GAI-2104 Lab Guide

Building Agentic AI with Model Context Protocol

ascendient learning

Ascendient, LLC
1 California Street Suite 2900
San Francisco, CA 94111
https://www.ascendientlearning.com/

USA: 1-877-517-6540, email: getinfousa@ascendientlearning.com
Canada: 1-877-812-8887 toll free, email: getinfo@ascendientlearning.com

Table of Contents

Setting up GitHub Copilot in VS Code

MODULE 1

- ✓ Install and configure GitHub Copilot in Visual Studio Code
- ✓ Verify your GitHub Copilot subscription status
- ✓ Test Copilot’s basic functionality

1.1. Introduction

This lab will guide you through the process of installing and configuring GitHub Copilot in Visual Studio Code. GitHub Copilot is an AI-powered code completion tool that helps developers write code faster and with fewer errors.

By the end of this setup, you'll have a working GitHub Copilot installation and understand how to verify its functionality.

Whether you're new to AI-assisted development or looking to enhance your coding workflow, this guide will ensure you're ready to leverage Copilot's capabilities in your projects.

1.2. Setting up GitHub Copilot

1.2.1. Step 1: Installing the GitHub Copilot Extension

1. Open Visual Studio Code
2. Navigate to the Extensions view:
    - Click the Extensions icon in the Activity Bar on the left side of VS Code, or
    - Use the keyboard shortcut: `Ctrl` + `Shift` + `X` (Windows/Linux) or `Cmd` + `Shift` + `X` (macOS)

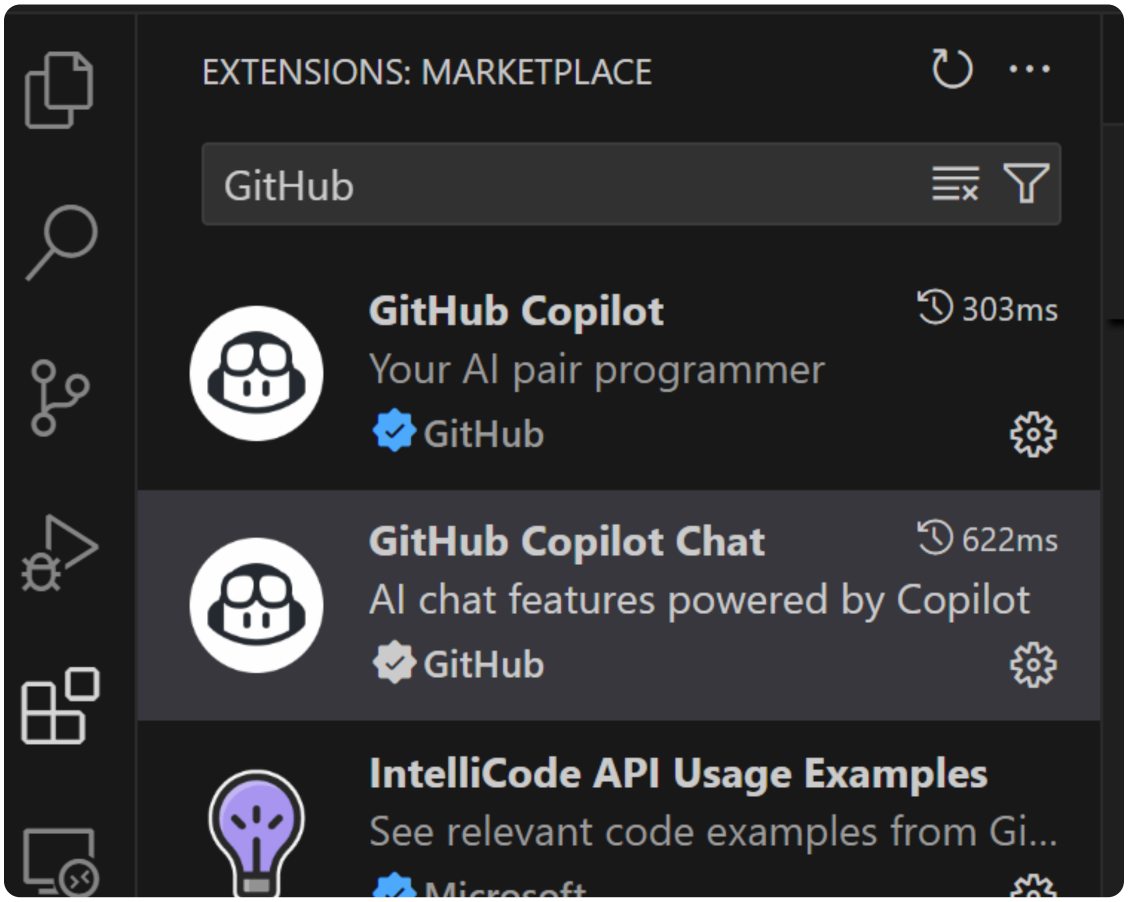


Figure 1. VS Code Extensions View

3. In the Extensions search box, type “GitHub Copilot”
4. Look for the official GitHub Copilot extension and click “Install”

1.2.2. Step 2: Authentication

1. After installation, you’ll be prompted to sign in to GitHub
2. Click “Sign in to GitHub”
3. Your default web browser will open to GitHub’s authentication page
4. Sign in with your GitHub credentials
5. Authorize VS Code to access GitHub Copilot
6. Return to VS Code once authentication is complete

1.2.3. Step 3: Verifying Your Setup

1. Create a new Python file named `test_copilot.py`
2. Press `Ctrl` + `I`.
3. Type: "Write a function that takes a number and returns the factorial of that number"
4. Press Enter and wait for Copilot to suggest code

1.3. Conclusion

This lab covered the essential steps for setting up GitHub Copilot in Visual Studio Code. You've installed the extension, authenticated with GitHub, and verified that Copilot is working correctly. With these fundamentals in place, you're now equipped to explore more advanced features and integrate AI-assisted coding into your development workflow.

1.3.1. Additional Resources

- GitHub Copilot documentation: https://docs.github.com/en/copilot
- VS Code documentation: https://code.visualstudio.com/docs
- Troubleshooting guide: https://docs.github.com/en/copilot/how-tos/troubleshoot-copilot/troubleshoot-common-issues

MCP Architecture Explorer

MODULE 2

- ✓ Understand the fundamental MCP architecture components (host, client, server)
- ✓ Install and configure the MCP Inspector for protocol observation
- ✓ Analyze real-time JSON-RPC message flows between clients and servers
- ✓ Explore capability negotiation and connection lifecycle patterns
- ✓ Debug common MCP connection and communication issues

2.1. Introduction

The Model Context Protocol (MCP) enables AI applications to connect with external data sources and tools through a standardized architecture. Understanding how MCP components interact—and being able to observe this interaction in real-time—is essential for building robust agentic applications.

In this lab, you'll use the MCP Inspector, a professional debugging tool, to observe live protocol communication between clients and servers. You'll see JSON-RPC messages flowing in real-time, watch capability negotiation happen, and understand the complete connection lifecycle that makes MCP work.

2.1.1. The Scenario

Your development team is building a travel planning AI assistant that needs access to weather information. The team has created a Weather Information MCP server that provides current conditions, forecasts, and alerts through a standardized protocol. As the integration lead, you need to understand exactly how this server communicates with AI clients.

Before deploying the weather server to production, you want to observe its protocol behavior in detail. You'll use the MCP Inspector to examine how the server exposes its tools (weather data retrieval), resources (location information), and prompts (weather report templates). You also have a simple calculator server for comparison to understand how different server types behave.

Your goal is to validate that the weather server follows proper MCP protocol patterns, handles errors gracefully, and provides the rich data structure that the AI assistant needs for travel recommendations.

By the end of this lab, you'll have hands-on experience with the protocol layer that powers all MCP applications, giving you the foundation to build and debug your own MCP integrations.

2.2. Getting Started

1. **Open Your Terminal**
   Open a Terminal in the `LabFiles/mcp-architecture` lab directory.

2. **Install Required Libraries**
   Create a virtual environment, activate it, and install the required libraries:

```
uv venv --seed --python=3.13
.\.venv\Scripts\activate
pip install fastmcp==2.12.4
```

2.3. Core

2.3.1. Step 1: Reviewing the weather server implementation

1. **Understand the lab files**
   Open `LabFiles/mcp-architecture/weather_server.py`. This file contains the MCP server code you'll be analyzing. In order to connect to this server, you'll need an MCP host which will create a client.

   flowchart LR Host ←→ Client ←→ Server

2. **Examine the server code**
   Review the `weather_server.py` code to understand how it implements MCP primitives:

   - `mcp = FastMCPServer(…)` - Initializes the MCP server instance
   - `@mcp.tool(…)` - Defines MCP tools for weather data retrieval
   - `@mcp.resource(…)` - Sets up MCP resources for location data
   - `@mcp.prompt(…)` - Creates MCP prompts for weather reports
   - `mcp.run(…)` - Starts the MCP server with HTTP transport

3. **Identify server capabilities**
   Review tools, resources, and prompts provided. Once you feel that you understand the server's functionality, proceed to start it up for validation.

2.3.2. Step 2: Starting the server for validation

1. **Launch the Weather Server**
   In the terminal, run the following command to start the weather server:

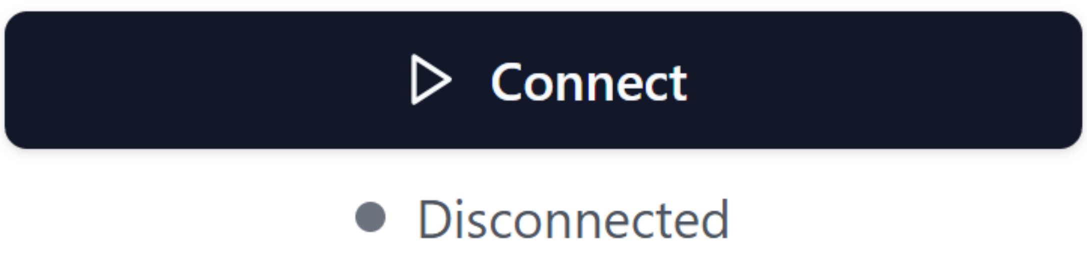
   ```
   fastmcp dev weather_server.py
   ```

2. **Verify server startup**
   The browser should open automatically. If it doesn't, look for the URL in the terminal output which includes the MCP_PROXY_AUTH_TOKEN. Open that URL in your browser to access the server's MCP endpoint.

   This is the MCP Host that you will use to connect to the weather server. It doesn't have any clients. In fact, it doesn't even have an AI connection! The inspector is simply a useful utility for testing MCP servers.

2.3.3. Step 3: Connecting to the MCP Server

1. **Connect to the MCP Server**
   In the lower left of the sidebar, there is a connect button. Click it to connect to the weather server.



   In the middle bottom of the screen, there is a "History" window. You should see a message named `initiation`. Explore this message to understand how the connection process works. At each step in the lab, you can look here to see the messages exchanged between the client and server.

Example Message

```
{
  "method": "notifications/message",
  "params": {
    "level": "info",
    "logger": "stdio",
    "data": {
      "message": "[<timestamp>] INFO     Starting MCP server"
    }
  }
}
```

2. **Navigate the Inspector interface**
   The Inspector opens with several key tabs:
   - **Connection** - Shows server connection status and protocol information
   - **Tools** - Lists available server tools with their descriptions and schemas
   - **Resources** - Shows server resources that provide read-only data access
   - **Prompts** - Displays server prompt templates for AI interactions

2.3.4. Step 4: Validating server protocol behavior

1. **Accessing a Resource**
   MCP Resources are read-only data endpoints. They differ from tools which can perform actions and accept parameters. To explore resources:
   - Click the **Resources** tab.
   - Click **List Resources** to see available resources. You should see a single resource named `list_locations`.
   - Click **`list_locations`** to view the resource data. In the right-hand column, you should see a JSON response listing supported weather locations.

This took two requests to retrieve the data. First, a call to `resources/list` to get the resource metadata, then a call to `resource/read` to fetch the actual data.

2. **Review the Tools tab**
   Select the button:[Tools] tab. Click **List Tools**]. You should see three tools exposed by the server. These tools were listed using the call `tools/list`. You should see three tools exposed by the server: `get_current_weather`, `get_forecast`, and `get_weather_alerts`.

3. **Test the `get_current_weather` tool**
   Let's test the tool that retrieves current weather data.

   - Click **`get_current_weather`** to open the parameter form.
   - Enter one of the locations that was listed by the locations resource. (e.g., "seattle")
   - Click **Run Tool** to execute the tool.
   - Observe the JSON response containing temperature, conditions, humidity, and timestamp.

     ```
     {
       "temp": 52,
       "condition": "Partly Cloudy",
       "humidity": 68,
       "location": "Seattle",
       "updated": "<timestamp>"
     }
     ```

     This response was returned via a `tools/call` request. *Save this repsonse for later use in the Prompts tab.*

4. **Test error handling**
   Validate the server's error responses by testing invalid inputs (e.g., "invalid-city"). The system should return an error message indicating the location was not found, along with a list of available locations.

5. **Test a prompt**
   Next, let's test how a prompt works.

   - Click **Prompts** to open the Prompts tab.

  - Click **List Prompts** to see available prompts. You should see a prompt named `weather_report`. This was retrieved via the `prompts/list` call.
  - Click **`weather_report`** to open the prompt details.
  - Enter the `location` parameter (e.g., "Seattle") and the `current_conditions` parameter using the JSON response you saved earlier from the `get_current_weather` tool.
  - Click **Get Prompt**. Inside `messages[0].content.text`, you should see a formatted weather report based on the template. This was generated via the `prompts/get` call.

2.4. Challenge 1: Message Exchange (Optional)

Consider the following series of actions exchanged with the server.

- Get current weather for Chicago.
- Get the a 7-day forecast for Chicago.
- Get any weather alerts for Chicago.
- Generate a weather report prompt for Chicago using the current conditions.

Make a list of the messages that you would expect to see exchanged between the client and server for each action. Identify the method names, parameters, and expected responses. You don’t need to write out every detail, just the high-level flow of requests and responses.

After you’re done, run the steps in order and compare your expected messages with the actual messages observed in the MCP Inspector.

2.5. Challenge 2: Open with Fast-Agent (Optional)

In this challenge, you'll connect the weather server to a simple AI agent using Fast-Agent.

1. **Install Fast-Agent**
   Install Fast-Agent in your terminal:

   ```
   pip install fast-agent-mcp==0.3.15
   ```

2. **Add your OpenAI API Key**
   Set your OpenAI API key in the `fastagent.secrets.yaml` files located in the lab directory.

3. **Review the MCP configuration**
   Open `fastagent.config.yaml` in the lab directory. This file configures Fast-Agent to connect to the weather server. `fast-agent` will launch the MCP server automatically based on the settings in this file.

4. **Launch the Fast-Agent client**
   In a new terminal window, run the following command to start the Fast-Agent client that connects to the weather server:

   ```
   fast-agent run go --servers=weather
   ```

5. **Interact with the AI agent**
   In the Fast-Agent interface, enter a prompt such as "What's the weather like in Chicago today?" The agent will use the weather server to retrieve current conditions and generate a response.

2.6. Conclusion

You've successfully explored MCP architecture using the Inspector tool:

- MCP Inspector mastery - You can now use the professional debugging tool to examine any MCP server

- Protocol understanding - You observed JSON-RPC message flows for resources, tools, and prompts in action
- Server validation skills - You tested server functionality, error handling, and data structures through the Inspector interface
- MCP primitives knowledge - You understand how tools, resources, and prompts work together in a real server implementation

You now have the foundation to debug MCP integrations, validate server implementations, and understand the protocol layer that powers all MCP applications.

Building MCP Resources with FastMCP

MODULE 3

- ✓ Understand what MCP resources are and how they provide read-only data access to LLMs
- ✓ Create dynamic resources using the `@mcp.resource` decorator with database integration
- ✓ Implement template parameters with single and multiple URI path parameters
- ✓ Test resources using the MCP Inspector development tool

- ✓ Think like an AI agent to identify essential missing resources
- ✓ Build practical inventory management resources with SQLite integration

3.1. Introduction

In this hands-on lab, you'll learn how to build MCP resources using FastMCP. Resources are a fundamental component of the Model Context Protocol—they provide read-only access to data that LLMs and client applications can discover and use.

Think of resources as information sources: files, database content, configuration data, or any dynamically generated information that's relevant to a conversation. When a client requests a resource URI, FastMCP finds the corresponding definition, executes it if it's dynamic, and returns the content to the client.

By the end of this lab, you'll understand how to create both simple and sophisticated resources that make your data accessible to AI assistants through the standardized MCP protocol.

3.1.1. The Scenario

You're building an MCP server for an inventory management system. The system has a SQLite database containing information about items across different categories like Electronics, Books, Furniture, and Clothing. Each item has details like name, quantity, price, and description.

Your goal is to expose this inventory data through MCP resources so AI agents can:

- Get overall inventory summaries
- Browse items by category
- Access detailed item information
- Discover what categories exist in the system

This scenario demonstrates how MCP resources transform static databases into dynamic, conversational interfaces that AI assistants can intelligently query and understand.

3.1.2. Understanding MCP Resources

Resources in MCP serve a specific purpose: **providing context and data to LLMs**. Unlike tools (which perform actions), resources are read-only—they supply information that helps the AI understand the domain and answer questions intelligently.

The diagram below shows how the MCP resource protocol works:

```
sequenceDiagram
    participant Client
    participant Server

    Note over Client,Server: Resource Discovery
    Client->>Server: resources/list
    Server-->>Client: List of resources

    Note over Client,Server: Resource Access
    Client->>Server: resources/read
    Server-->>Client: Resource contents

    Note over Client,Server: Subscriptions (Optional)
    Client->>Server: resources/subscribe
    Server-->>Client: Subscription confirmed

    Note over Client,Server: Updates (Optional)
    Server--)Client: notifications/resources/updated
    Client->>Server: resources/read
    Server-->>Client: Updated contents
```

As shown above, MCP defines a standardized protocol for resource discovery (`resources/list`), access (`resources/read`), and optional subscription-based updates—all using JSON-RPC messages.

3.1.3. About FastMCP

This lab uses FastMCP, a Python library that simplifies MCP server implementation. While MCP itself operates through JSON-RPC messages, FastMCP provides a decorator-based approach that handles the protocol details for you.

When you write:

```
@mcp.resource("inventory://category/{cat}")
async def get_category_summary(cat: str):
    return {"category": cat, "count": 100}
```

FastMCP automatically: - Registers this as a discoverable resource template - Handles JSON-RPC `resources/list` and `resources/read` requests - Validates parameters and manages error responses - Serializes your return values to proper MCP content types

3.2. Getting Started

1. **Open Your Terminal**
   Open a Terminal in the `LabFiles/resource-integration` lab directory.

2. **Install Required Libraries**
   Create a virtual environment, activate it, and install the required libraries:

   ```
   uv venv --seed --python=3.13
   .\.venv\Scripts\activate
   pip install fastmcp==2.12.4 mcp[cli]==1.18.0 fast-agent-mcp==0.3.15
   ```

3. **Initialize the Database**
   The `init_db.py` script is provided in your course files. It sets up the `inventory.db` SQLite database with the required schema and sample data. To initialize the database, run:

   ```
   python init_db.py
   ```

   This will create the `items` table with fields: `id`, `name`, `quantity`, `price`, `category`, `description`, and `last_updated`, and populate it with sample items across several categories.

3.3. Core

3.3.1. Step 1: Review the Server Foundation

1. **Open the Server File**
   Open `server.py` in your code editor. Take a moment to review the starter code provided.

   Key concepts in this setup:
   - `FastMCP("InventoryServer")` creates your server instance with a name
   - `DB_FILE` points to the SQLite database initialized earlier
   - We import `Context` for accessing MCP request metadata
   - Several database functions are already implemented - your job is to expose them as MCP resources

3.3.2. Step 2: Expose Your First Dynamic Resource

1. **Add the Resource Decorator**
   Find the `get_inventory_summary()` function in `server.py`. This function already queries the database for inventory statistics. Your task is to expose it as an MCP resource by adding the decorator:

```
# Add this decorator above the function
@mcp.resource("inventory://summary")
async def get_inventory_summary() -> dict:
    # ... existing database code ...
```

   What's happening here:
   - The `@mcp.resource()` decorator registers this function as a discoverable resource
   - The URI `"inventory://summary"` is how clients will request this data
   - The function executes **only when requested**, not on server startup
   - Return type `dict` is automatically serialized to JSON by FastMCP
   - The database code is already implemented - you're just making it available via MCP

2. **Test the Summary Resource**
   Start your server with the MCP Inspector:

   ```
   fastmcp dev server.py
   ```

   The MCP Inspector will open in your browser automatically. Then:

   - Click **Connect** in the left sidebar
   - Navigate to the *Resources* tab
   - Click **List Resources** - you should see `inventory://summary` in the list
   - Click on `inventory://summary`
   - Click **get_inventory_summary**
   - You should see a JSON response with `total_items`, `total_categories`, `total_value`, and `timestamp`

     > Tip
     >
     > If you don't see the resource, check that you added the decorator correctly and restarted the server.

3. **Stop the Server**
   Press Ctrl+C in your terminal to stop the server before continuing to the next step.

3.3.3. Step 3: Create a Template Parameter Resource

Template parameters allow you to create parameterized resources that respond to different inputs. Instead of creating separate resources for each category, you can create one template that handles any category.

3.3.3.1. Add Template Parameter Decorator

1. **Add the Decorator**
   Find the `get_category_summary()` function in `server.py`. The database logic is already implemented. Add the decorator to expose it as a template resource:

   ```
   @mcp.resource("inventory://category/{cat}")
   async def get_category_summary(cat: str) -> str:
   # ... existing database code ...
   ```

Key concepts about template parameters:
- `{cat}` in the URI becomes a parameter in your function signature
- Clients can request `inventory://category/Electronics`, `inventory://category/Books`, etc.
- FastMCP automatically extracts the parameter value and passes it to your function
- This follows RFC6570 URI Template standards
- The function name `cat` must match the template parameter name `{cat}`

2. **Test the Category Stats Resource**
   Restart your server with the MCP Inspector:

```
fastmcp dev server.py
```

Then test the template resource:

- Click **Connect** in the left sidebar
- Navigate to the *Resources* tab
- Click **List Templates**
- Click **get_category_summary**
- Enter "Books" for the `cat` parameter
- Click **Read Resource**
- You should see statistics for the Books category
- Try different categories:
  - `inventory://category/Books`
  - `inventory://category/Furniture`
  - `inventory://category/Clothing`
  - `inventory://category/InvalidCategory` (should return "No items found")

> ⓘ Note
>
> Template resources don’t appear in the simple list - you need to type the full URI with a parameter value to test them.

3. **Stop the Server**
   Press Ctrl+C in your terminal before continuing.

3.3.4. Step 4: Add Multiple Template Parameters

Template resources can include multiple parameters in the URI path.

1. **Add Multi-Parameter Decorator**
   Find the `get_item_in_category()` function in `server.py`. The database logic is complete - add the decorator with multiple parameters:

```
# Add this decorator with TWO template parameters
@mcp.resource("inventory://category/{cat}/item/{item_id}")
async def get_item_in_category(cat: str, item_id: int) -> dict:
    # ... existing database code ...
```

> ⓘ Note
>
> FastMCP automatically converts the `item_id` parameter from a string to an `int` based on your function's type hint. The parameter names in the URI `{cat}` and `{item_id}` must match the function parameter names.

2. **Test Multi-Parameter Resource**
   Restart the server:

```
fastmcp dev server.py
```

   Test with multiple parameters:

   - Click **Connect**
   - Navigate to the *Resources* tab
   - Click **List Templates**
   - Click **get_item_in_category**
   - Enter "Electronics" for `cat` and "1" for `item_id`
   - Click **Read Resource**
   - You should see detailed information about item 1 in the Electronics category

     > Tip

> Pay attention to the error message when requesting an invalid item - notice how FastMCP converts Python exceptions into proper error responses.

3. **Stop the Server**
   Press Ctrl+C in your terminal.

3.3.5. Step 5: Consider Missing Resources for Agent Interactions

Think like an AI agent working with your inventory system. What information would be essential but is currently missing?

1. **Analyze Agent Needs**
   Consider this scenario: An AI agent wants to help a user find items in a specific category. Currently, the agent would need to guess category names or rely on trial and error. What resource would solve this problem?

   The answer: A resource that lists all available categories in the inventory system.

2. **Implement Categories Listing**
   Now you'll implement this resource yourself. Add a new function to your `server.py` file, right after the existing resource functions: Use this SQL query to get the data you need:

   ```sql
   SELECT DISTINCT category FROM items ORDER BY category
   ```

   > Tip
   >
   > Look at the existing functions for patterns on database connection, error handling, and return formatting.

3. **Test the Categories Resource**
   Restart your server:

   ```
   fastmcp dev server.py
   ```

   Test the new resource!

4. **Stop the Server**
   Press Ctrl+C in your terminal.

3.4. Challenge 1: Low Stock Alert Resource (Optional)

Inventory management systems need to alert managers when items are running low. Using the patterns you've learned, create a resource that lists items which are below a specified stock threshold. Use a URI such as: `inventory://alerts/low-stock/{threshold}`.

This SQL query will help you find low-stock items:

```sql
SELECT id, name, quantity, category, price
FROM items
WHERE quantity <= ?
ORDER BY quantity ASC, category
```

Make sure to test your resource with different threshold values to ensure it works correctly.

3.5. Challenge 2: File System Resource Server (Optional)

Create a new MCP server that exposes a directory and its files as resources. Consider the following resource URIs:

- **`files://directory/{path}`** - List directory contents
- **`files://file/{path}`** - Read file content
- **`files://info/{path}`** - Get file metadata

There is a directory named `sample-files` witch several subdirectories and files you can use for testing. It is AI-generated content for a fictional company named "BeanBotics" which makes robotic baristas.

3.6. Conclusion

You've successfully built an Inventory Management MCP server with FastMCP that exposes dynamic resources backed by a SQLite database. You learned to:

- Create dynamic resources with `@mcp.resource` decorators
- Use template parameters for flexible, parameterized resources
- Connect resources to databases for real-time data access
- Think like an AI agent to identify missing resources

Your inventory system now provides AI clients with discoverable, structured access to database content through the standardized MCP protocol.

What databases or data sources do you use in your projects? Consider how MCP resources could expose that data to AI assistants, enabling intelligent, context-aware interactions.

Building MCP Coding Prompts

MODULE 4

- ✓ Understand MCP prompts and how they provide dynamic text templates for AI assistants
- ✓ Create prompt templates with variable substitution and parameter handling
- ✓ Build prompts that enforce company coding standards and workflows
- ✓ Test prompts using the MCP Inspector and integrate with development workflows

- ✓ Design prompts that improve team consistency and productivity

4.1. Introduction

In this lab, you'll learn how to build MCP prompts that standardize development workflows and improve code quality. Prompts are dynamic text templates that provide context and guidance to AI assistants, helping developers follow best practices and company standards. Unlike tools that perform actions or resources that provide data, prompts generate contextual instructions that guide AI assistants in producing consistent, high-quality output.

We'll use real code from a sample application to demonstrate how prompts can catch bugs, enforce standards, and improve development workflows. You'll build prompts that any AI assistant can use to provide better guidance on debugging, code review, and documentation.

Think of this as creating a set of intelligent templates that capture your team's expertise and make it available to any developer working with AI assistants. By the end, you'll understand how to design prompts that improve consistency and catch issues before they become problems.

4.1.1. The Scenario

You're working with **BeanBotics**, a company that makes robotic coffee systems. They have a Flask-based ordering system that manages coffee orders, but the development team has been struggling with inconsistent code quality and hard-to-find bugs.

Your task is to create MCP prompts that help developers:

1. **Debug Issues** - The ordering system has a subtle pricing bug where displayed prices don't match charged amounts
2. **Enforce Code Standards** - Database connections aren't using proper context managers
3. **Improve Documentation** - Functions lack proper docstrings and type hints
4. **Generate Tests** - The system needs comprehensive test coverage

The BeanBotics ordering system will serve as your test case—you'll create prompts that can identify its issues and guide developers toward better solutions.

4.2. Getting Started

1. **Open Your Terminal**
   Open a Terminal in the `coding-prompts` lab directory.

2. **Install Required Libraries**
   Create a virtual environment, activate it, and install the required libraries:

   ```
   uv venv --seed --python=3.13
   .\.venv\Scripts\activate
   pip install fastmcp==2.12.4 mcp[cli]==1.17.0 flask==2.3.3
   ```

3. **Explore the BeanBotics Ordering System**
   The lab includes a sample Flask application in the `beanbotics-orders` directory. This is a very simplistic system that has several common issues your prompts will help identify and fix.

   ```
   cd beanbotics-orders
   python app.py
   ```

   Visit http://localhost:5000 to see the ordering system in action. Try placing a few orders and notice the pricing. Click the "debug menu" link to see order history.

   > **Note**
   >
   > Keep this application running as you'll be analyzing its code throughout the lab. The code should automatically refresh when you make changes. If it doesn't, restart the server. Also, if you can't get it running, that is okay—you can still complete the lab by reviewing the code files.

4.3. Core

4.3.1. Step 1: Creating Your MCP Prompts Server and Connecting to VS Code

1. **Explore the Prompt Templates**
   The lab includes pre-written prompt templates in the `prompts` directory. Each prompt is stored as a markdown file with specific formatting. Let's examine one:

   Open `prompts/document_function.prompt.md` and review its structure:

   Key concepts in this prompt template:
   - Template variables are marked with `{variable_name}` syntax
   - The prompt provides clear instructions and expected sformat
   - Multiple parameters can be used: `{function_code}` and `{function_name}`
   - The text guides the AI toward specific, actionable outputs

2. **Create the Server File**
   Create a new file called `prompts_server.py` in the labs directory with this basic structure:

```python
from fastmcp import FastMCP
import os

# Initialize the FastMCP server
mcp = FastMCP("CodingPromptsServer")

# Directory containing prompt template files
PROMPTS_DIR = os.path.join(os.path.dirname(__file__), "prompts")

# TODO: Add code to load and register prompts from files
if __name__ == "__main__":
    mcp.run(transport="http")
```

3. **Implement Prompt Loading**
   Your task is to write a function that:

   - Reads the `document_function.prompt.md` file from the prompts directory

- Registers it as an MCP prompt using the `@mcp.prompt()` decorator
- Accepts parameters that match the template variables (`function_code` and `function_name`)
- Replaces the template variables with actual parameter values using `.format()`

HINT: The prompt function should look something like:

```
@mcp.prompt()
async def document_function(function_code: str, function_name: str = "") -> s
tr:
    """Generate documentation standards for Python functions."""
    # Read the template file
    # Replace {function_code} and {function_name} with actual values
    # Return the formatted prompt text
    pass
```

4. **Start Your Server and Connect**
   Start your MCP server:

```
.\.venv\Scripts\activate
python prompts_server.py
```

5. **Configure VS Code to Connect to Your Server**

- Open the `coding-prompts` directory in VS Code.
- Press `Control` + `Shift` + `P`
- Type to filter and select `MCP: Add Server…`
- Select "HTTP (HTTP or Server-Sent Events)"
- Enter the server URL: `http://127.0.0.1:8000/mcp`
  - This should be the URL, but you can verify it from the terminal output when you start your MCP server.
- Provide a name like "CodingPromptsServer"
- Make it available locally.

This process will open `mcp.json` and show the configuration options for your MCP server. It should look something like this:

```
"CodingPromptsServer": {
    "url": "http://127.0.0.1:8000/mcp",
    "type": "http"
}
```

At this point, the terminal running your MCP server should show some activity as VS Code connects.

6. **Test the Documentation Prompt in VS Code**

    - Open GitHub Copilot Chat in VS Code.
    - Also open the BeanBotics `app.py` file in VS Code. Opening the file will add the file context to Copilot Chat.
    - Type the following in Copilot Chat: `/mcp`. You should see your `document_function` prompt listed. Select it.
    - This will open a prompt input box where you can provide parameters. Enter `get_menu_items` and submit.

      You should see the prompt being generated based on your template.

7. **Verify the Prompt Works Correctly**
   Run the prompt. The `get_menu_items` function in `app.py` should be analyzed and documented.

   If the prompt doesn't work as expected, check that:

    - Your template file is being read correctly
    - Parameter substitution is working (check for unreplaced `{variable_name}` markers)
    - The server is running without errors

4.3.2. Step 2: Add Code Review Checklist Prompt

The BeanBotics code has several patterns that could be improved, including database connection handling. Now you'll add a second prompt that performs systematic code review focusing on Python best practices.

1. **Review the Code Review Template**
   Open `prompts/code_review_checklist.prompt.md` and examine its structure. Notice how it uses `{code_snippet}` and `{focus_area}` as template variables.

2. **Implement the Code Review Prompt**
   Following the same pattern you used for `document_function`, add a new prompt function to `prompts_server.py`.

   - **Expand the Focus Area**
     Map the `focus_area` parameter to a description. For instance:
   - "database" → "Database operations, connection handling, and SQL practices"
   - "error_handling" → "Exception handling, error messages, and recovery"
   - "security" → "Input validation, SQL injection, authentication"
   - "performance" → "Efficiency, caching, and resource usage"
   - "general" → "Overall code quality, style, and maintainability"
   - "default" → "General Python best practices and code quality"

3. **Test in VS Code with BeanBotics**
   Restart your MCP server to load the new prompt. If Copilot's chat box has a red button, click it and click **Start Server**. Test your new code review prompt!

4.3.3. Step 3: Add Debugging Assistant Prompt

The BeanBotics ordering system has a pricing bug—the displayed prices ($4.50, $5.50, $6.50) don't match the actual charged amounts. Let's add a debugging prompt that helps developers systematically investigate issues like this.

1. **Review the Debugging Template**
   Open `prompts/debug_assistant.prompt.md` and examine how it structures a systematic debugging approach with multiple template variables.

2. **Implement the Debugging Prompt**
   Add a new prompt function `debug_assistant` to `prompts_server.py`.

3. **Test the Debugging Prompt with BeanBotics Bug**
   Restart your server and test the debugging prompt.

4. **Use the Debugging Guidance**
   Follow the systematic approach provided by the prompt to identify the cause of the pricing discrepancy in the BeanBotics ordering system.

4.4. Challenge 1: BeanBotics Support Ticket Triage (Optional)

BeanBotics has expanded rapidly and now receives dozens of customer support tickets daily. The support team is struggling with:

- **Inconsistent Information Gathering**: Different agents collect different information, leading to multiple back-and-forth exchanges with customers
- **Poor Escalation Decisions**: Junior agents escalate too many simple issues, while sometimes missing critical ones that need immediate attention
- **Wasted Engineering Time**: The development team receives incomplete tickets and has to ask basic troubleshooting questions

The support manager wants to create standardized prompts that help agents:

1. Ensure they collect complete information before investigation

2. Make consistent escalation decisions based on clear criteria

Create an MCP prompts server that implements these two prompts to help BeanBotics support agents triage tickets effectively.

4.5. Challenge 2: Operational Runbook Generator (Optional)

After resolving several incidents, BeanBotics realizes they keep solving the same problems repeatedly. When new engineers join or incidents happen during off-hours, there’s no standardized documentation to help them respond quickly.

The DevOps team wants to create operational runbooks from resolved incidents so that:

1. Future responders have step-by-step resolution guides

2. Common issues are documented with diagnostic steps

3. Tribal knowledge is captured in a searchable format

4. New team members (human and AI!) can handle incidents independently

Create a prompt that converts incident reports into structured operational runbooks.

4.5.1. Sample Support Incident: "Can't Complete Order"

Use this resolved support ticket as your test case:

```
[2:34 PM] Maria Garcia:
Hi, I'm trying to order a medium latte but the website won't let me complete my order.
I keep clicking the "Place Order" button but nothing happens!

[2:36 PM] Support Agent (James):
Hi Maria! I'm sorry you're having trouble placing your order. Let me help you with
that.
Can you tell me:
1. What device/browser are you using?
2. Did you fill in your name in the customer name field?
3. Are you seeing any error messages?

[2:38 PM] Maria Garcia:
I'm on Chrome on my laptop. No error messages. And oh... I didn't put my name in.
Is that required? There's no asterisk or anything that says it's required.

[2:39 PM] Support Agent (James):
Ah, that's the issue! Yes, the customer name field is required to place an order, but
you're right - the form doesn't clearly indicate this. Let me walk you through:

1. Enter your name in the "Customer Name" field at the top of the order form
2. Select your drink (you mentioned medium latte - that's the "Latte" option with
"Medium" size)
3. Click "Place Order"

You should then see a success page with your order confirmation.

[2:41 PM] Maria Garcia:
Perfect! I just tried it with my name and it worked! Got order #1847.
Thanks so much! But yeah, you might want to make it clearer that the name is required.

[2:42 PM] Support Agent (James):
Wonderful! Your order is confirmed. And thank you for that feedback - I'm adding
that to our improvement list. Enjoy your latte!
```

4.6. Conclusion

In this lab, you've built an MCP prompts server that standardizes development workflows:

- **Documentation Standards**: Ensures consistent type hints and docstrings
- **Code Review Automation**: Systematic review focusing on Python best practices and database patterns
- **Debugging Assistance**: Structured approach to investigating and solving bugs using rubber duck debugging

Your prompts server demonstrates how MCP can encode team knowledge and best practices into reusable templates that any AI assistant can use. This approach helps maintain code quality, catches common issues early, and ensures consistent development practices across your team.

The BeanBotics example showed how prompts can identify real issues like missing context managers and calculation bugs, providing concrete value in software development workflows.

Building a Todo List MCP Server

MODULE 5

- ✓ Build an MCP server with tools for managing a todo list
- ✓ Implement stateful operations that persist across tool calls
- ✓ Create tools for adding, listing, and completing tasks
- ✓ Test your MCP server with Fast-Agent AI assistant
- ✓ Extend functionality with nested todos and YAML import features

5.1. Introduction

In this lab, you'll build a todo list that AI assistants can use to track their own work. When working on complex tasks, AI assistants need to remember what's been done and what's still pending—this MCP server gives them that capability.

You'll create tools that let AI assistants add tasks, view their list, and mark items complete. This is a practical example of how MCP tools maintain state and help AI manage multi-step workflows.

Think of this as building a memory system for AI—when working through complicated tasks like refactoring code or setting up infrastructure, the AI can break down the work into steps, track progress, and ensure nothing gets forgotten.

5.2. Getting Started

1. **Open Your Terminal**
   Open a Terminal in the `LabFiles/todo-list-tools` lab directory.

2. **Install Required Libraries**
   Create a virtual environment, activate it, and install the required libraries:

   ```
   uv venv --seed --python=3.13
   .\.venv\Scripts\activate
   pip install fastmcp==2.12.4 fast-agent-mcp==0.3.15
   ```

5.3. Core

5.3.1. Step 1: Understanding the Todo Server Structure

1. **Open the Starter File**
   Open `todo_server.py` in your code editor. Review the basic structure that's already set up for you.

2. **Understand the Todo Class**
   The starter code uses Python's `@dataclass` decorator to create a simple `Todo` class. It represents a single todo item with an ID, description, and completion status.

3. **Understand the TodoList Class**
   The `TodoList` class manages all todos:

   - `next_id`: Tracks the next available ID (auto-incrementing)
   - `todos`: Dictionary storing todos by ID (fast lookup)
   - `add()`: Creates and stores a new todo
   - `complete()`: Marks a todo as completed

     Creating a todo list may look like this:

     ```
     todo_list = TodoList()
     todo_list.add("Write documentation")
     todo_list.add("Update tests")
     todo_list.complete(1)
     print(todo_list.todos)
     todo_list.clear()
     ```

5.3.2. Step 2: Implement Add Todo Tool

1. **Implement the `add_todo` Function**
   A simple MCP tool function looks like this:

   ```
   @mcp.tool()
   def get_url(url: str) -> str:
       """Fetch the content of a URL."""
       return "<content>"
   ```

2. **Test Your Function**
   In the terminal, start your MCP server:

   ```
   fastmcp dev todo_server.py
   ```

   Use the interface to call `add_todo` with a sample todo item like "Write documentation". Verify it returns a Todo object with an ID and `completed` set to False.

5.3.3. Step 3: Implement the Remaining Tools

1. **Implement `list_todos`**
   Write a tool function that returns all todos. It should:
   - Take no parameters
   - Have return type `List[Todo]`
2. **Implement `complete_todo`**
   Write a tool function that marks a todo as completed. It should:
   - Accept a `todo_id` parameter (int)
   - Return the completed Todo object
3. **Implement `clear_todos`**
   Write a tool function that clears all todos. It should:
   - Take no parameters
   - Return nothing
4. **Test Your Functions**
   Use the `fastmcp dev` interface to:
   - Call `add_todo` to create a few todos
   - Call `list_todos` to see all your todos
   - Call `complete_todo` with one of the todo IDs
   - Call `list_todos` again to verify the todo was marked complete

5.3.4. Step 5: Testing with Fast-Agent

1. **Add Your OpenAI API Key**
   Set your OpenAI API key in the `fastagent.secrets.yaml` file located in the lab directory. Replace `your-api-key-here` with your actual OpenAI API key.
2. **Review the MCP Configuration**
   Open `fastagent.config.yaml` in the lab directory. This file configures Fast-Agent to connect to your todo server. Fast-Agent will launch the MCP server automatically using the settings in this file.

3. **Launch Fast-Agent**
   In a new terminal window, run the following command to start Fast-Agent connected to your todo server:

   ```
   .\.venv\Scripts\activate
   fast-agent go --servers=todo
   ```

   Fast-Agent will automatically launch your MCP server and connect to it.

4. **Test the AI Assistant**
   In the Fast-Agent interface, test your todo functionality.

5.4. Challenge 1: Populate Todos from YAML File (Optional)

Enable bulk import of todos from a YAML file. This is helpful when an AI assistant needs to load a pre-defined task list from a recipe or checklist.

Several coffee recipe "todo list" files are provided in the `LabFiles/recipes/` directory. Each file contains a simple structure with a `todos` key containing a list of steps.

5.5. Challenge 2: Add Nested Todos (Optional)

Extend your todo list to support hierarchical tasks—where a todo can have a parent task. This is useful for AI assistants breaking down large tasks into subtasks.

5.6. Conclusion

In this lab, you've built an MCP server with task management capabilities:

- A todo list server with stateful operations that persist across tool calls
- Tools for adding, listing, and completing tasks

- Integration with Fast-Agent AI assistant using MCP configuration

The todo list demonstrates how MCP servers can maintain state and provide AI assistants with memory for managing complex, multi-step workflows.

Using Sampling to Enable Agentic Behavior

MODULE 6

- ✓ Build MCP tools that use sampling for creative content generation
- ✓ Implement constrained sampling with business rules and requirements
- ✓ Demonstrate iterative refinement sampling techniques
- ✓ Create tools for comparative analysis of generated content
- ✓ Test sampling strategies with Fast-Agent integration

6.1. Introduction

BeanBotics marketing team needs to quickly generate diverse seasonal menu items for their upcoming fall campaign. Instead of brainstorming one idea at a time, they want to use AI sampling to explore the creative space and generate multiple variations efficiently.

In this lab, you'll build MCP tools that demonstrate different sampling strategies for creative content generation. You'll learn how to balance creativity with business constraints, refine concepts iteratively, and analyze generated content for market appeal.

Think of this as building a creative assistant that can rapidly explore the design space for seasonal beverages—generating dozens of variations while respecting dietary restrictions, brand guidelines, and seasonal themes.

6.2. Getting Started

1. **Open Your Terminal**
   Open a Terminal in the `LabFiles/seasonal-menu-sampling` lab directory.

2. **Install Required Libraries**
   Create a virtual environment, activate it, and install the required libraries:

```
uv venv --seed --python=3.13
.\.venv\Scripts\activate
pip install fastmcp==2.12.4 fast-agent-mcp==0.3.15
```

6.3. Core

6.3.1. Step 1: Seasonal Drink Generation

1. **Review the Server Skeleton**
   Open `seasonal_menu_server.py` and review the foundation code provided. You'll see the `FastMCP` server initialization, dataclass definitions for `DrinkConcept` and `DrinkRecipe`, and JSON schema methods that define the expected response format.

   The server is already configured with the name "BeanBoticsSeasonalMenu" and includes all the imports you'll need. Notice how the dataclasses use type hints and provide `json_schema()` class methods—these schemas will guide the AI in generating properly structured responses.

2. **Implement the `generate_themed_drinks` tool**
   Your first task is to implement the `generate_themed_drinks` tool using the `@mcp.tool()` decorator. This tool should:

   - Accept a `theme` parameter (like "autumn", "winter", "holiday")
   - Accept a `count` parameter for how many drinks to generate
   - Use `ctx.sample()` to generate creative drink concepts
   - Parse the JSON response and return a list of `DrinkConcept` objects

     Here's a complete example of how sampling works with an MCP tool:

     ```python
     @mcp.tool()
     async def translate_hello_world(
         language: str,
         ctx: Context = None
     ) -> dict:
         prompt = f"Translate 'Hello World' into {language}."
         response = await ctx.sample(messages=prompt)
         return response.text
     ```

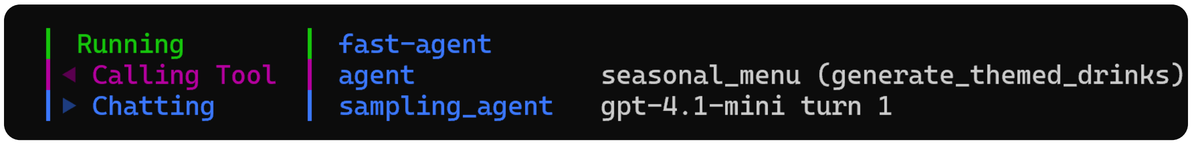
> ⓘ Note
>
> Use the `DrinkConcept.json_schema()` method in your prompt to specify the expected JSON format. This ensures the AI generates responses that match your data structure exactly.

3. **Test sampling with fast-agent**
   Now let's test your tool with an actual AI assistant. First, make sure your OpenAI API key is set in `fastagent.secrets.yaml` in the lab directory.

   Launch Fast-Agent to connect to your seasonal menu server and test the drink generation tool.

   ```
   fast-agent go --servers=seasonal_menu
   ```

   You should notice the sampling behavior as the AI generates multiple drink concepts based on your prompts.



```
┃  Running        ┃ fast-agent
┃ ◀ Calling Tool  ┃ agent            seasonal_menu (generate_themed_drinks)
┃ ▶ Chatting      ┃ sampling_agent   gpt-4.1-mini turn 1
```

6.3.2. Step 2: Recipe Generation Sampling

1. **Implement the `generate_drink_recipe` tool**

2. **Test sampling with fast-agent**

6.3.3. Step 3: Drink Scoring

1. **Implement the `score_drink` tool**
   Now you'll create a tool that uses sampling to evaluate drink concepts. This demonstrates how sampling can be used for analysis and decision-making, not just content generation.

Your `score_drink` tool should:

- Accept parameters for `drink_name` and `recipe` (the recipe description/ingredients)
- Use `ctx.sample()` to evaluate the overall appeal of the drink
- Return a simple integer score from 1-5 (1=poor, 5=excellent)

  The prompt should ask the AI to consider factors like creativity, market appeal, and complexity, then provide a single rating. Keep the implementation simple—just return the numeric score as an integer.

2. **Test the complete workflow with Fast-Agent**
   Restart the `fast-agent` server and test it with a few prompts, such as:

   - "Generate 5 summer drink ideas and rank them."
   - "Ideate on 3 winter drinks and pitch me the best one."

6.4. Challenge 1: Seasonal Bundle Creator (Optional)

BeanBotics marketing team wants to create promotional bundles that combine a seasonal drink with a complementary pastry and compelling promotional copy. Instead of manually brainstorming each bundle, they want a tool that generates complete, cohesive bundles where all elements work together thematically.

Create a `generate_seasonal_bundle` tool that uses sampling to create complete promotional packages. It should include drinks, recipes, menu descriptions, and social media announcement posts.

6.5. Challenge 2: Dietary Restriction Adapter (Optional)

BeanBotics serves customers with diverse dietary needs—vegan, gluten-free, keto, and diabetic-friendly options are frequently requested. Rather than creating entirely new recipes, the team wants a tool that adapts existing seasonal drinks to meet dietary restrictions while preserving the original flavor profile.

Create a `adapt_recipe_for_dietary_needs` tool that uses sampling to intelligently substitute ingredients. Your tool should. Consider these common adaptations:

- **Vegan**: Replace dairy milk with oat/almond milk, use coconut whipped cream
- **Keto**: Substitute sugar-free syrups, use heavy cream or unsweetened almond milk
- **Gluten-free**: Ensure toppings and add-ins contain no gluten
- **Diabetic-friendly**: Use sugar-free sweeteners like monk fruit or stevia

Test your tool by adapting one of the seasonal drinks you generated in the Core section to multiple dietary restrictions.

6.6. Conclusion

In this lab, you've built MCP tools that demonstrate key sampling strategies:

- Constrained creative sampling with business rules and requirements
- Iterative refinement sampling for concept development
- Recipe generation sampling for practical drink creation
- Integration with Fast-Agent for practical AI assistant workflows

You now understand how sampling enables rapid creative exploration while maintaining business constraints and quality standards.

Creating an AI Agent for Help Desk Support

MODULE 7

- ✓ Build a complete ticketing system with real-time WebSocket events for BeanBotics support
- ✓ Create an MCP server that provides ticketing tools and intelligent troubleshooting guide selection
- ✓ Implement a WebSocket agent monitor that responds to support events using FastAgent

- ✓ Understand WebSocket communication patterns and real-time event-driven architectures
- ✓ Demonstrate end-to-end integration of MCP tools, sampling, and WebSocket monitoring
- ✓ Create a practical AI-powered support system for robotic coffee machine troubleshooting

7.1. Introduction

BeanBotics has deployed robotic baristas across coffee shops worldwide, and when these sophisticated machines encounter issues, support tickets flow into their central ticketing system. Rather than waiting for human technicians to manually review each ticket, BeanBotics wants an AI-powered support system that can respond to issues in real-time.

In this lab, you'll build a complete support ecosystem that combines modern web technologies with AI intelligence. You'll create a ticketing API, implement real-time WebSocket event streaming, build MCP tools for ticket management, and deploy an AI agent that monitors the system and responds to support requests automatically.

The system demonstrates three key integration patterns: RESTful APIs for data management, WebSocket communication for real-time events, and MCP tools for AI-powered decision making. By the end, you'll have a production-ready support system where AI agents can create tickets, add comments, access troubleshooting guides, and resolve customer issues autonomously.

7.1.1. The Architecture

Your system will have four main components:

- **Ticketing API Server**: Flask-based REST API with SQLite database for ticket management
- **WebSocket Event System**: Real-time event broadcasting when tickets are created, updated, or commented on
- **MCP Ticketing Server**: FastMCP server providing tools and intelligent troubleshooting guide selection
- **AI Agent Monitor**: FastAgent-powered WebSocket client that listens for events and responds intelligently

7.1.2. Understanding WebSockets

WebSockets enable real-time, bidirectional communication between client and server. Unlike traditional HTTP requests that follow a request-response pattern, WebSockets maintain persistent connections that allow servers to push updates to clients instantly when events occur.

In our ticketing system, when a customer creates a support ticket through the web interface, the server immediately broadcasts a `ticket_created` event to all connected WebSocket clients—including our AI agent monitor. This enables the AI to respond to support requests within seconds rather than polling for updates.

7.2. Getting Started

1. **Open Your Terminal**
   Open a Terminal in the `LabFiles/beanbotics-troubleshooting` lab directory.

2. **Install Required Libraries**
   Create a virtual environment, activate it, and install the required libraries:

   ```
   uv venv --seed --python=3.13
   .\.venv\Scripts\activate
   pip install -r requirements.txt
   ```

3. **Explore the Ticketing System Structure**
   Examine the lab directory structure:

   ```
   ticketing-system/
   ├── app.py                     # Flask API server with WebSocket events
   ├── database.py               # SQLite database operations and seeding
   ├── ticket_mcp_server.py      # MCP server with tools and sampling
   ├── websocket_agent_monitor.py # FastAgent WebSocket event monitor
   ├── troubleshooting/          # Markdown troubleshooting guides
   │   ├── robotic_arm.md
   │   ├── grinder_motor.md
   │   ├── facial_recognition.md
   │   ├── boiler_temperature.md
   │   ├── milk_frother.md
   │   └── bean_hopper.md
   ├── fastagent.config.yaml     # FastAgent MCP server configuration
   ├── fastagent.secrets.yaml    # API keys and secrets
   └── templates/                # Web interface templates
       └── index.html
   ```

This structure separates concerns clearly: the Flask app handles HTTP and WebSocket communication, the MCP server provides AI tools, and the agent monitor bridges real-time events with AI responses.

7.3. Core

7.3.1. Step 1: Setting Up the Ticketing Server and Database

1. **Start the Flask API server**
   Launch the ticketing system server, which will automatically initialize the database.

   ```
   python app.py
   ```

   The server will start on `http://localhost:5000` and display startup messages indicating both HTTP and WebSocket services are running. On first launch, you'll see messages about database initialization and sample ticket creation.

2. **Test the web interface**
   Open your browser to `http://localhost:5000` to access the BeanBotics ticketing web interface. You should see:

   - A form for creating tickets — You can use this for testing.
   - A button labeled "🌱 Seed Random" that allows you to clear existing tickets and create a new sample ticket. — This is particularly useful for testing.

3. **Seeding a Ticket"**
   Click the "🌱 Seed Random" button. You should see a new ticket be created. This ticket is randomly selected from a predefined list of options. Each ticket includes realistic error messages and customer descriptions.

4. **Test the API**
   Open your browser to `http://localhost:5000/openapi/redoc`. You should see documentation of the API. You can explore the available endpoints and their

request/response formats. Since this is a demo system, there are no authentication requirements.

7.3.2. Step 2: Building the MCP Ticketing Tools

1. **Review the MCP server foundation**
   Open `ticket_mcp_server.py` and examine the starter code. You'll see the FastMCP server initialization, but not much else. You'll be implementing MCP primitives to interact with the ticketing API and troubleshooting guides.

2. **Review the Agent**
   Open `websocket_agent_monitor.py` and examine the FastAgent setup. A few key things are happening in this file:

   - SocketIO connects to `http://localhost:5000` to listen for ticketing events. These events are marked by `@sio.event` decorators.
   - `print_event` function logs incoming events for visibility.
   - `FastAgent` is initialized with a system prompt Notice the system prompt that guides the agent's behavior. The agent is configured to connect to the MCP server you'll be building.
   - **Run the Agent**
     Open `fastagent.secrets.yaml` and add your OpenAI API key. Then, open a new terminal window in the lab directory and run:
     ```
     python websocket_agent_monitor.py
     ```
     You should see the agent start up and connect to the MCP server once it's running. Since the MCP server isn't implemented yet, the agent won't be able to do much.

3. **Test WebSocket Events**
   With both `app.py` and `websocket_agent_monitor.py` running, create a new ticket using the web interface at `http://localhost:5000`. You should see the agent log incoming `ticket_created` events in its terminal window.

   The agent will attemp to respond, but won't affect anything yet since the MCP server tools aren't implemented.

7.3.3. Step 3: Implementing MCP Ticketing Tools

1. **Select the Tools**
   Review the API documentation at `http://localhost:5000/openapi/redoc` to understand the available endpoints. From these, make a list of the MCP tools that you will need to implement.

2. **Implement the Tools**
   In `ticket_mcp_server.py`, implement the tools that you identified. You may choose to implement one tool at a time, testing each with the MCP Inspector before moving on to the next. Or take the dangerous approach of implementing all tools before testing! Both the MCP inspector and the `fastagent go --servers=tickets` are great ways to test your tools.

   Here's the pattern for making API calls in Python:

```
try:
    response = requests.get(f"{API_BASE_URL}/endpoint")
    if response.status_code == 200:
        return response.json()
    else:
        return {"error": "<custom message based on response>"}
except Exception as e:
    return {"error": f"Request failed: {str(e)}"}
```

3. **Try out your Agent**
   To test the agent with your MCP server, you'll need to restart the agent. The agent will automatically start and connect to the MCP server when launched.

7.4. Challenge 1: Add Troubleshooting Guide Resources (Optional)

Expose your troubleshooting guides as MCP resources so AI agents can browse and access them. If you notice that your agent is struggling to select the right guide based on issue descriptions, you might try implementing the selection and retrieval of the guide as a tool with MCP sampling.

7.5. Challenge 2: Intelligent Response Templates (Optional)

Build an MCP tool that generates contextual response templates for support agents using sampling.

Add a `generate_response_template(ticket_id, response_type)` tool that uses MCP sampling to create professional support responses. The `response_type` should support four categories: **request_more_info** (ask specific diagnostic questions), **provide_solution** (reference troubleshooting guide steps), **escalate** (explain escalation reasoning), and **close** (summarize resolution). This enables your agent to post contextually appropriate responses instead of generic messages.

7.6. Conclusion

You've built a comprehensive AI-powered support system that integrates real-time events with MCP tools. You learned to:

- Create MCP ticketing tools that interact with REST APIs
- Use MCP sampling for intelligent troubleshooting guide selection
- Build WebSocket event monitors with FastAgent
- Integrate event-driven architectures with AI assistant workflows
- Handle error conditions and prevent infinite comment loops

Your BeanBotics ticketing system demonstrates how AI agents can monitor support systems continuously and respond to customer issues autonomously using standardized MCP protocols.