$\text{Assignment}_5 Summary$

John Lahoud

November 2025

Summary of MCP Concepts and Map Server Design

1. What MCP Is The Model Context Protocol (MCP) is an open standard that enables AI assistants to interact with external tools, data sources, and computational workflows through a unified integration layer. Instead of relying on ad-hoc API calls or custom function schemas, MCP introduces the idea of standardized servers that expose consistent interfaces to any compatible AI model. This approach is often compared to a "USB-C for AI," where the assistant speaks a single protocol and can seamlessly connect to diverse capabilities such as file access, databases, code execution, or cloud services. MCP servers expose tools, resources, and prompt templates, and MCP SDKs abstract most of the implementation details, making it easier for developers to build reusable and interoperable services.

2. Key Concepts From the Kseniase Article The Kseniase article highlights why MCP has quickly gained traction: it simplifies how AI systems integrate with external resources, reduces fragmentation, and replaces custom OpenAPI specifications or hand-engineered function-call interfaces. MCP servers are designed as independent, reusable components that can be plugged into any MCP-compatible host, whether it is ChatGPT, Claude, IDE extensions, or automated agents. The article showcases common examples such as filesystem servers, GitHub integration servers, JSON utilities, calculators, and other modular tools. All of these follow the same interaction model, demonstrating how MCP unifies previously disparate integration patterns into a standard, predictable framework.

3. Design Patterns in Map Servers When examining existing MCP-based or MCP-inspired map servers such as those built around Google Maps, Baidu Maps, MeasureSpace, or other geospatial APIs, several design patterns consistently appear. First, these servers provide a set of core operations including geocoding, reverse geocoding, route planning, distance and duration estimation, point-of-interest search, and occasionally overlays for weather or traffic. The dominant design pattern is the use of tool-style APIs: clearly named functions

with typed parameters (e.g., `geocode(address)`, `route(origin, destination)`) and stable JSON schemas that LLMs can reliably parse. Another common pattern is parameterization through `ServerParams`, which define configurable elements such as base URLs, API keys, rate limits, or default units. Finally, robust map servers emphasize safety and reliability through controlled timeouts, structured error handling, and separation between the raw HTTP API and the MCP tool interface, ensuring predictable behavior even when underlying services fluctuate.

4. How These Ideas Influenced My Design The map servers implemented for this assignment follow the same principles observed in real-world MCP-based systems. The project includes two distinct servers: *OSMGeoServer*, which handles geocoding and point-of-interest search using OpenStreetMap data, and *SimpleRoutingServer*, which provides routing, distance estimation, and travel-time calculations. Each server is implemented as a collection of tools defined using the OpenAI Agents SDK, enabling the assistant to call these functionalities programmatically. Both servers rely on structured, MCP-style `ServerParams` objects that specify base URLs, user-agent strings, and timeouts. This design ensures clarity, modularity, and maintainability, while adhering to the integration patterns recommended by MCP development practices.

Reflection

Throughout this project, I gained valuable practical insight into how the Model Context Protocol (MCP) shapes interactions between AI assistants and external tools. Working with the OpenAI Agents SDK transformed the abstract concepts of MCP into concrete workflows, allowing me to design modular servers with clearly defined tool interfaces. Developing the geocoding/POI server and the routing server helped me understand essential ideas such as well-structured tool functions, the use of `ServerParams` for configuration, and the importance of stable JSON outputs for reliable LLM reasoning. The implementation process also reinforced the need for safety mechanisms—including timeouts, controlled errors, and response validation—to ensure consistent behavior when dealing with real-world APIs.

Looking ahead, the project revealed several promising directions for further development. One potential next step is converting these tool-based implementations into fully independent MCP servers using official MCP SDKs, enabling broader interoperability across multiple AI hosts. Additional enhancements could include integrating traffic-aware routing, elevation and terrain data, batch geocoding, or caching layers to improve performance. It would also be beneficial to incorporate geospatial visualization tools or build a lightweight interface to display the agent's outputs visually. Overall, this assignment strengthened my understanding of protocol-driven AI integrations and provided a strong foundation for expanding these capabilities in more advanced geospatial or multi-agent systems.