your budget/plan).

4. **Resource Limits:**
    - **Problem:** Worker crashes with "Exceeded CPU limit" or "Memory limit exceeded."
    - **Solution:** Cloudflare Workers have limits on CPU time (e.g., 50ms for free tier, 30s for paid) and memory. Complex agent operations, very long LLM interactions, or large tool outputs can hit these.
        - **Optimize agent logic:** Streamline tool calls, reduce unnecessary processing.
        - **Chunk large inputs/outputs:** Process data in smaller batches.
        - **Offload heavy computation:** If an agent task is truly compute-intensive, consider offloading parts to a dedicated compute service, and using the Worker as an orchestrator.
5. **`nodejs_compat` issues:**
    - **Problem:** Some Node.js APIs used by Flue or its dependencies don't work as expected.
    - **Solution:** Ensure `compatibility_flags = ["nodejs_compat"]` is set in `wrangler.toml`. If specific Node.js modules are still problematic, you might need to find Worker-compatible alternatives or polyfills, or abstract those parts of your agent.

Summary: Agents Live at the Edge

In this chapter, we've taken a significant step from local development to global deployment.

Here are the key takeaways:

- **Cloudflare Workers are an excellent fit for Flue agents** due to their edge computing capabilities, serverless scalability, and direct integration with Flue's `AgentRouteHandler`.
- **`wrangler` is your essential tool** for managing and deploying Workers, handling authentication, and configuring your project via `wrangler.toml`.
- **Adapting your Flue agent is straightforward**, primarily involving exposing the `AgentRouteHandler` through the Worker's `fetch` API.
- **Persistent state is a critical consideration** for production Flue agents on Workers. Cloudflare KV, D1, and R2 offer robust solutions for managing session data, structured information, and large assets.

- **Troubleshooting involves checking `wrangler` config, environment variables, and understanding serverless constraints** like cold starts and resource limits.

You now have the knowledge to deploy your Flue agents to a powerful, globally distributed platform. This opens up immense possibilities for building responsive, scalable AI products.

References

- Cloudflare Workers Documentation: https://developers.cloudflare.com/workers/
- `wrangler` CLI Documentation: https://developers.cloudflare.com/workers/wrangler/
- Cloudflare KV Documentation: https://developers.cloudflare.com/workers/runtime-apis/kv/
- Flue Framework GitHub Repository: https://github.com/withastro/flue
- Flue Deploy to Cloudflare Documentation: https://github.com/withastro/flue/blob/main/docs/deploy-cloudflare.md

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 08

Designing Robust Agents: Best Practices for Scalability and Maintainability

Introduction

You've built your first Flue agent, perhaps deployed it, and seen it spring to life! That's a fantastic start. But moving from a functional prototype to a production-ready AI agent requires a deeper understanding of design principles that prioritize scalability, reliability, and maintainability. In the real world, agents need to handle diverse inputs, recover from errors gracefully, and provide insights into their operations.

This chapter is your guide to elevating your Flue agents from "it works" to "it works well and reliably." We'll dive into architectural best practices, explore advanced state management, and equip your agents with robust error handling and observability features. By the end, you'll have a blueprint for building AI agents that can confidently tackle complex tasks in a production environment.

To make the most of this chapter, you should be familiar with the core concepts of Flue, including creating basic agents, understanding `AgentRouteHandler`, and the general deployment process, as covered in previous sections.

Core Concepts: Building for Production

Building robust AI agents is similar to constructing any complex software system. It requires careful thought around structure, data flow, and failure modes. Let's explore the foundational concepts that enable production-grade Flue agents.

Modular Agent Design: The Power of Skills

Imagine an AI agent as a highly skilled team leader. This leader doesn't do everything themselves; instead, they delegate tasks to specialized team members. In Flue, these "team members" are your agent's **skills** or **tools**.

What is it? Modular agent design means breaking down your agent's overall logic into smaller, independent, and reusable units. Each unit, often implemented as a class or function, handles a specific capability or interacts with a particular external system.

Why does it exist?

- **Readability:** Complex agent logic becomes easier to understand when segmented.
- **Reusability:** A well-defined skill, like "search the web" or "summarize text," can be reused across multiple agents.
- **Testability:** Individual skills can be tested in isolation, simplifying debugging.
- **Maintainability:** Changes or updates to one skill are less likely to impact others.

How it functions: Your main `Agent` class acts as the orchestrator. It uses an LLM to decide which skill to invoke, when, and with what parameters. The skills themselves encapsulate the actual implementation details, making the agent's core logic cleaner and more focused on decision-making.

📌 **Key Idea:** An agent is a conductor, not a single instrument. It directs specialized skills to perform actions.

State Management Strategies: Beyond Ephemeral Sessions

Flue provides robust state management, but knowing when to use which type of state is critical for production.

- **Ephemeral Session State:** This is the default. When a user interacts with an `AgentRouteHandler`, a session is created, and any state stored within that session (e.g., `agent.state.set('key', 'value')`) persists only for the duration of that specific interaction or until the session expires. This is ideal for short-lived conversations or single-turn requests.
- **Persistent State:** For more complex agents, especially those that need to remember information across long periods, restarts, or different user sessions (e.g., a coding agent remembering its workspace, or a content agent tracking generated drafts), Flue allows for explicit persistent state. This typically involves integrating with a database or persistent storage solution.

Tradeoffs:

- **Complexity:** Persistent state adds architectural complexity (database setup, serialization, error handling).
- **Cost:** Storing data persistently incurs storage and potentially read/write operation costs.

- **Performance:** Retrieving and saving persistent state can introduce latency compared to in-memory session state.

⚡ **Real-world insight:** For a customer support agent, session state is usually enough. For an agent managing a project, tracking tasks, or maintaining a knowledge base, persistent state is essential. Cloudflare D1 or KV stores are common choices when deploying on Cloudflare Workers.

Robust Tooling and Error Handling

Agents are only as reliable as their tools. In a production environment, tools can fail due to network issues, invalid inputs, API rate limits, or unexpected external service behavior. Your agent needs to anticipate and handle these failures gracefully.

Designing Reliable Tools:

- **Input Validation:** Always validate inputs to your skills. Don't trust the LLM's output implicitly.
- **Graceful Failure:** Instead of crashing, a skill should return a clear error message or a fallback value.
- **Retries:** For transient errors (like network glitches), implement simple retry mechanisms with exponential backoff.
- **Timeouts:** Prevent skills from hanging indefinitely, consuming resources.

Agent-Level Error Handling: When a skill fails, the agent needs to decide what to do next.

- **Report:** Inform the user or log the error.
- **Retry:** Attempt the same skill again, perhaps with modified parameters.
- **Fallback:** Use an alternative skill or provide a generic response.
- **Ask for Clarification:** If the input was ambiguous, ask the user for more information.

⚠️ **What can go wrong:** Unhandled errors in skills can lead to agent crashes, infinite loops, or provide nonsensical responses to users, eroding trust.

Observability: Logging and Monitoring

In production, you can't always be watching your agent. Observability—the ability to understand an agent's internal state from its external outputs—is crucial.

Why it matters:

- **Debugging:** Pinpoint the exact step where an agent went wrong.
- **Performance Analysis:** Identify bottlenecks or slow-running skills.
- **Behavior Understanding:** See how the agent makes decisions, what prompts it generates, and how it reacts to different inputs.
- **Compliance & Auditing:** In some domains, logging agent actions is a regulatory requirement.

What to Log:

- **Agent Decisions:** Which skill was chosen, why, and what parameters were passed.
- **LLM Interactions:** The full prompt sent to the LLM and its raw response.
- **Skill Calls:** Inputs to skills, outputs from skills, and any errors.
- **Session State Changes:** How the agent's internal state evolves.
- **Errors and Warnings:** Detailed stack traces and context.
- **Latency:** How long each step or skill call takes.

How to Implement:

- Use a structured logging library (e.g., Winston, Pino) in Node.js environments.
- Integrate with platform-specific logging services (e.g., Cloudflare Workers automatically logs `console.log` output to their analytics dashboard).
- Consider tracing tools for complex multi-step agent workflows.

Step-by-Step Implementation: Refactoring for Robustness

Let's take a hypothetical "Content Idea Generator" agent and refactor it to incorporate modularity, basic error handling, and logging. This agent will suggest blog post ideas and outline them, leveraging separate "Idea Generation" and "Outline Creation" skills.

First, ensure you have a basic Flue project set up (e.g., `npm create flue@latest`).

Step 1: Define a Modular Skill Interface

We'll define a simple interface for our skills and then implement two concrete skills. This enhances type safety and makes it clear what each skill is expected to do.

Create a new file `src/skills/index.ts`:

```
// src/skills/index.ts

/**
 * Defines a common interface for agent skills.
 * Each skill should have a `name` and an `execute` method.
 */
export interface AgentSkill {
  name: string;
  description: string; // A description for the LLM to understand its purpose
  execute: (input: string, context?: Record<string, any>) => Promise<string>;
}

// Now, let's create our first skill: IdeaGenerationSkill

export class IdeaGenerationSkill implements AgentSkill {
  name = "generate_ideas";
  description = "Generates a list of blog post ideas based on a given topic.";

  async execute(topic: string): Promise<string> {
    console.log(`[Skill: ${this.name}] Generating ideas for topic: "$
{topic}"`);
    // In a real scenario, this would call an external API or a more complex
LLM prompt.
    // For demonstration, we'll simulate a simple generation.
    if (!topic || topic.trim() === "") {
      throw new Error("Topic cannot be empty for idea generation.");
    }
    const ideas = [
      `"The Future of AI in Content Creation"`,
      `"Mastering Flue: A Deep Dive into Agent Architectures"`,
      `"Building Scalable AI: Lessons from Production Flue Deployments"`,
    ];
    // Simulate a delay for async operation
    await new Promise(resolve => setTimeout(resolve, 500));
    return `Generated ideas for "${topic}":\n- ${ideas.join('\n- ')}`;
  }
}

// Our second skill: OutlineCreationSkill

export class OutlineCreationSkill implements AgentSkill {
  name = "create_outline";
  description = "Creates a detailed blog post outline for a given idea.";

  async execute(idea: string): Promise<string> {
    console.log(`[Skill: ${this.name}] Creating outline for idea: "${idea}"`);
    if (!idea || idea.trim() === "") {
      throw new Error("Idea cannot be empty for outline creation.");
    }
    // Simulate a more complex outline generation
    // In production, this might involve another LLM call with a specific
```

```
system prompt.
    const outline = `Outline for "${idea}":
      1. Introduction: Hook, problem statement.
      2. Core Concepts: Key terms, explanations.
      3. Practical Application: Code examples, walkthrough.
      4. Best Practices: Tips for production.
      5. Conclusion: Summary, future outlook.`;
    await new Promise(resolve => setTimeout(resolve, 800));
    return outline;
  }
}
```

Explanation:

- We define an `AgentSkill` interface, ensuring all skills have a `name`, `description`, and an `execute` method. The `description` is crucial as it will be fed to the LLM so it knows when to use the skill.
- `IdeaGenerationSkill` and `OutlineCreationSkill` implement this interface.
- Each `execute` method now includes `console.log` statements for basic observability and a simulated delay (`await new Promise...`).
- Crucially, we've added basic input validation and `throw new Error()` for invalid inputs within the skills. This is our first step towards robust error handling.

Step 2: Integrate Skills into an Agent

Now, let's update our main agent to use these modular skills. This agent will need to be aware of the skills and use its LLM to decide which one to call.

Modify your `src/agent.ts` file:

```
// src/agent.ts
import { Agent, AgentState, AgentConfig, AgentRouteHandler } from '@flue/
core';
import { IdeaGenerationSkill, OutlineCreationSkill, AgentSkill } from './
skills';
import { ClaudeCode } from '@flue/llms'; // Or your preferred LLM

// ⚡ Quick Note: Using ClaudeCode as a placeholder. Replace with your actual
LLM setup.
// Ensure you have CLAUDE_API_KEY or equivalent env var set.
const llm = new ClaudeCode({
  apiKey: process.env.CLAUDE_API_KEY ||
'YOUR_CLAUDE_API_KEY', // Replace with your actual API key or env var
});

export class ContentAgent extends Agent {
  // Store our skills as a map for easy lookup
  private skills: Map<string, AgentSkill>;

  constructor(config?: AgentConfig) {
```

```
    super(config);

    // Initialize our skills
    this.skills = new Map<string, AgentSkill>();
    const ideaGenSkill = new IdeaGenerationSkill();
    const outlineSkill = new OutlineCreationSkill();

    this.skills.set(ideaGenSkill.name, ideaGenSkill);
    this.skills.set(outlineSkill.name, outlineSkill);

    // Provide skill descriptions to the LLM for tool calling
    const skillDescriptions = Array.from(this.skills.values()).map(skill => ({
      name: skill.name,
      description: skill.description,
      parameters: {
        type: "object",
        properties: {
          input: { type: "string", description: "The main input for the
skill." }
        },
        required: ["input"]
      }
    }));

    this.llm = llm; // Assign the LLM
    this.tools = skillDescriptions; // Inform the LLM about available tools
  }

  protected async handleMessage(message: string, state: AgentState): Promise<s
tring> {
    console.log(`[Agent] Received message: "${message}"`);

    // The agent's core decision-making loop
    // This is where the LLM decides which tool to call or what to say.
    const agentResponse = await this.llm.chat(
      [
        { role: 'system', content: `You are a helpful content generation
assistant. You can generate blog post ideas and create outlines for them.
        Use the available tools to fulfill user requests.
        Current session state: ${JSON.stringify(state.getAll())}` },
        { role: 'user', content: message },
      ],
      { tools: this.tools } // Pass the tools to the LLM for function calling
    );

    // Check if the LLM decided to call a tool
    if (agentResponse.toolCalls && agentResponse.toolCalls.length > 0) {
      const toolCall = agentResponse.toolCalls[0]; // Assuming one tool call
for simplicity
      const skill = this.skills.get(toolCall.name);

      if (skill) {
        console.log(`[Agent] Calling skill: "${toolCall.name}" with input: "$
{toolCall.parameters.input}"`);
        try {
          const skillOutput = await skill.execute(toolCall.parameters.input);
          state.set('last_skill_output', skillOutput); // Store output in
session state
          // After skill execution, we might want the LLM to process the
output or continue.
          // For this example, we'll return the skill output directly.
          // In a more advanced agent, you'd feed this back to the LLM to
```

```
decide the next step.
          return `Skill "${toolCall.name}" executed successfully:\n${skillOutp
ut}`;
        } catch (error: any) {
          console.error(`[Agent] Error executing skill "${toolCall.name}":`, e
rror.message);
          return `I encountered an error while trying to ${toolCall.name}.
Please try again or rephrase your request. Error: ${error.message}`;
        }
      } else {
        console.warn(`[Agent] LLM requested unknown skill: ${toolCall.name}`);
        return `I don't have a skill called "${toolCall.name}".`;
      }
    } else if (agentResponse.content) {
      // If no tool call, the LLM generated a direct response
      return agentResponse.content;
    } else {
      return "I'm not sure how to respond to that.";
    }
  }
}

// Export the AgentRouteHandler for deployment
export const handler = new AgentRouteHandler(new ContentAgent());
```

Explanation:

- We import our `AgentSkill` interface and the concrete `IdeaGenerationSkill` and `OutlineCreationSkill`.
- The `ContentAgent` now has a `skills` map to hold instances of our skills.
- In the constructor, we initialize these skills and populate the `this.tools` array with their names and descriptions. This `tools` array is how Flue (and the underlying LLM) knows about the capabilities of your agent. The `parameters` object tells the LLM the expected input structure for each tool.
- The `handleMessage` method is where the agent's intelligence lives. It sends the user's message and the available `tools` to the LLM.
- **Error Handling:** The `try...catch` block around `skill.execute()` is crucial. If a skill throws an error (as our `IdeaGenerationSkill` does for empty topics), the agent catches it, logs it, and returns a user-friendly error message instead of crashing.
- We store the `last_skill_output` in the `AgentState` for potential future use by the agent.

Step 3: Implement Basic Error Handling in a Skill (Already done above, but emphasizing it)

Notice how in `src/skills/index.ts`, we added:

```
    if (!topic || topic.trim() === "") {
      throw new Error("Topic cannot be empty for idea generation.");
    }
```

And similarly for `OutlineCreationSkill`. This is a simple yet powerful form of defensive programming. The skill itself validates its inputs before attempting its core logic. This prevents the agent from passing invalid data to potentially expensive or fragile external services.

Step 4: Add Logging for Agent Decisions and Tool Use

We've already started adding `console.log` statements throughout the agent and skills. This is the simplest form of logging.

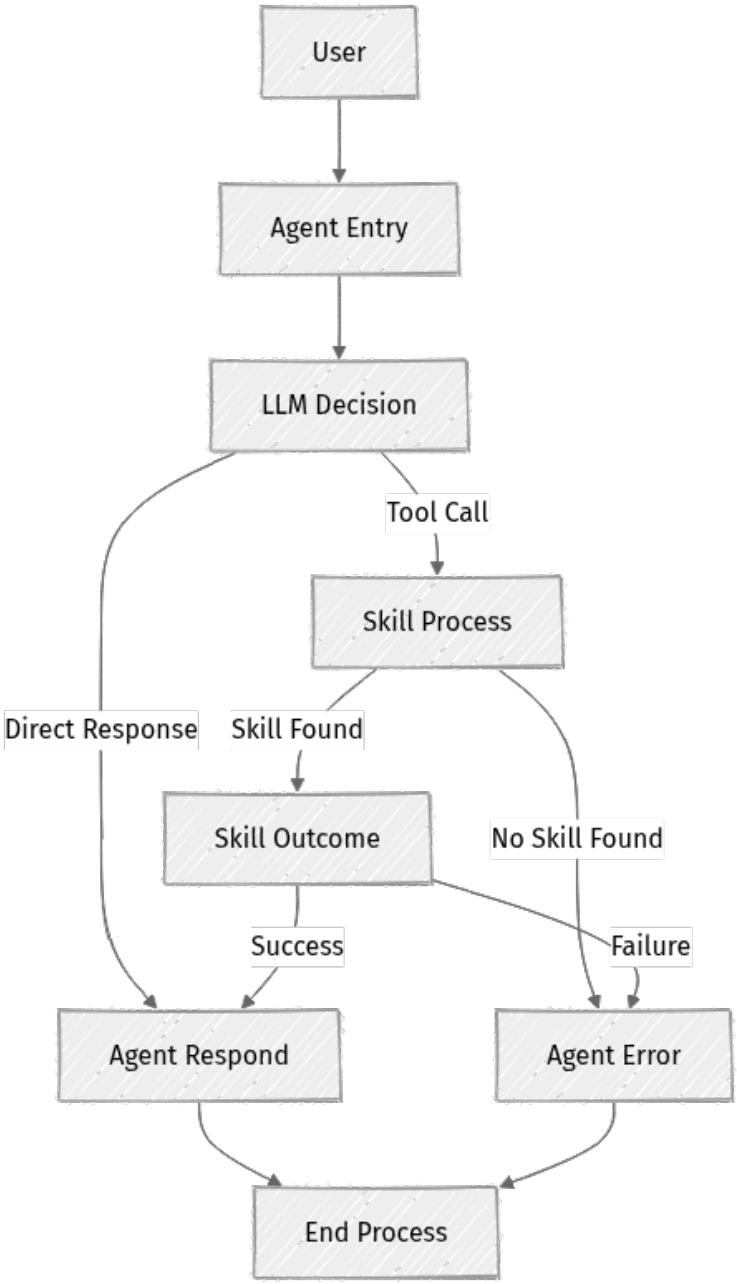
```
// Example from src/skills/index.ts
console.log(`[Skill: ${this.name}] Generating ideas for topic: "${topic}"`);

// Example from src/agent.ts
console.log(`[Agent] Received message: "${message}"`);
console.log(`[Agent] Calling skill: "${toolCall.name}" with input: "$
{toolCall.parameters.input}"`);
console.error(`[Agent] Error executing skill "${toolCall.name}":`, error.messa
ge);
```

While `console.log` is convenient for development, for production, consider a more structured approach. For Cloudflare Workers, `console.log` messages are automatically captured and visible in your Worker logs. For Node.js deployments, a library like `pino` or `winston` would allow you to log structured JSON, which is easier for log aggregation and analysis tools (like Splunk, Elastic Stack, or cloud-native logging services) to process.

Visualizing the Agent's Flow

Let's visualize the improved flow of our modular agent with error handling:



Explanation: This diagram shows how the `AgentRouteHandler` directs user input to the `ContentAgent`. The agent then uses the LLM to make a decision: either generate a direct response or call one of its defined skills. Crucially, if a skill is called, there are distinct paths for success and failure, ensuring that errors are caught and handled by the agent before a response is sent back to the user.

Mini-Challenge

Now it's your turn to enhance the robustness of our agent!

Challenge: Modify the `OutlineCreationSkill` in `src/skills/index.ts`. Add a new validation rule: if the `idea` input string is too short (e.g., less than 10 characters), throw a specific error, for example, `"Idea too short to create a meaningful outline."`.

Hint: Locate the `execute` method within the `OutlineCreationSkill` class. Use an `if` statement to check the `idea.length` before proceeding with the outline generation logic.

What to observe/learn: After implementing the change, try sending a very short idea to your agent. Observe how the `ContentAgent` catches this specific error from the skill and returns your custom error message to you, rather than attempting to generate a nonsensical outline or crashing. This demonstrates how fine-grained error handling in skills leads to a more resilient overall agent.

Common Pitfalls & Troubleshooting

Even with best practices, developing agents can present unique challenges. Here are a few common pitfalls and how to troubleshoot them:

1. **Infinite Loops / Hallucinations:**
    - **Pitfall:** The agent repeatedly calls the same skill with slightly different (or identical) inputs, gets stuck in a loop, or generates nonsensical output.
    - **Troubleshooting:**
        - **Detailed Logging:** Review the agent's logs (especially LLM prompts and responses) to understand why it's making repetitive decisions. Is the system prompt unclear? Is a tool description ambiguous?
        - **Prompt Engineering:** Refine your system prompt. Add explicit instructions for when to stop calling tools or to summarize and respond.
        - **Guardrails:** Implement explicit checks in your `handleMessage` method to detect and break loops (e.g., a counter for maximum tool calls per turn).
        - **Tool Descriptions:** Ensure tool descriptions are precise, and their expected outputs are clear.

2. **State Management Issues (Lost Context, Inconsistent Behavior):**
    - **Pitfall:** The agent forgets previous interactions, behaves inconsistently across turns, or fails to retrieve/save persistent data.
    - **Troubleshooting:**
        - **Inspect Session State:** Log the contents of `agent.state` at critical points in your `handleMessage` method. What's being stored? What's missing?
        - **Serialization/Deserialization:** If using persistent state, ensure your data is correctly serialized to and deserialized from your chosen storage (e.g., JSON.stringify/parse).
        - **Concurrency:** If multiple users interact with the same agent instance, ensure session state is isolated per user. Flue's `AgentRouteHandler` handles this by default, but custom state management needs careful consideration.
3. **Deployment-Specific Quirks (Cold Starts, Resource Limits):**
    - **Pitfall:** Your agent works perfectly locally but is slow, crashes, or hits resource limits when deployed (e.g., on Cloudflare Workers).
    - **Troubleshooting:**
        - **Cold Starts:** Initial requests to a serverless function (like a Cloudflare Worker) can be slow as the environment "spins up." This is inherent to serverless. Optimize your agent's initialization to be as lean as possible.
        - **Resource Limits:** Cloudflare Workers have CPU time and memory limits.
            - **Optimize Dependencies:** Minimize the number and size of external libraries.
            - **Asynchronous Operations:** Ensure long-running tasks are truly `await`ed and don't block the event loop.
            - **Memory Usage:** Avoid loading very large models or datasets directly into the Worker's memory. Consider external services or streaming.
        - **Platform Monitoring:** Use the monitoring tools provided by your deployment platform (e.g., Cloudflare's dashboard) to view logs, CPU usage, and memory consumption.

🔥 **Optimization / Pro tip:** For Cloudflare Workers, a common pattern to mitigate cold starts for LLM clients is to initialize the LLM client outside the `AgentRouteHandler`'s constructor, but within the global scope of the Worker, allowing it to be reused across invocations.

Summary

Building production-ready Flue agents means moving beyond basic functionality to embrace robust software engineering principles. In this chapter, we've explored:

- **Modular Agent Design:** Breaking down complex agent logic into smaller, reusable `AgentSkill` units for improved readability, testability, and maintainability.
- **Advanced State Management:** Understanding when to use ephemeral session state versus persistent storage for long-running, complex agent interactions.
- **Robust Error Handling:** Implementing defensive programming within skills and comprehensive `try...catch` blocks within the agent to gracefully handle failures and provide clear feedback.
- **Observability:** Emphasizing the importance of logging agent decisions, LLM interactions, and skill executions to enable effective debugging and performance monitoring.

By applying these best practices, you're not just building agents; you're crafting reliable, scalable components that can stand up to the demands of real-world AI products. The journey of agent development is iterative. Continuously monitor your agents in production, analyze their behavior, and refine their design based on real-world usage.

References

- Flue — The Agent Harness Framework
- withastro/flue: The sandbox agent framework - GitHub
- flue/docs/deploy-cloudflare.md at main · withastro/flue - GitHub
- Cloudflare Workers Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.