Project_Report_B06

📂 Project: MCP CSV Explorer (Analytics Edition)

Esame: Advanced Programming

Studente: Michele Sagone

Matricola: 720510

Email: m.sagone1@studenti.unipi.it

1. Introduction

This document discusses the building and validation process for the code generated by the AI assistant for the "MCP CSV Explorer" project. It details how the output was tested, verified, and refined following a strict "Human-in-the-Loop" methodology.

The objective was to create a Model Context Protocol (MCP) server capable of exposing local datasets (CSV) to Large Language Models, not just as raw text, but as a structured, queryable knowledge base. The project emphasizes the shift from the developer as a "Code Writer" to a "Critical Verifier," focusing on Type Safety, Asynchronous I/O, and Deterministic Analytics.

2. Gaining Knowledge & Architectural Decisions

Before starting to build the project, I conducted research on the technologies required to bridge the gap between local filesystems and LLMs. I asked the AI specific, architectural questions to determine the best stack.

I decided to use **Gemini 2.0 Flash / 3.0** for its superior reasoning capabilities in Python and its large context window, which was crucial when analyzing the documentation for the `mcp` SDK.

2.1 Protocol Selection: STDIO vs. SSE

The first critical architectural decision was choosing the transport layer for the MCP server. I queried the AI to compare the two main options provided by the SDK: `stdio` (Standard Input/Output) and `sse` (Server-Sent Events).

- **Initial Hypothesis:** I initially considered `stdio` for its simplicity, as it pipes data directly between processes.
- **The Problem:** During the research phase, I learned that `stdio` makes debugging difficult because the communication channel is occupied by the protocol itself, preventing the use of standard print-debugging or external inspection tools.
- **The Decision:** I chose **SSE (Server-Sent Events)** over HTTP. This decision aligns with the course topics on **Asynchronous Programming** and **Non-Blocking I/O**.
- **Reasoning:** SSE allows the server to run as a standalone web service (using `uvicorn` as the ASGI server), decoupling it from the client. This enables the use of the **MCP Inspector** web tool for validation, which was critical for the "Critical Verifier" workflow.

2.2 Data Marshalling Strategy: Raw I/O vs. Pandas

I needed to determine how to read and process the CSV files. I asked the AI to evaluate different approaches for "Marshalling" data from disk to the LLM.

- **Option A: Built-in `csv` module.** The AI suggested this for lightweight dependencies. However, I rejected it because it treats data as strings, lacking type awareness.
- **Option B: `pandas` DataFrames.** I selected this option despite the heavier dependency.
- **Theoretical Justification:** As studied in the course (Runtime Environments), Python objects carry significant overhead. However, Pandas uses C-optimized structures (NumPy arrays) for memory management.
- **Validation Benefit:** Using Pandas allows for **Deterministic Analytics** (e.g., `df.describe()`). Instead of asking the LLM to calculate the average of 1000 rows (which leads to hallucinations), the server performs the math deterministically

and sends only the result. This drastically reduces the "Token Context Window" usage and eliminates calculation errors.

2.3 Security & Isolation

To ensure the project would run on the examiner's machine without "it works on my machine" issues (a core concept of **DevOps** and **Reproducibility**), I researched Python environment isolation.

- **Containerization vs. Virtualization:** While Docker was an option, I opted for a lightweight Python Virtual Environment (`venv`).
- **Dependency Management:** I mandated the creation of a strict `requirements.txt` file. I verified that all libraries (`mcp`, `pandas`, `uvicorn`, `tabulate`) were pinned to compatible versions to prevent "Dependency Hell" during the exam evaluation.

3. Development Platform and Tools

To develop the project, I used **VS Code** running on **WSL (Windows Subsystem for Linux)**. This setup was chosen to mimic a production-grade Linux environment while maintaining the usability of Windows.

3.1 Context Management Strategy

I adopted a strategy of **Context Compartmentalization**. Based on my experience, maintaining a single long chat session with the AI leads to "Context Drift," where the model becomes confused by previous, now obsolete, instructions.

- **Strategy:** I treated each major feature (e.g., "Implement SSE", "Add Analytics Tool", "Refactor for Type Hints") as a separate "Thread".
- **Hallucination Recovery:** When the model started hallucinating non-existent `fastmcp` methods, I did not argue with it in the same chat. I closed the session and started a new one with a "Clean Slate" prompt, providing the correct documentation snippet as context. This proved far more effective than trying to "debug" the AI's confusion.

3.2 "Chain of Thought" Prompting

Instead of a single "One-Shot" generation for the whole server, I used a **Chain of Thought** approach, breaking the development into 4 distinct phases:

1. **Skeleton:** Establishing the server connection and basic file listing.
2. **Schema Introspection:** Implementing `get_schema` (Reflection).
3. **Analytics Layer:** Adding `get_stats` and `search_in_table`.
4. **Hardening:** Adding Type Hints, Pathlib Security, and Async Optimizations.

This granular approach allowed me to validate each component independently before moving to the next, reducing the complexity of the debugging phase.

4. Installation and Usage (Reproducibility)

To ensure consistent execution across different environments, I avoided complex containerization in favor of a standard, lightweight Python Virtual Environment (`venv`). This decision enforces **Dependency Isolation**, preventing conflicts with system-wide packages.

4.1 Prerequisites

- **Python 3.10+**: Required for modern type hinting support.
- **Git**: For version control.

4.2 Installation Protocol

The application strictly defines its dependencies in `requirements.txt`.

1. **Clone & Setup:**

```
git clone <https://github.com/Micheles111/mcp-csv-server>
cd mcp-csv-server
python -m venv venv
source venv/bin/activate  # On Windows: venv\\Scripts\\activa
te
```

1. **Dependency Locking:**

```
pip install -r requirements.txt
```

Verification: I manually verified that `uvicorn` (the ASGI server) and `pandas` were correctly installed, as the initial AI-generated `requirements.txt` missed the `[cli]` extra for the `mcp` package.

4.3 Running the Server (SSE Transport)

Unlike standard scripts, this server runs as a long-lived process listening for **Server-Sent Events**.

```
python server.py
```

- **Observation:** Upon startup, the console confirms: `Starting MCP Server with SSE Transport on port 8000...`. This confirms that the asynchronous event loop is active and ready to accept connections.

5. Client-Side Usage (The Inspector)

Since the server uses the MCP protocol, it does not have a traditional frontend. Instead, I used the **MCP Inspector** as a universal client to validate the server's API. This demonstrates the **Decoupling** between the Backend (Python) and Frontend (Web/LLM).

5.1 Connection Procedure

1. **Launch Inspector:** `npx @modelcontextprotocol/inspector`
2. **Transport Configuration:**

- **Type:** SSE (Server-Sent Events)
- **URL:** `http://127.0.0.1:8000/sse`

1. **Connection:** Clicking "Connect" establishes a persistent bidirectional channel.

5.2 Functional Testing

Through the Inspector, I validated the available Tools:

- `list_tables`: Verified it returns a clean list (`['products', 'users', 'orders']`) without file extensions.
- `get_schema`: Verified it correctly identifies column types (e.g., `price` as `int64` or `float`).
- `get_stats`: Verified that the Markdown output is rendered correctly, providing a deterministic statistical summary (Mean, Min, Max) without hallucinated values.

6. Testing and Validation Methodology

The core of this project was not writing code, but **validating** AI-generated logic. I adopted a strict **"Generate-Verify-Accept"** workflow. Unlike traditional debugging, where the developer fixes syntax errors, my role was to audit the logic and **revert** architectural mistakes.

6.1 The "Generate-Verify-Accept" Loop

1. **Prompting:** Issuing a specific, technical request (e.g., "Implement a tool to calculate stats using Pandas").
2. **Immediate Verification:** Running the code in WSL.
3. **Decision Gate:**

- **Accept:** If the tool worked and handled edge cases.
- **Revert & Retry:** If the AI introduced "Code Smells" (e.g., hardcoded paths or blocking calls), I reverted the changes and refined the prompt.

6.2 Failure Modes and Interventions

The "Revert and Retry" strategy was triggered by specific failure modes:

- **Hallucinated SQL Capabilities:** Initially, the AI attempted to treat CSV files as a SQL database using `sqlite3` on raw text files.
- *Intervention:* I rejected this approach because it was fragile. I forced a **Revert** and explicitly prompted for a **Pandas-based approach**, which provides a robust, in-memory dataframe engine for reliable analytics.

- **Path Traversal Vulnerabilities:** Early versions of the `get_csv_path` function blindly concatenated user input strings.
- *Intervention:* I recognized the security risk (Path Traversal). Instead of patching it with fragile string checks, I prompted the AI to refactor the code using `pathlib`. This allowed for an Object-Oriented approach where `path.resolve()` strictly confines file access within the `data/` directory.
- **Type Hinting Inconsistencies:** The AI often omitted return types.
- *Intervention:* As a "Critical Verifier," I enforced a refactoring pass solely dedicated to adding strict **Type Hints** (`> List[str]`, `> Dict`), aligning the code with professional Python standards.

6.3 Cross-Environment Validation (WSL vs. Windows)

A critical validation step was ensuring the server ran on both Linux (WSL) and native Windows.

- **Issue:** Different OSs handle file paths differently (`/` vs `\\`).
- **Resolution:** The adoption of `pathlib` (see section 6.2) automatically resolved these discrepancies, as it abstracts filesystem differences. The final code was tested on **Windows 11 (via PowerShell)** and **Ubuntu 22.04 (via WSL)**, confirming that the "Marshalling" of file paths works consistently in both environments.

7. Validation Results & Case Studies

7.1 Success Rate and Efficiency

The AI demonstrated high proficiency in generating "Boilerplate Code" (e.g., setting up the FastMCP class and basic decorators). In these instances, the **"Generate-Verify-Accept"** cycle was extremely fast. However, for logic involving complex system interactions (Async I/O), the success rate dropped, requiring active intervention.

7.2 Case Study: Concurrency and the GIL

A specific validation challenge arose during the implementation of the search tool and analytics.

- **Initial AI Approach:** The AI proposed standard `async def` functions that called `pd.read_csv()` directly.
- **Critique:** As a "Critical Verifier," I identified a major architectural flaw. Pandas operations are CPU-bound and **Synchronous**. Calling them directly inside an `async` function blocks the Python **Event Loop**, causing the SSE heartbeat to freeze and making the server unresponsive to other requests.
- **Refinement:** I mandated an architectural change: wrapping all DataFrame operations in `asyncio.to_thread`.
- **Outcome:** This offloads the heavy computation to a separate thread pool, keeping the main Event Loop free. This demonstrates a deep understanding of Python's **GIL (Global Interpreter Lock)** and asynchronous programming patterns.

7.3 Security Analysis: Path Traversal

While the application is designed for local usage, I performed a security audit on the file access logic, treating it as if it were a production web service.

- **Vulnerability:** A potential **Path Traversal** attack was identified. If a user requested `../../etc/passwd` as a `table_name`, the naive implementation might have accessed sensitive system files.
- **Resolution:** I enforced the use of `pathlib`'s `resolve()` method. The code now explicitly checks: `if not target_path.startswith(DATA_DIR): raise SecurityError`.
- **Limitation:** Unlike the reference project which implemented HTTPS, this server operates over plain HTTP (localhost). I acknowledge that in a production environment, this would require a Reverse Proxy (Nginx) or a VPN tunnel.

8. Conclusion

The development of the **MCP CSV Explorer** confirms the shift in the modern developer's role from "Code Writer" to **"System Architect and Quality Assurance Lead."**

The project successfully achieved its goal: transforming a static filesystem into a dynamic, queryable knowledge base for LLMs. The use of **SSE Transport** proved superior to STDIO for debugging and decoupled testing.

However, the reliability of the output was not absolute. The project's success relied heavily on the **"Human-in-the-Loop"** methodology. As predicted by the "Turing Prophecy" discussed in the course, my primary responsibility was not to type syntax, but to specify requirements (Prompt Engineering) and verify behavior (Auditing).

The ability to **Revert and Retry** proved to be a more powerful tool than traditional debugging, allowing for rapid prototyping while maintaining the integrity of the application's requirements.