## **Files and Their Roles**

* main.py: Orchestrates everything, loads data, sets up agents, runs the simulation.
* engineering\_data.yml: Initial requirements and artifacts (manually filled).
* prompts.yml: Agent definitions, prompts, and tool mappings.
* tools\_event\_sourcing.py: Implements all agent actions as event-sourced functions.
* neo4j\_event\_sourcing.py: Handles event logging, session management, and replay.
* Neo4jAgent.py: Defines agent behavior, context updates, and message handling.

## **Client and Server Side in This Codebase**

### **Server Side**

* Neo4j Database acts as the server-side data store.
* All requirements, artifacts, and their relationships (traceability) are stored as nodes and relationships in Neo4j.
* The server is responsible for:
* Persisting all changes as events (event sourcing)
* Serving queries about the current state and history
* Supporting traceability and impact analysis

### **Client Side**

* The Python application (our code) is the client.
* It:
* Loads initial data from YAML files (engineering\_data.yml, prompts.yml)
* Instantiates agents and tools
* Orchestrates the simulation and agent interactions
* Sends/receives data to/from Neo4j via the Neo4j Python drive

## **Data Flow: From Where Is Data Taken?**

### **Startup**

* main.py loads configuration from config.yaml.
* It then loads:
* Requirements and artifacts from engineering\_data.yml
* Agent definitions and prompts from prompts.yml

### **Initialization**

* The loaded data is imported into Neo4j as nodes and relationships.
* A new session is created in Neo4j to track all subsequent changes.

### **During Execution**

* Agents interact with the data by calling tools (functions in tools\_event\_sourcing.py).
* Each tool call (e.g., add requirement, add artifact, add link) creates an event in Neo4j.
* The current state is always reconstructed by replaying these events (event sourcing).

## **3. Agent Orchestration: Who Works First and How?**

### **Agent Setup**

* Agents are defined in prompts.yml (e.g., SystemArchitect, RequirementsManager, MBSE\_EngineeringAssistant, MBSE\_Orchestrator).
* Each agent has:
* A role-specific prompt
* A set of tools it can use
* A context (requirements, architecture, traceability)

### **Agent Execution Flow**

1. Orchestrator/Team Setup

* In main.py, a SelectorGroupChat is created with all agents as participants.
* The orchestrator (MBSE\_Orchestrator) coordinates the process.

1. First Agent Actions

* The simulation starts with a task (e.g., "Check if the architecture is complete and requirements are consistent").
* Agents receive the initial context and system message.

1. Agent Reasoning and Tool Use

* Each agent reasons about its context (using LLMs, prompts, and available tools).
* If an agent decides a change is needed (e.g., a requirement is unclear, an artifact is missing), it calls the appropriate tool.
* Tool calls are logged as events in Neo4j.

1. Traceability Agent (MBSE\_EngineeringAssistant)

* Specifically monitors and manages the links between requirements and artifacts.
* Ensures every requirement is linked to at least one artifact and vice versa.
* Uses tools like tool\_add\_link and tool\_remove\_link.

1. Orchestrator Feedback

* The orchestrator evaluates the state after each round (using scoring, feedback, and status).
* If the model is complete and consistent, the orchestrator ends the process; otherwise, it provides feedback and the agents continue.

### **Agent Communication**

* Agents communicate by:
* Reading the shared context (from Neo4j and system messages)
* Broadcasting updates (e.g., via UpdateSystemMessageEvent)
* Responding to orchestrator feedback
* A2A communication is possible by having agents connect to each other (or to a central orchestrator) via MCP endpoints.
* Each agent can be exposed as an MCP server, and other agents (or orchestrators) can call their tools/prompts via the MCP protocol.

## **2. Which files will need to be changed?**

Assuming we want to MCP-enable our agents and use LangChain’s MCP adapter, we’ll need to:

### **a) Add/Modify MCP Server Setup**

* Create a new file (e.g., mcp\_server.py) or modify our main entrypoint to:
* Instantiate a FastMCP server
* Register our agent(s) as MCP tools or prompts

### **b) Agent Wrapping**

* Wrap our agent logic (e.g., the classes in Neo4jAgent.py) as MCP tools or prompts.
* If using LangChain, use the MCP adapter to expose our LangChain agent as an MCP tool.

### **c) Remove/Refactor Direct Neo4j/Tool Calls**

* Instead of calling tools directly, route all tool/agent calls through the MCP protocol (using the adapter).

### **d) Orchestrator/Client**

* If we have an orchestrator or a client agent, make it an MCP client (using the LangChain MCP adapter) that calls other agents via MCP.

## **3. How to scale and add more agents?**

### **a) Horizontal Scaling**

* Run multiple instances of our MCP server (each hosting one or more agents).
* Use a service registry or load balancer (e.g., NGINX, Envoy, or a cloud load balancer) to distribute requests among agent instances.

### **b) Adding More Agents**

* Register new agent classes as additional MCP tools/prompts in our FastMCP server.
* Or, run each agent as a separate FastMCP server and register them with a central orchestrator or service registry.

### **c) Load Balancing**

* Use a load balancer in front of our MCP servers.
* For A2A, agents can discover each other via a registry or a service mesh (e.g., Consul, Kubernetes, etc.).

## **4. What will the architecture look like?**

### **High-Level Architecture Diagram**

text

Apply to Neo4jAgent.p...

+-------------------+ +-------------------+ +-------------------+

| Agent MCP Srv | <-----> | Agent MCP Srv | <-----> | Agent MCP Srv |

| (FastMCP+Agent A) | | (FastMCP+Agent B) | | (FastMCP+Agent C) |

+-------------------+ +-------------------+ +-------------------+

^ ^ ^

| | |

+-----------+ +-----------+-------------+ +-----------+

| | Load Balancer/Service | |

+---+ Registry/Orchestrator +---+

+-------------------------+

* Each agent is exposed as an MCP server (using FastMCP).
* Agents communicate with each other via MCP (A2A).
* A load balancer or orchestrator can route requests and manage agent discovery.
* we can scale horizontally by running more agent instances.

## **5. LangChain Integration**

* Use the LangChain MCP adapter to wrap our agents as MCP tools/prompts.
* Use LangChain’s agent orchestration to manage workflows, chains, and memory.
* All agent-to-agent and client-to-agent calls go through MCP.

## **6. Summary of Steps**

1. Wrap each agent as an MCP tool or prompt (using LangChain MCP adapter).
2. Deploy multiple agent servers for scaling.
3. Use a load balancer or orchestrator for routing and discovery.
4. (Optional) Use a service mesh or registry for advanced scaling and monitoring.

## **7. Files to Change or Add**

* mcp\_server.py (new): MCPserber server setup, agent registration
* Neo4jAgent.py (modify): Wrap agent logic as MCP tools/prompts
* main.py (modify): Remove direct tool calls, use MCP client calls
* (Optional) Add orchestrator/registry logic for agent discovery and load balancing
* Keep Neo4j for state, traceability, and event sourcing.
* Use MCP for A2A communication (message passing, tool calls).
* Refactor: Agents should not call each other via Neo4j, but via MCP. They should use Neo4j only for data storage and retrieval.

## **Scaling and Load Balancing**

* Run multiple FastMCP (or MCPServer) instances for each agent.
* Use a load balancer (e.g., NGINX, Envoy) in front of our MCP servers.
* Agents discover each other via service registry or orchestrator.

## **Scalability: Adding More Agents**

* To add more agents:
* Define new agent classes or prompts in our code and prompts.yml.
* Register them in our team/orchestrator setup.
* No architectural change is needed—just instantiate more agents in our team.
* To scale horizontally (multiple servers):
* Run multiple instances of our MCP server (each with its own team/agents).
* Use a load balancer in front of our MCP servers.
* Each server can handle independent sessions/tasks.

## 

* we can use open-source tools such as NGINX or HAProxy to distribute requests across multiple FastMCP server instances.
* This can be run on our own servers, VMs, or even locally for development and small-scale production.
* Gives we full control over routing logic, session persistence, and custom algorithms.
* FastMCP supports running multiple instances behind a load balancer, and documentation provides guidance for configuring this setup[2](https://github.com/yjacquin/fast-mcp/blob/main/docs/integration_guide.md)[3](https://glama.ai/mcp/servers/@timothywangdev/McpToolKit).

## 3. In-Code Load Balancing

* For simple or experimental setups, we can implement basic load balancing logic directly in our application code (e.g., using round-robin or random selection to pick a FastMCP server instance).
* This approach is less robust and not recommended for production, but it is possible if we want to avoid external dependencies.

## Key Considerations

* Session State: For horizontal scaling, ensure session and token state are shared (e.g., using Redis), so requests can be handled by any server instance without session loss[3](https://glama.ai/mcp/servers/@timothywangdev/McpToolKit).
* Scalability and High Availability: Both cloud and self-managed solutions can achieve these goals if we architect our system with stateless servers and shared state[3](https://glama.ai/mcp/servers/@timothywangdev/McpToolKit)[5](https://www.byteplus.com/en/topic/541432).
* Security: Self-managed solutions require we to handle SSL, firewall rules, and DDoS protection, which cloud providers often include.