AI Protocol Engineer Challenge: Week 1

MCP Integration, Proxy & RAG Foundations

BUSINESS OBJECTIVE / BACKGROUND CONTEXT

Scenario: NexusAI's business requires intelligent agents to seamlessly access, aggregate, and reason over information scattered across numerous development and productivity tools – project management (JIRA via Atlassian MCP), version control (GitHub MCP), documentation (Google Drive MCP), local codebases (Filesystem MCP), and potentially more. Direct agent interaction with each tool's specific MCP server can become complex and inefficient to manage.

As a new AI Protocol Engineer, your first week focuses on building foundational capabilities for unified tool access. You will explore and interact with various existing MCP servers provided for common tools. You will use MCP client libraries (like Python fastmcp or Typescript fastmcp) to communicate with these servers. A key task will be to design and implement a basic **MCP Proxy/Gateway Server**. This proxy will act as a single, unified endpoint, intelligently routing requests to the appropriate downstream MCP server (e.g., GitHub, GDrive, Filesystem, Atlassian). You will integrate this proxy into a simple "Dev Assistant" agent (using Python fast-agent or NodeJS LangGraph.js) that performs Retrieval-Augmented Generation (RAG) over a local knowledge base, potentially exploring real-time indexing concepts. You will test the MCP proxy using integrated development environment (IDE) clients like **VS Code Copilot Chat** or **Cursor**, and research advanced protocol concepts.

Goal: By the end of this week, you will demonstrate a strong grasp of MCP fundamentals, the ability to interact with various existing MCP servers, experience designing and implementing a basic MCP proxy server, proficiency in building a simple agent (Python or NodeJS) integrating MCP tool use (via the proxy) and RAG (including awareness of real-time indexing), and the ability to test the MCP proxy using standard IDE integrations. You will also gain initial insights into enterprise protocol challenges like unified access and security.

DATA FOR EVALUATION (CHALLENGE INPUTS)

- **MCP Specification:** Link to Official MCP Documentation (Focus on Core Concepts, Server/Client Responsibilities, Request/Response structure). See **Appendix A**.
- **A2A Specification:** Link to Official A2A GitHub Repository/Docs (Focus on Core Concepts, relation to MCP). See **Appendix A**.
- **MCP Server Implementations & Frameworks:**
  - **Python:** jlowin/fastmcp - Framework for building MCP servers and clients.
  - **Typescript:** punkpeye/fastmcp - Framework for building MCP servers and clients.
  - **GitHub:** github/github-mcp-server - MCP Server for GitHub access.
  - **Filesystem:** modelcontextprotocol/servers/tree/main/src/filesystem - MCP Server for local filesystem access.
  - **Google Drive:** modelcontextprotocol/servers/tree/main/src/gdrive - MCP Server for Google Drive access.
  - **Atlassian (JIRA/Confluence):** sooperset/mcp-atlassian - MCP Server for Atlassian tools.
  - **MCP Gateway (Reference):** lasso-security/mcp-gateway - Example implementation for testing MCP Gateway concepts locally.
- **MCP Proxy/Gateway Inspirations:**
  - OpenMCP Standard - A token-efficient standard concept for converting web APIs into MCP servers (relevant to proxy design).
  - Metatool App - Example of unified middleware MCP.
- **Awesome MCP Servers List:** punkpeye/awesome-mcp-servers - Curated list for reference.
- **Mock Knowledge Base:** Local directory structure with sample markdown docs, code snippets (.py/.js), etc. as defined in **Appendix D** (for RAG component).
- **Agent Framework Documentation:**
  - Python: fast-agent Documentation.
  - NodeJS (Option): LangGraph.js Documentation. LangChain JS Agents Overview.
- **RAG Framework Documentation:**
  - Python: LlamaIndex Python Documentation (Focus on basic indexing and querying local files).
  - NodeJS: LangChain JS RAG Documentation (Focus on document loaders, basic vector stores, retrieval).
- **Real-time RAG Concepts (Pathway):**
  - Pathway LLM App Examples - Repository demonstrating real-time data processing for LLM applications.

  - Pathway Demo Document Indexing Pipeline - Specific example relevant to RAG.
- **MCP Client Libraries:**
  - Python: Use client capabilities within jlowin/fastmcp or standard libraries like aiohttp/httpx.
  - Typescript: Use client capabilities within punkpeye/fastmcp or standard libraries like node-fetch/axios.
- **IDE MCP Client Documentation:**
  - VS Code Copilot Chat: Using MCP Servers
  - Cursor: Model Context Protocol (MCP)
- **Python Documentation:** Link to Official Python Docs.
- **Node.js Documentation:** Link to Official Node.js Docs (If choosing NodeJS agent path).
- **Development Environment Setup Guide:** Provided in **Appendix C: IT Setup Guide**.
- **Research Articles (Advanced Concepts):**
  - A2A, MCP, Kafka, and Flink: The New Stack for AI Agents? (Inspiration)
  - API Gateway Pattern - NGINX (Example)
  - Microservices Security Patterns (Example)
- **AI Assistant Access:** Access to an LLM like Gemini or ChatGPT.

LEARNING OUTCOMES (SKILLS, KNOWLEDGE, AND BEHAVIORS TO BE DEMONSTRATED)

Knowledge:

- Explain MCP purpose, architecture, core methods, request/response structure. [K4]
- Explain A2A purpose and its complementary role to MCP. [K4]
- Understand asynchronous programming concepts (Python asyncio or Node.js event loop). [K1]
- Describe the basic architecture of the chosen agent framework (fast-agent or LangGraph.js). [K11]
- Understand the purpose and common patterns for MCP Proxy/Gateway servers. [K4, K5]
- Explain how to use MCP client libraries (fastmcp or others) to interact with different MCP servers. [K4, K11]
- Explain the core concepts of Retrieval-Augmented Generation (RAG). [K2]
- Understand basic principles of indexing (e.g., VectorStoreIndex in LlamaIndex) and querying local documents for RAG. [K2, K11]
- **Describe the concept of real-time data indexing for RAG using frameworks like Pathway.** [K2, K10]

- **Understand how to configure IDEs like VS Code Copilot Chat or Cursor to use MCP servers.** [K11]
- Identify potential benefits/challenges of MCP Gateways, RBAC, and Streaming based on research. [K4, K10, K15]
- Understand local environment setup using Python virtual environments or npm.

Skills:

- Set up a local Python virtual environment or NodeJS project environment. [C3 related]
- Write and debug asynchronous Python/Node.js code for network communication. [K1 related]
- Configure and run existing MCP servers (e.g., GitHub, Filesystem, GDrive, Atlassian) locally. [C1 related]
- Develop an MCP Client capable of interacting with multiple MCP server endpoints. [S1.1.2]
- Implement a basic asynchronous MCP Proxy/Gateway server (using fastmcp or equivalent) that routes requests to downstream MCP servers. [S1.1.1, FC5.1]
- Build a simple agent (Python/NodeJS) that uses an MCP Client (potentially via the proxy) for tool interaction. [S2.1.1, S2.3.1]
- Implement a basic RAG pipeline within the agent to query a local knowledge base (using LlamaIndex/LangChain JS). [S2.1.2 related]
- **Configure and test MCP server integration with VS Code Copilot Chat or Cursor.** [FC1.4]
- Write unit/integration tests for MCP client interactions and proxy routing logic using pytest/Jest. [FC1.4]
- Read and synthesize information from technical specifications, repositories, and articles (including Pathway examples). [C5, C6]
- Document technical understanding, conceptual designs (proxy, RAG), and IDE integration steps clearly. [S6.1.1]

Behaviors:

- Demonstrate **Curiosity** (B5) and **Adaptability** (B5, FC7.1) exploring various MCP servers, proxy patterns, RAG (including real-time concepts), IDE integrations, and chosen frameworks.
- Exhibit **Detail-Orientation & Rigor** (B7) in implementing protocol interactions, proxy logic, RAG pipeline, writing clean, tested code, and configuring IDEs.

- Practice **Innovative Problem-Solving** (B1) when designing the MCP proxy, integrating multiple services, and troubleshooting IDE connections.
- Show **Accountability** (B10) by completing tasks and committing code regularly.
- Engage in **Effective Communication** (B9) during simulated stand-ups and in documentation.

COMPETENCIES (TARGETED FOR MEASUREMENT)

This challenge focuses intensely on building foundational competencies:

- **C1: AI Protocol Engineering & Integration:** MCP Client Usage (FC1.1), MCP Server/Proxy Implementation (FC1.1), Protocol Testing (FC1.4 - including IDE testing).
- **C2: Intelligent & Autonomous Agent Development:** Basic Agent Building (FC2.1), Agent Tool Integration (FC2.3), Basic RAG Implementation (FC2.1 related).
- **C4: AI Data Pipeline & Storage Management:** (Introduced via RAG indexing and real-time concepts).
- **C5: Innovative Problem Solving & Protocol Advancement:** Applying Protocol/Proxy/RAG Concepts (FC5.1), Researching Advanced Topics (FC5.3).
- **C7: Professionalism, Ethics & Security:** Adaptability (FC7.1), Rigorous Implementation.
- **C6: Cross-Functional Collaboration & Communication:** (Introduced via stand-ups, documentation).

TOOLS

- Git & GitHub Account
- Python 3.10+ & venv / pip
- (Optional) Node.js (LTS version) & npm / yarn
- **Code Editor:**
- **VS Code** (with Copilot Chat extension enabled)
- **OR Cursor IDE**
- Terminal/Command Line Interface
- Web Browser
- Docker (Recommended for running some MCP servers)
- **Python Path:** jlowin/fastmcp, aiohttp or httpx, pytest, pytest-asyncio, fast-agent, llama-index, pathway (for exploring real-time RAG example).
- **NodeJS Path:** punkpeye/fastmcp, langchain, @langchain/langgraph, node-fetch or axios, testing framework (e.g., jest), potentially express.
- AI Assistant (e.g., Gemini, ChatGPT)
- API Testing Tool (Optional, e.g., curl, Postman, Insomnia)

INSTRUCTION IN NUMBERED TASKS

Goal: Explore existing MCP servers, implement an MCP proxy, integrate with a basic agent (Python or NodeJS) capable of simple RAG (exploring real-time concepts), test the proxy via IDE integration, and research advanced concepts.

Task 1: Environment Setup & Protocol Study (Est. 4 hours)

- **Objective:** Establish the local development environment and gain a thorough theoretical understanding of MCP, A2A, and the target MCP servers.
- **Steps:**
  1. **Choose Agent Path:** Decide: Python (fast-agent) or NodeJS (LangGraph.js).
  2. **Setup Environment:** Clone starter repo (if provided) or create project structure. Follow **Appendix C** to set up your chosen local environment(s). Install base dependencies (Python/Node, Git, Docker, **VS Code/Cursor**). Verify interpreters/package managers.
  3. **Create Mock KB:** Follow **Appendix D** to create the local mock knowledge base directory structure (for RAG).
  4. **MCP/A2A Deep Dive:** Read official MCP/A2A specs. Focus on MCP request/response structures.
  5. **Explore Target MCP Servers:** Review the READMEs and code for the provided MCP servers (GitHub, Filesystem, GDrive, Atlassian). Understand their purpose and basic setup instructions (many might use Docker). Note their default ports.
  6. **Document Understanding (Action 1.1):** In protocols_understanding.md, explain MCP invoke_method flow. Contrast MCP/A2A use cases. Briefly summarize the function of each target MCP server.
  7. **Commit Setup:** Commit environment files (e.g., requirements.txt, package.json, Dockerfile if used), mock KB structure, and documentation.

Task 2: Explore & Test Existing MCP Servers (Est. 6 hours)

- **Objective:** Get hands-on experience running and interacting with several pre-built MCP servers using a basic client script.
- **Steps:**
  1. **Setup Servers:** Following their documentation, set up and run at least **three** of the provided MCP servers locally (e.g., Filesystem, GitHub, Atlassian). Use Docker if recommended. Note down the local URL/port for each. *Configuration (API keys, paths) might be needed*.

2. **Build Basic Client Script:** Create a simple script (mcp_client_tester.py or .js) using the relevant fastmcp client library (or aiohttp/axios). This script should allow specifying the target MCP server URL and sending basic get_methods and invoke_method requests.
3. **Test Interaction (Action 2.1):** Use your client script to call get_methods and attempt a simple invoke_method call relevant to each running server. Print responses/errors.
4. **Document Findings:** Briefly document methods available and setup/interaction challenges in mcp_server_exploration.md.
5. **(Optional) Build Simple MCP Server:** Use fastmcp framework to build a very simple MCP server (e.g., echo method). Test with your client script.
6. **Commit Code:** Commit client tester script, optional simple server, and documentation.

Task 3: Design & Implement MCP Proxy Server (Est. 10 hours)

- **Objective:** Build a basic MCP proxy/gateway server that routes requests to the appropriate downstream MCP server.
- **Steps:**
  1. **Setup:** Create mcp_proxy_server/. Choose language (Python/FastAPI or NodeJS/Express, likely using fastmcp framework). Add dependencies. Create proxy_server.py (or .js), config.py (or .js).
  2. **Design Routing Logic:** Decide how the proxy will determine the downstream server (e.g., prefix-based /github/mcp/..., header-based X-Target-MCP, etc.). *Choose a simple approach (e.g., prefix-based).*
  3. **Configuration:** In config.py (or .js), define a mapping from your routing key (e.g., 'github') to the downstream MCP server's local URL (from Task 2).
  4. **Implement Proxy Endpoint (Action 3.1):** Create the main request handling endpoint(s). This endpoint should receive the MCP request, determine the target server, look up its URL, forward the original request payload, receive the response, and return it. Handle errors (unknown target, downstream unavailable). Add logging.
  5. **(Optional) Implement get_methods Aggregation:** Implement /mcp/get_methods on the proxy to call and aggregate get_methods from all downstream servers.
  6. **Run & Test Manually:** Run your proxy server and ensure downstream servers are running. Use your client script (or Postman/curl) to send requests *to the proxy*.

Verify routing and responses. Test error cases.

7. **Write Tests (Action 3.2):** Create test_proxy_server.py (or .js). Use testing frameworks and mocking (mock downstream HTTP calls) to test routing, payload forwarding, response returning, and error handling.
8. **Commit Code:** Commit proxy server code, configuration, tests.

Task 4: Implement Basic RAG Agent with MCP Integration (Est. 8 hours)

- **Objective:** Build an agent that uses the MCP Proxy to get data and uses basic RAG (with awareness of real-time concepts) to find related info in the local mock knowledge base.
- **Steps (Choose ONE path - Python or NodeJS):**
  1. **Setup:** Create dev_assistant_agent_py/ or dev_assistant_agent_node/. Add agent framework, RAG library (llama-index or langchain), MCP client dependencies, and potentially pathway (Python). Create agent.py/.js, rag_setup.py/.js, run.py/.js.
  2. **Implement RAG Setup (Action 4.1A/B):** In rag_setup.py/.js:
     - **LlamaIndex:** Use SimpleDirectoryReader to load documents from mock_knowledge_base. Use an embedding model (e.g., local sentence-transformers or API-based). Create a VectorStoreIndex from the documents. Create a QueryEngine from the index (index.as_query_engine()). Encapsulate this in a function/class.
     - **LangChain JS:** Use document loaders, splitters, embeddings, and a vector store (e.g., MemoryVectorStore) to create a retriever.
     - **(Conceptual) Real-time Indexing:** Review the Pathway Document Indexing Example. In a separate markdown file (realtime_rag_notes.md), briefly explain how Pathway's approach differs from the static indexing done above (e.g., watching a directory, updating the index incrementally). *No implementation required this week.*
  3. **Implement Agent (Action 4.2A/B):** In agent.py/.js:
     - Define Agent class/graph. Instantiate RAG QueryEngine/Retriever.
     - Instantiate an MCP Client configured to talk to your **MCP Proxy Server** (Task 3).
     - Implement a message handler/graph node taking a user query (e.g., "Tell me about GitHub issue #5 related to file Y").
     - Parse query to identify target tool and request details.

- **Call MCP via Proxy:** Use MCP client to send invoke_method to the proxy. Store result.
- **Call RAG:** Use the RAG query engine/retriever with the query/keywords. Store result.
- **Synthesize:** Combine MCP result and RAG context into a response.
- Return response. Handle errors. Add logging.

4. **Run Script:** Create run.py/.js to start the agent.
5. **Write Unit Tests (Action 4.3A/B):** Create test_agent.py/.js. Mock MCP Client calls and RAG query engine. Test agent logic (parsing, MCP call, RAG call, synthesis, errors).
6. **Commit Code:** Commit agent, RAG setup, run script, tests, and realtime_rag_notes.md.

1.

Task 5: Research Advanced MCP Concepts (Est. 3 hours)

- **Objective:** Explore concepts relevant to scaling and securing MCP in an enterprise environment.
- **Steps:**
  1. **Reading Task:** Read provided research articles/links (API Gateways, Security, Streaming). Review MCP Gateway inspirations (OpenMCP, Metatool).
  2. **Concept Exploration:** Research MCP Gateway possibilities beyond basic routing (auth, rate limiting, transformation), RBAC for MCP methods, and MCP Streaming concepts.
  3. **Document Design Ideas (Action 5.1):** Create advanced_mcp_concepts.md. Explain each concept (Advanced Gateway, RBAC, Streaming) in MCP context, discuss benefits/challenges for NexusAI, outline high-level conceptual designs.
  4. **Commit Documentation:** Commit advanced_mcp_concepts.md.

Task 6: Test MCP Proxy with IDE Integration (Est. 3 hours)

- **Objective:** Configure and use IDE-based MCP clients (VS Code Copilot Chat or Cursor) to interact with your MCP Proxy Server.
- **Steps:**
  1. **Ensure Servers Running:** Make sure your downstream MCP Servers (Task 2) and your MCP Proxy Server (Task 3) are running locally. Note the URL of the **proxy**

server.

2. **Configure IDE (Action 6.1):**
    - **VS Code:** Follow the Copilot Chat MCP documentation to add your running MCP Proxy Server URL to the github.copilot.chat.mcp.include setting in your VS Code settings.json. Reload VS Code.
    - **Cursor:** Follow the Cursor MCP documentation to add your MCP Proxy Server URL via its configuration interface (e.g., @mcp command or settings).
3. **Test via IDE Chat (Action 6.2):** Open the chat interface in your configured IDE (Copilot Chat panel or Cursor chat). Send queries that should utilize your proxy and downstream servers. Examples
    - "@workspace /mcp invoke_method filesystem list_files {'path': './mock_knowledge_base/code'}" (Adjust path as needed)
    - "@workspace /mcp invoke_method github get_user_info {'username': 'some_github_user'}" (Requires GitHub MCP server running & configured)
    - "@workspace /mcp invoke_method atlassian get_issue {'issue_key': 'NEX-123'}" (Requires Atlassian MCP server running & configured)
    - *(Note: Exact syntax for invoking methods might vary slightly between IDEs/versions - consult their docs)*
4. **Verify Responses:** Check if the IDE chat receives the expected JSON responses (or errors) forwarded from the downstream servers via your proxy. Check proxy server logs for incoming requests.
5. **Document Process (Action 6.3):** Create ide_mcp_integration.md. Document the steps taken to configure your chosen IDE, the test queries used, and the observed results (success/failure, responses). Include screenshots if helpful.
6. **Commit Documentation:** Commit ide_mcp_integration.md.

Task 7: Documentation & Stand-up Preparation (Est. 2 hours)

- **Objective:** Finalize documentation and prepare for stand-ups.
- **Steps:**
    1. **Update README:** Explain the overall architecture (Downstream Servers -> Proxy -> Agent -> IDE Client). Detail how to set up and run each component. Reference all documentation (protocols_understanding.md, mcp_server_exploration.md, realtime_rag_notes.md, advanced_mcp_concepts.md, ide_mcp_integration.md).

2. **Daily Stand-up Prep:** Prepare concise daily updates (Done / Doing / Blockers).
3. **Final Commit & Push:** Ensure all code, tests, documentation are committed and pushed.
4. **Self-Reflection:** Write a brief paragraph reflecting on Week 1: Challenges (MCP servers, proxy, RAG, Pathway concepts, IDE integration)? Most powerful concept? Remaining questions? AI usage?

LLM BASED SELF LEARNING/PRACTICING PATHWAY/GUIDELINE

- **MCP Concepts (Task 1, 2, 3):**
  - "Explain the request and response structure for MCP invoke_method."
  - "How can I use the Python jlowin/fastmcp library to make a client call to an MCP server at http://localhost:8001?"
  - "Show an example of using Python FastAPI (or Node.js Express) to receive a POST request, forward it to another URL based on a path parameter, and return the response." (Proxy core logic)
  - "What are common strategies for routing requests in an API gateway or proxy?"
- **Agent Framework & RAG (Task 4):**
  - (Tailor to chosen framework: fast-agent or LangGraph.js) "Show the basic structure for defining an agent..."
  - "Provide a LlamaIndex example using SimpleDirectoryReader, VectorStoreIndex, and as_query_engine()."
  - "Explain the core idea behind real-time RAG indexing as shown in the Pathway demo-document-indexing example."
  - "How can my agent parse a user query like 'Get GitHub issue #50' to determine the target service ('GitHub') and parameters ({'issue_number': 50})?"
- **Testing (Task 3, 4, 6):**
  - "How can I use pytest and aioresponses (or httpx.mock) to test an async Python function that makes an HTTP call?"
  - "How can I mock external HTTP calls in Jest for testing a Node.js function?"
  - "How do I configure VS Code Copilot Chat to use a local MCP server running at http://localhost:8002/proxy/mcp?"
  - "What is the syntax for invoking an MCP method via Cursor's chat?"
- **Critical Evaluation Reminder:** Use AI for syntax, library examples, concepts. Always verify, adapt, understand, and test generated code, especially proxy routing, RAG pipelines, and IDE configurations.

TUTORIALS

- MCP: Official MCP Documentation
- A2A: Official A2A GitHub Repository/Docs
- MCP Frameworks: jlowin/fastmcp README, punkpeye/fastmcp README
- Target MCP Servers: See READMEs in their respective GitHub repositories.
- FastAPI: FastAPI Tutorial (If using Python for proxy)
- Express: Express "Hello World" Example (If using Node.js for proxy)
- AIOHTTP/HTTPX/Axios/Node-fetch: Relevant library documentation.
- Agent Frameworks: fast-agent Documentation, LangGraph.js Documentation
- RAG Frameworks: LlamaIndex Python Docs, LangChain JS RAG Docs
- **Real-time RAG:** Pathway LLM App Repo, Pathway Document Indexing Example
- **IDE MCP Clients:** VS Code Copilot Chat MCP Docs, Cursor MCP Docs
- Testing: pytest Docs, pytest-asyncio Docs, aioresponses Docs, Jest Docs, Nock (HTTP Mocking)
- Advanced Concepts Reading: (Links provided in Data section)

DELIVERABLES

- **Git Repository:** Link to your GitHub repository containing:
  - MCP Client Tester script (mcp_client_tester.py or .js).
  - MCP Proxy Server code and tests.
  - Dev Assistant Agent code (Python/fast-agent or NodeJS/LangGraph) including RAG logic, and tests.
  - Updated requirements.txt and/or package.json.
  - Configuration files (e.g., proxy config).
  - Documentation (README.md, protocols_understanding.md, mcp_server_exploration.md, realtime_rag_notes.md, advanced_mcp_concepts.md, ide_mcp_integration.md).
  - Your self-reflection paragraph for Week 1.
  - Mock knowledge base files/structure (as per Appendix D).
- **Report:** A summary of your research and work

APPENDICES

Appendix A: Protocol Overviews (Simplified)

Model Context Protocol (MCP) Summary:

- **Purpose:** Standardizes how AI agents access external tools and data sources (like APIs, databases, files). It acts like a universal adapter.

- **Problem Solved:** Prevents needing custom code for every single tool an agent might use. Creates a consistent way for agents to discover and interact with tools.
- **Key Components:**
    - **MCP Server:** Wraps around an existing tool/API and exposes its functionality via the MCP standard methods.
    - **MCP Client:** Used by the agent to communicate with MCP Servers.
    - **Methods:** Standardized operations like get_methods (discover what the tool can do), invoke_method (use a specific tool function).
- **Interaction:** Agent -> MCP Client -> MCP Server -> Tool/API

Agent-to-Agent (A2A) Protocol Summary:

- **Purpose:** Enables different AI agents (potentially built by different teams or using different frameworks) to communicate and collaborate securely and effectively.
- **Problem Solved:** Lack of a standard way for agents to talk to each other, discover capabilities, and orchestrate complex workflows involving multiple specialized agents.
- **Key Components:**
    - **Transport:** Uses common web technologies like HTTP/WebSockets.
    - **Messaging Format:** Primarily uses JSON-RPC 2.0 for requests/responses.
    - **Asynchronous Events:** Uses Server-Sent Events (SSE) for push notifications/updates from one agent to another.
    - **Discovery:** Uses "Agent Cards" – standardized descriptions of an agent's capabilities, allowing other agents to find and understand how to interact with them.
    - **Security:** Recommends standards like OAuth 2.1 / OpenID Connect for secure authentication and authorization between agents.
- **Interaction:** Agent 1 -> Agent 2 (via A2A Interface)

Appendix D: Mock Data Source Setup

Objective: Create local simulated data for JIRA, GitHub, and Docs to be used by the MCP server and RAG pipeline.

Instructions:

1. **Create Base Directory:** In your project root, create a directory named mock_knowledge_base.
2. **Mock JIRA Data:**
   - Create a file mock_knowledge_base/jira_tickets.json.
   - Populate it with a JSON array of mock ticket objects. Example:

```
[
  {
    "ticket_id": "NEX-123",
    "summary": "Fix login button alignment on mobile",
    "description": "The login button is slightly off-center on screens smaller than
480px.",
    "status": "Done",
    "assignee": "dev_a",
    "reporter": "qa_b",
    "code_refs": ["commit_abc123", "PR #45"],
    "doc_refs": ["ui_guidelines.md", "login_feature.md"]
  },
  {
    "ticket_id": "NEX-456",
    "summary": "Implement MCP server for Task API",
    "description": "Wrap the internal Task Management API with an MCP server.",
    "status": "In Progress",
    "assignee": "protocol_eng_c",
    "reporter": "lead_eng",
    "code_refs": ["commit_def456", "PR #52"],
    "doc_refs": ["mcp_server_design.md", "task_api_spec.md"]
  },
  {
    "ticket_id": "NEX-789",
    "summary": "Research A2A security models",
    "description": "Investigate OAuth and other options for securing agent
communication.",
    "status": "To Do",
    "assignee": null,
    "reporter": "arch_d",
    "code_refs": [],
  "doc_refs": ["a2a_spec.md"]
 }
]
```

- Your MCP JIRA Server (Task 2) will read this file.

3. **Mock GitHub Data (Code Snippets):**
   - Create a subdirectory mock_knowledge_base/code/.
   - Create simple text files representing code commits or snippets mentioned in JIRA tickets. Example mock_knowledge_base/code/commit_abc123.py:
     ```
     # Fix for NEX-123: Adjusted login button CSS
     def apply_mobile_styles(button_element):
       if screen_width < 480:
         button_element.style.marginLeft = 'auto';
         button_element.style.marginRight = 'auto';
       # ... other styles
     ```
   - Create another for commit_def456.py.
4. **Mock Google Drive Data (Docs):**
   - Create a subdirectory mock_knowledge_base/docs/.
   - Create simple markdown files representing documentation mentioned in JIRA tickets. Example mock_knowledge_base/docs/login_feature.md:
     ```
     # Login Feature Documentation

     This document describes the user login flow.
     ## UI Elements
     - Username field
     - Password field
     - Login Button (See ui_guidelines.md for styling)
     ## Known Issues
     - Alignment on mobile (NEX-123)
     ```
   - Create other files like ui_guidelines.md, mcp_server_design.md, etc., with brief relevant content.
5. **Mock JIRA Summaries (for RAG):**
   - Create a subdirectory mock_knowledge_base/tickets/.
   - For each ticket in jira_tickets.json, create a simple text file named NEX-XXX.txt containing its summary and description. Example mock_knowledge_base/tickets/NEX-123.txt:

Ticket: NEX-123
Summary: Fix login button alignment on mobile
Description: The login button is slightly off-center on screens smaller than 480px.
Status: Done
Refs: commit_abc123, PR #45, ui_guidelines.md, login_feature.md

6. **RAG Target:** Your RAG pipeline (Task 4) should be configured to index all files within the mock_knowledge_base directory (including subdirectories).

Appendix D: IT Setup Guide

(This should be provided to the trainee)

Objective: Set up a consistent Dockerized Python development environment.

Prerequisites:

- Docker Desktop installed and running.
- Git installed.
- A terminal or command prompt.
- A code editor (like VS Code) with Docker integration (recommended).

Steps:

1. **Clone Repository:**
```
git clone <instructor_provided_repository_url> week1-challenge
cd week1-challenge
```

2. **Review Files:** Familiarize yourself with the provided files:

- Dockerfile: Defines how to build the Docker image with Python and dependencies.
- docker-compose.yml: Defines the service(s) to run using the Docker image (makes running easier).
- requirements.txt: Lists Python libraries needed (initially might include pytest, pytest-asyncio, aiohttp, and the chosen agent framework like fastagent or letta).
- Potentially skeleton Python files (worker_agent.py, etc.).

1. **Build and Start Container:** Open your terminal *in the week1-challenge directory* and run:
   docker-compose up --build -d

- --build: Tells Docker Compose to build the image based on the Dockerfile if it doesn't exist or if the Dockerfile changed.
- -d: Runs the container in detached mode (in the background).
- *Troubleshooting:* If the build fails, check the output for errors (e.g., typos in requirements.txt, network issues).

1. **Access Container Terminal:** You need to run commands *inside* the container.

- Find the container name/ID: docker ps (Look for an image name related to the project, e.g., week1-challenge_app_1).
- Execute a bash shell inside the container:
  docker exec -it <container_name_or_id> /bin/bash

  You should now see a prompt like root@<container_id>:/app#. You are inside the container!

1. **Verify Environment:** Inside the container terminal, check installations:
   python --version
   pip list | grep pytest # Check if key packages are installed

2. **Working with Code:**

- The docker-compose.yml likely sets up a volume mount, meaning the code files in your week1-challenge directory on your host machine are mirrored inside the container (usually at /app).
- **Edit code on your host machine** using your preferred code editor (VS Code).
- **Run code and tests inside the container terminal** you opened in step 4. For example:
  # Inside the container terminal
  python async_practice.py
  pytest test_async_practice.py

1. **Installing New Packages:** If you need to add a library (like aiohttp):

- Add the library name (e.g., aiohttp) to the requirements.txt file on your host machine.
- Stop the running containers: docker-compose down (in the terminal on your host machine, in the project directory).
- Rebuild and restart: docker-compose up --build -d
- Access the container terminal again (step 4). The new package should be installed.

Stopping the Environment: When finished for the day, run this command in the terminal on your *host machine* (in the project directory):
docker-compose down