**MCP Definition:**

MCP (Model-Context Protocol) is a framework for defining and orchestrating prompt-driven interactions between clients, tools, and services based on contextual data (e.g., lifecycle stage, domain, task), where:

* Model = the data structure or schema (e.g., clam batch, QA record)
* Context = where in the lifecycle you are, or what operation needs to be performed (e.g., depuration → QA → shipping)
* Protocol = the standardized way of describing, discovering, and invoking tools/prompts (across domains or systems)

**MCP Scope in Project:**

**Build a pluggable, prompt-aware MCP framework** where:

* Clients **discover and use prompts/tools dynamically**
* Prompt libraries are **registered via MCP framework**
* MCP servers are **domain-specific**, flexible, and can use **microservices, ESB, SOA/SOAP**, or **platform-native** integrations (e.g., Databricks, stored procs)
* The entire system is **modular, discoverable, and data-lake-ready**



[MCP Client]

|

|-- Discovers available prompts via Registry

|-- Selects a tool based on current context

v

[Prompt Discovery Service] ←→ [Prompt Library Registry]

|

|-- Describes: input schema, output schema, execution method

v

[MCP Server Registry] ←→ [MCP Server Instance (Domain X)]

├─ Microservices

├─ SOAP/SOA Endpoints

├─ ESB Routes

├─ Data Lake (Databricks, etc.)

└─ DB Stored Procedures

[MCP Server: clam\_qc]

|

|-- Executes logic on model based on context

|

V

[Response to Client]

**Advantage of Model-Context-Protocol:**

 Prompts act as protocol units that take models and contexts as input

 Everything is discoverable, composable, and stateless

 Supports plug-and-play architectures across domains

Overall smart design choice — emphasizes structured, contextual, prompt-based orchestration across clients and services.

**I. MCP Client:**

**Key Components:**

\* Discover available prompts/tools from a central registry

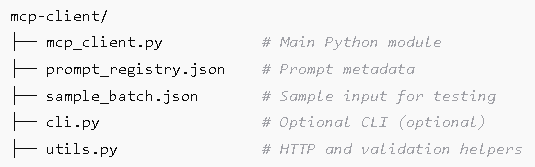
\* invoke domain-specific **MCP servers** based on context or lifecycle stage

\* Supports **structured prompt input/output** (JSON/YAML preferred)

**Implementation:**

Use OpenAPI or gRPC for client-server communication

**Directory Structure:**

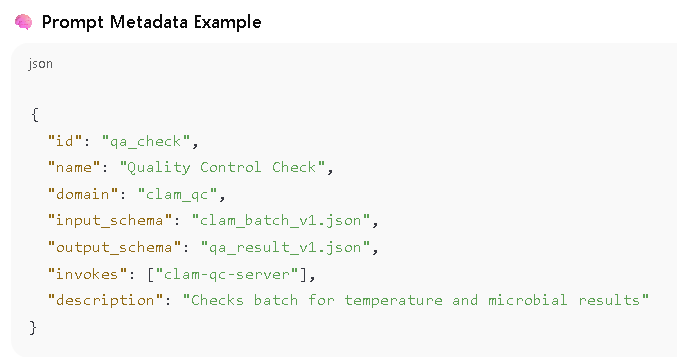


**II.PROMPT LIBRARY:**

\* A centralized registry of prompts, tools, workflows

\* Prompts are tagged and categorized by domain, type (validation, transformation, lifecycle), and input/output schema

\* Discoverable by clients using MCP tooling

****

**Implementation :**

\* Store in Git, JSON file, or database

\* Expose via HTTP API (prompt discovery service)

**III.** **MCP SERVER FRAMEWORK**

* MCP Servers can be developed independently by domains
* Can be built using any one or by group:
  + Microservices (REST/gRPC APIs)
  + ESB routes (Apache Camel, MuleSoft, etc.)
  + SOAP/SOA services (for legacy)
  + Platform-native workflows (Databricks notebooks/jobs, Airflow DAGs, stored procs)
* Each server registers capabilities and tools to the framework (prompt library)

🔧 **Implementation** :

* Use OpenAPI + Swagger for microservices
* Wrap SOAP/WSDL with microservice facades for modern consumption
* Define tool runners for platform-specific tools (e.g., Databricks job runner)

**IV. MCP TOOLS / RESOURCE FRAMEWORK**

\* Set of reusable utilities for:

* Prompt execution orchestration
* Input validation / schema matching
* Logging / tracing / auditing
* Authentication / authorization

\* Used by both clients and servers

🔧 **Implementation** :

\* Python SDK (mcp\_tools)

\* Shared Docker base image for MCP servers

\* Tool runner abstraction:

**V. DATA LAKE + DB INTEGRATION ( Data Alex):**

\* MCP servers can push/pull to/from data lakes:

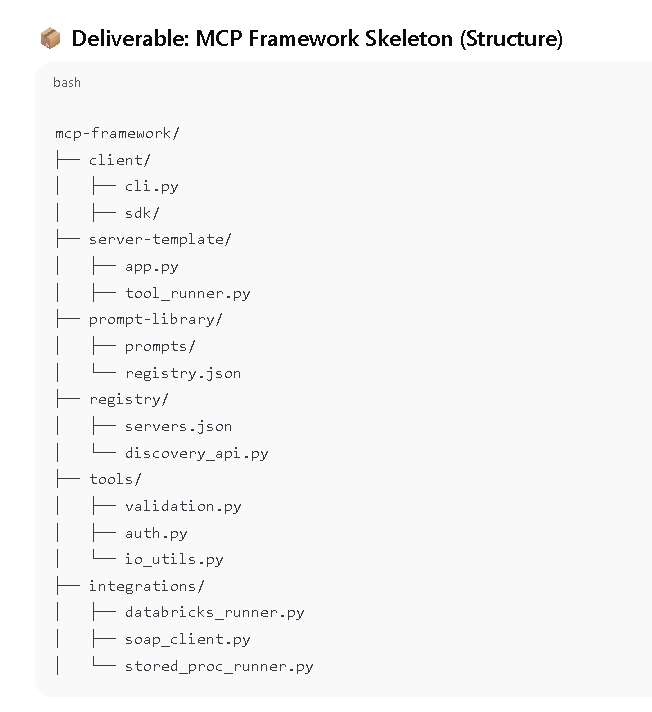
* Databricks, S3, ADLS, GCS

\* Invoke stored procedures or run SQL transformations

\* Output audit-ready logs and data lineage

**🔧 Implementation Ideas**

* Databricks integration:
* Abstract storage via connectors:

****