LLMC MCP Integration System – Technical Research & Design Report

1. Product Design Document (PDD)

Problem Statement

Large Language Models (LLMs) currently have limited direct access to enterprise data and system operations. In a corporate setting, critical data sources (filesystems, databases, ERP systems, etc.) remain siloed from LLMs due to security and integration complexity. Existing stop-gap solutions require custom coding or manual data injection, which is neither scalable nor secure. The **Model Context Protocol (MCP)** addresses this by defining a standard way to expose tools and data to LLMs via an API. However, current MCP implementations are fragmented or platform-specific, making it hard to integrate *any* LLM (OpenAI GPT-4, Anthropic Claude, Google Gemini, MiniMax, etc.) with *any* MCP-compatible tool server in a production-grade way. 082525

Our goal is to design an **LLMC (Large Language Model Controller) integration system** that bridges this gap. LLMC will serve as the orchestrator enabling LLMs to invoke filesystem and system operations through MCP servers, effectively giving the LLM controlled access to enterprise tools and data. The solution should match the capabilities of “Desktop Commander” (a prototype local PC automation via LLM) but in a robust, enterprise-ready form. It must handle the diversity of LLM APIs and ensure secure, reliable communication with MCP servers. In summary, we need a unified, secure, and extensible system so that *any LLM can execute authorized*

operations on enterprise systems in context, solving the current barriers to LLM integration.

Market Analysis

Landscape of MCP Integrations: The concept of MCP is emerging as a de facto standard for tool-augmented AI. Microsoft's Copilot ecosystem uses Graph Connectors and the MCP to integrate business data into AI experiences. For example, Oracle NetSuite provides an *MCP Standard Tools SuiteApp* so that an AI client can interact with ERP records via a standardized interface. Google has also embraced this trend, releasing an open-source MCP server for their Google Ads API, allowing a Copilot or other AI assistant to query Ads data in real time. GitHub, targeting developers, introduced a remote MCP server for code context integration in editors. Daily huddle-202… +1

Switch from MCP Sa… R&D LLM Request MCP gets easier, cod…

Gaps in Existing Solutions: Early solutions like **Desktop Commander** (a community tool to let ChatGPT control a desktop) demonstrated the potential but are not enterprise-ready – they often require custom setups, have security shortcomings (essentially giving the LLM free rein on a machine), and are tied to specific LLMs or UIs. Similarly, **"Claude Desktop"** prototypes exist (Anthropic Claude controlling local apps), but these are unofficial and not standardized. Each vendor's approach is slightly different (Microsoft's closed ecosystem, Oracle's product-specific tools, etc.), which can **fail to accommodate heterogeneous environments** where multiple LLMs and data sources need to interoperate. No current solution provides an *agnostic controller* that can broker between any LLM and any MCP server seamlessly.

Need for a Unified System: Given this landscape, a bespoke LLMC integration system can fill the gap by being LLM-agnostic and tool-agnostic. It can take inspiration

from the successes of existing implementations (ease of setup in GitHub's MCP server, rich tool sets in NetSuite, etc.) while avoiding their limitations (closed platform tie-in, lack of cross-LLM support, etc.). By doing so, LLMC can become the enterprise's central hub for AI tool usage, increasing the usefulness of LLMs significantly.

MCP gets easier, cod… Switch from MCP Sa…

User Stories

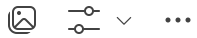
- **LLM Developer:** As an AI platform engineer, I want to easily connect a new LLM (from any provider) to our company's data and operations, so that the LLM can answer questions and perform tasks using real tools (file access, process execution) without custom-coding each integration.
- **Data Analyst:** As a data analyst using LLMC, I can ask an LLM "What's the content of the latest error log?" and the LLM will use the filesystem tool to retrieve the log file and summarize it, all in one conversation. I don't need to manually fetch the file myself.
- **IT Administrator:** As an IT admin, I want fine-grained control over what actions the LLM can take on our servers. I can allow read-only file access but not file deletion, and see audit logs of every tool invocation the LLM made.
- **Business User:** As a business user (with appropriate permissions), I can query "What were our top 5 products last month?" and the LLM will use an MCP connector to our ERP's reporting tool to fetch the data and answer, saving me from running a separate report.
- **DevOps Engineer:** As a DevOps engineer, I want to run an "AI agent" mode where the LLM can execute shell commands (via an MCP process tool) to assist in automating deployments. The LLMC ensures those

commands run in a safe sandbox and returns results to the LLM.

These scenarios highlight personas from developers to non-technical users, all benefiting from an LLM that can *act* on their behalf through approved tools.

Success Metrics

We will measure the success of the LLMC MCP integration by the following key metrics (KPIs):



Other success criteria include: **ease of integration** (e.g., a new MCP tool server can be added in <1 day of development time), **security incidents = 0** (no unauthorized operations via the LLM), and **positive user adoption** (measured by number of queries using tools, and user feedback on usefulness). We will also track system resource usage and error rates. For example, if any tool call fails, it should be properly handled and not confuse the user – we aim for >99% successful task

completion rate when the underlying systems are functioning.

Competitive Analysis

Desktop Commander vs LLMC: Desktop Commander paved the way by allowing a chatbot to control a desktop, but it is a monolithic solution tied to specific LLMs and not enterprise-grade. LLMC goes further by using the MCP standard, enabling *any* tool across networked servers (not just a single desktop) and working with multiple LLMs. Unlike Desktop Commander, which might not enforce strict permissions, LLMC will incorporate authentication and role-based access on tools.

Claude's and Others' Integrations: Some community efforts have connected Anthropic's Claude to local tools ("Claude Desktop"), but these often involve ad-hoc prompt engineering rather than a formal protocol. They lack the structured interface MCP provides. LLMC's advantage is in the formal MCP layer: the LLM doesn't need to "figure out" how to run a tool from natural language; it will have a predefined list of tools with structured inputs.

Microsoft 365 Copilot and AI Connectors: Microsoft's Copilot can interface with external systems via something analogous to MCP (Graph connectors and plugins). However, those are mostly available within Microsoft's ecosystem. Our LLMC is vendor-neutral and can integrate similar connectors (for example, hooking up the same NetSuite tools Copilot would use) but also support other LLMs beyond Copilot. This gives an organization flexibility to use OpenAI, Anthropic, or local open-source LLMs with the same connectors, avoiding lock-in. Switch from MCP Sa…

GitHub's Remote MCP Server: GitHub's approach focuses on developer tools – providing a server that integrates with VS Code and GitHub Copilot. It simplifies setup and adds authentication and auto-updates, which

are strengths we aim to incorporate (e.g., easy auth configuration and upgradability). However, GitHub's solution is tied to code editing scenarios. Our LLMC will be more general, handling not just code or repos but also files, processes, corporate databases, etc., in a unified way. MCP gets easier, cod…

OpenAI Plugins & LangChain Agents: OpenAI's plugin system and frameworks like LangChain allow LLMs to call APIs. These are somewhat analogous: an OpenAI plugin could expose a filesystem API. However, those require specific support in the LLM (OpenAI's server) and operate on a per-LLM basis. LangChain agents can execute Python code or hit REST endpoints but require running a Python environment alongside the LLM. LLMC's MCP integration offers a *standardized JSON-RPC interface* that can be tapped by any LLM that supports function calling, without needing a full Python runtime with the LLM. This is more lightweight and consistent across different AI systems.

In summary, **LLMC's integration system differentiates by offering a single, standardized gateway for tool usage, while competitors either tie to one AI platform or one domain of tools.** We combine the breadth of available operations (like Desktop Commander and Copilot) with the flexibility of open standards (MCP) and multi-LLM compatibility, which is currently unmatched in one package.

Risk Assessment

Other Risks: Compliance and data governance must be considered – e.g., ensuring that using these tools does not expose data to the LLM beyond policy (MCP results will be treated as user-provided context to the model, and per corporate policy, the model's outputs remain company property). There is also a risk of **user misunderstanding**: users might over-rely on the LLM's actions. We mitigate this by thorough documentation and perhaps prompting the LLM to explain its actions. Draft Revised 2025 A…

Finally, **change management** is a risk – introducing this system into workflows requires training and trust. We plan

to start with read-only, low-risk use cases to build confidence (e.g., the Google Ads query scenario as a pilot). Gradually, we'll expand capabilities as users and stakeholders become comfortable. R&D LLM Request

By identifying these risks early and planning mitigations, we aim to deliver a solution that is not only powerful but also safe, reliable, and aligned with business needs.

---

2. High-Level Design (HLD)

System Architecture

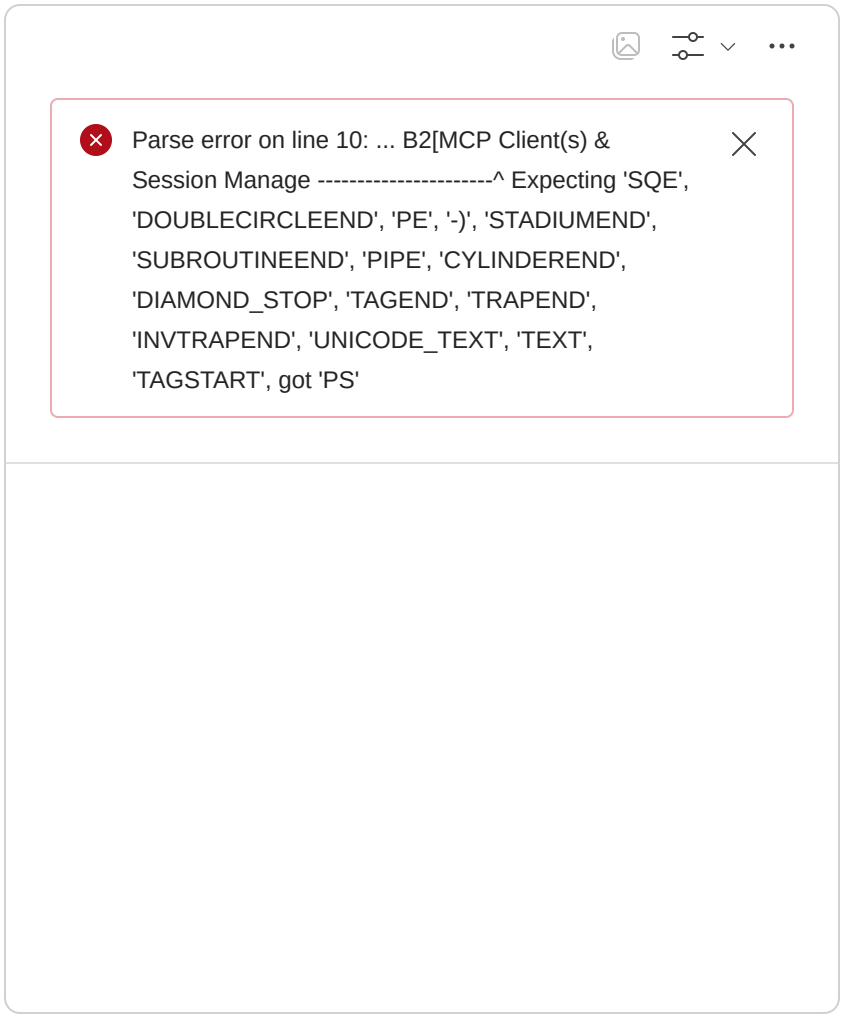
At a high level, the LLMC integration system sits as an intermediary between LLMs and one or more MCP servers (which in turn interface with actual systems and data). The architecture comprises several key components:

- **LLM Controller (LLMC Core):** Orchestrates LLM conversations and tool usage. This includes the function-calling interface to the LLM and logic to decide which MCP server/tool to invoke when an LLM requests an action.
- **MCP Client Module:** Responsible for communicating with MCP servers using the MCP protocol (HTTP + SSE). It handles sending requests (like "call this tool with these params") and receiving responses/events.
- **Translator:** Translates tool definitions from MCP format into the format each LLM expects (OpenAI function specs, Anthropic tool format, etc.), and vice versa translates LLM function call results into MCP requests.
- **Session & State Manager:** Manages MCP sessions and any needed state so that multi-turn conversations can maintain context (like an open process or a search in progress). It stores the active session IDs for each

MCP server and tracks tool-related state if needed (e.g., process IDs).

- **Tool Executor/Registry:** Presents a unified list of all available tools from all connected MCP servers, and routes each tool call to the appropriate MCP server. This layer abstracts multiple servers so the LLM sees one “toolbox.”

Below is a simplified component diagram of the system:



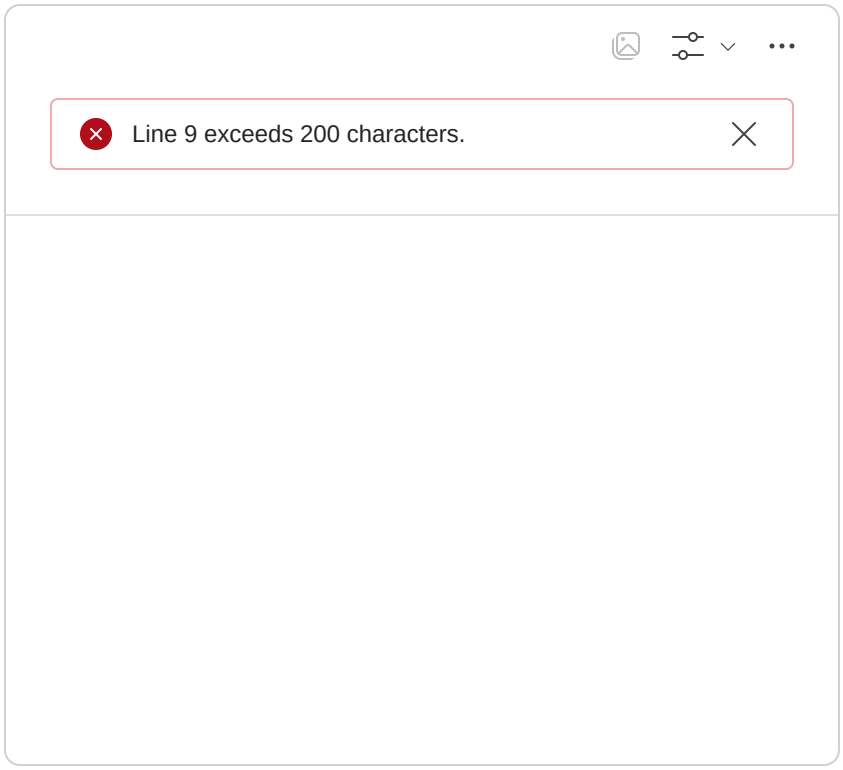
Explanation: The LLM clients (which could be OpenAI’s API, Anthropic’s API, etc.) are configured with function/tool definitions provided by LLMC. When an LLM wants to use a tool, it sends a function call back to LLMC. LLMC’s Tool Registry/Executor determines which MCP server has that tool and forwards the request via the MCP Client. The MCP Server then executes the operation on the underlying

system (e.g., reading a file, querying a DB) and streams back results. LLMC receives those results and forwards them to the LLM as the function's result.

This design cleanly separates concerns: LLM-specific logic is in the Translator, protocol handling in the MCP Client, and decision logic in the Executor. It is extensible – we can add another MCP server or another LLM client by adding to the respective subcomponent without affecting the others.

Data Flow

The following sequence diagram illustrates the typical data flow for an LLM using a tool via LLMC and MCP:



1. **LLM initialization**: LLMC first provided the LLM with a list of available tools (functions). So when the user prompt comes in, the LLM knows it can use "read_file" as a function.
2. **User Prompt**: The user asks a question that requires a tool (e.g., asking about a file's content).

3. **LLM decision**: The LLM decides to invoke the `read_file` function rather than answer from its own knowledge. It pauses its normal response and returns a function call request to LLMC. For example, OpenAI's API would return a `{"function_call": {"name": "read_file", "arguments": "{\"path\": \"...\"}"}}` message.
4. **LLMC forwards to MCP**: LLMC receives this and uses the MCP Client to call the corresponding tool on the appropriate server. In this case, it sends an HTTP request to the Filesystem MCP server's `tools/call` endpoint for `read_file` with the parameter.
5. **MCP executes operation**: The Filesystem MCP server (which might be running on an on-premises VM) reads the file from disk. Daily huddle-202510… 082525
6. **Streaming result**: The file content could be large, so the MCP server streams it back via Server-Sent Events (SSE) in chunks. LLMC's MCP Client module receives these SSE events, concatenates the chunks, and once complete, has the full result (or it could stream partial data to the LLM if the LLM supports streaming function results). Daily huddle-202510…
7. **LLMC returns function result**: LLMC packages the file content as the return value of the `read_file` function and sends it to the LLM. In OpenAI's case, this means sending a new assistant message with role "function" and content containing the file text.
8. **LLM responds to user**: Now with the file content in context, the LLM continues its process. It might either directly return the file content (if the user explicitly asked for it) or produce a summarized answer. The final answer is then delivered to the user.

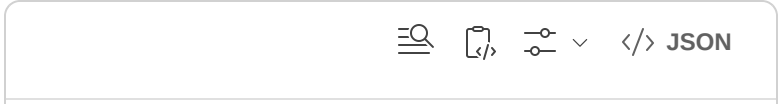
This flow ensures the LLM never hallucinated the file's content; it actually retrieved it via the tool. The user's original question is answered using real data from the system.

Alternate flows: If an error occurs (e.g., file not found), the MCP server would return an error event. LLMC would catch that and inform the LLM (possibly as a function result indicating the error). The LLM could then tell the user "Sorry, I couldn't find that file." Another scenario: multi-step tool usage – e.g., the LLM might call a search tool first, then use a result to call a read tool. The system supports this, as the session manager can preserve context or results between calls if needed (the LLM simply makes multiple function calls in sequence within the same conversation).

Protocol Specification (MCP 2024-11-05)

We adopt the **Model Context Protocol (MCP)** as of the 2024-11-05 spec for communication between LLMC and MCP servers. Key aspects of this protocol:

- **Transport**: HTTP for requests and Server-Sent Events (SSE) for asynchronous responses. This means LLMC sends HTTP POST requests to the MCP server for tool invocations, and the server keeps the connection open to stream back results/events in real-time. SSE was chosen over alternatives like WebSockets for its simplicity and compatibility with HTTP infrastructure (no special upgrade needed) – it's well-suited for one-way event streams from server to client, which matches tool output streaming.
- **Message Format**: JSON-RPC 2.0 is used to structure requests and responses. Every request has a JSON body with the format:



```
{
  "jsonrpc": "2.0",
  "id": <request_id>,
  "method": "<tool_or_operation>",
  "params": { ... }
}
```

For example, calling `read_file` might send:

```
{"jsonrpc": "2.0", "id": 1, "method":
"tools/call",
 "params": {"tool": "read_file",
"arguments": {"path":
"/path/to/file.txt"}}}
```

The use of JSON-RPC allows correlating responses with requests via the `id` field, though in our case we often handle one call at a time per SSE stream.
The **tools/list** operation (to fetch available tools) is similarly a JSON-RPC method (`"method": "tools/list"`).

- **Session Initialization**: On first connection to an MCP server, LLMC will typically call an *initialize* method (or include an implicit session create in the first call). The MCP server returns a `sessionId` (often in a header like `Mcp-Session-Id` or in the JSON response). This session ID is then included in subsequent requests (e.g., as a header or part of JSON params) to tie them to the same session context. The session context allows the server to maintain state, such as open processes or search results, across multiple tool calls. Our system will manage these session IDs automatically in the Session Manager.

  Daily huddle-202… +1

- **Tool Listing**: When LLMC requests the list of available tools, the MCP server responds with a JSON structure describing each tool. For example:


```
{
  "jsonrpc": "2.0", "id": 2,
"result": {
    "tools": [
       {
         "name": "read_file",
         "description": "Read the
contents of a text file from the
server",
         "parameters": {
             "type": "object",
             "properties": { "path":
{"type": "string", "description":
"File path"} },
             "required": ["path"]
```


Show more lines

The exact schema may vary, but generally it provides the tool name, a human-readable description, and a JSON Schema for parameters. This schema is crucial as it can be directly converted to function signatures for different LLMs.

- **Calling Tools**: To invoke a tool, we send a `tools/call` request with the tool name and arguments as shown above. The server will start executing the tool and immediately return a response to acknowledge the call (possibly something like `{"id":1,"result":"accepted","session":"<id>","toolRunId": "<id>"}`) and then stream results. In practice, some implementations might simply not send a full JSON response but just begin SSE events.
- **SSE Response Stream**: Events sent over SSE have the format:

```
event: <eventType>
data: <JSON payload>
```

For example, one event type might be `result` or `output` carrying chunks of output, and a final event could be `result_end` or the stream simply closes when done. The GitHub MCP server and others often just send `data:` lines with JSON. We will parse events as they arrive:

- Partial outputs might be marked with a specific field or event type (for instance, a long file read may stream data in multiple `output` events).
- A final result (or error) might be a JSON object that the client interprets as completion.

For simplicity, our design will treat the entire content between call and connection close as the tool's output. We will consider the stream finished when the connection closes or a special event (like `{"status": "complete"}`) is received.

- **Error Handling**: If a tool fails or is not found, the MCP server will send an error in JSON-RPC format. For example:

or using the reserved range -32000 to -32099 for server errors. Our system will handle common error codes: Daily huddle-202510…

- `-32601`: Method/Tool not found – likely indicates a misnamed tool or outdated list.

- `-32602`: Invalid params – e.g., missing a required argument, we might log and inform the user to correct the prompt.
- `-32000` to `-32099`: Application-specific errors (e.g., file not found might use -32001, permission denied -32002, etc.). We will map these to user-friendly error messages or structured error results to the LLM (so the LLM can say "I couldn't read the file: access denied.").
- In SSE, errors might be sent as an event or as a final data chunk with an error field.

Additionally, timeouts are a concern: if a tool is taking too long, LLMC may abort the request (closing the connection) and inform the LLM that it timed out.

- **Security**: MCP supports an authentication mechanism (e.g., an `Authorization` header or a custom header like `Mcp-Auth-Token`). In our system, if an MCP server requires a secret key (e.g., our filesystem MCP might be configured with `MCP_AUTH_SECRET_KEY`), the MCP Client will include the appropriate header in each request. This prevents unauthorized clients from invoking tools. We'll make this configurable per server.

This protocol design ensures that the conversation between LLMC and the MCP servers is stateless between requests (each request has all needed info including session) except for the session context managed on the server side. This statelessness at the transport level means we can scale out (multiple LLMC instances can talk to the same MCP server if needed, each with different session IDs).

Session Management

Sessions play a crucial role in maintaining context over multiple tool calls:

- **Lifecycle**: LLMC will initiate a session with each MCP server either at startup or on first use. For example, when LLMC starts up, it might call `initialize` on the Filesystem MCP and receive session `ABC123`. It will store this. As long as that session is valid, subsequent calls include `ABC123` so that, for instance, an `edit_file` tool knows which virtual workspace or context to use. Sessions can expire or be closed; our Session Manager will detect if a session expires (e.g., if an MCP server returns a session error) and reinitialize as needed.
- **Stateful Interactions**: Some tools are inherently stateful. For example, `start_process` might launch a process and associate it with the session. The MCP server might maintain a list of running processes per session. Another example is a search: a `start_search` tool could begin a long-running search and later calls like `get_more_results` refer to that search context. The session ties these together on the server side.
- **Session storage**: On the LLMC side, we will store minimal session info: essentially a mapping of server -> session ID, plus any metadata if needed (like last activity timestamp if we implement keep-alive). All heavy state (process handles, search results, etc.) remains on the MCP server and is identified by the session and perhaps a handle ID returned by the tool (e.g., process ID).
- **Thread Safety**: Our design ensures that each conversation/user has their own session context. If multiple user sessions hit the same LLMC and tools, they will either get separate sessions or carefully segregated usage of one session depending on context isolation needs. Typically, we'll use separate sessions per *concurrent conversation* per server to

avoid cross-talk. The Session Manager will index by (conversation or user) and server.

- **Session Termination**: We will provide a mechanism to gracefully close sessions if needed (for example, if a user conversation ends or times out). This can trigger the MCP server to clean up any state (kill leftover processes, free resources). The Session Manager might call a `close_session` method or simply let the session idle out if the server auto-expires it.

Session Example: If a user asks the LLM to start a terminal session and run a few commands, the LLM might call `start_process("bash")`. The MCP server (filesystem/OS server) will start a shell and keep it running, associated with session ID `ABC123` and returning a process handle. The user then says "run `ls`", the LLM calls `interact_with_process(handle, "ls\n")` – the Session ID `ABC123` ensures the server knows which shell process to send this to, and it returns the output. Finally, if the user ends the session, LLMC calls `force_terminate(handle)` or `close_session`, and the server kills the shell. Throughout this, Session Manager made sure to use the correct session and handle IDs.

By transparently handling sessions, LLMC will allow LLMs to use persistent tool contexts (like an open shell, or an ongoing database transaction) across multiple calls in one conversation, which greatly expands the power of what the LLM can do (e.g., perform a multi-step procedure via tools).

Multi-Server Support

The design accounts for multiple MCP servers providing tools:

- **Tool Discovery**: LLMC can connect to multiple MCP endpoints. For instance, one server might be

a *Filesystem MCP* (exposing tools like read/write file, list directory, etc.) and another might be an *ERP MCP* (exposing tools like `query_sales_data`, `update_record` for an Oracle NetSuite account). On startup, LLMC will query each server for its tool list and then merge these into a single catalog of tools. Tools are identified by unique names. In case of name collisions (two servers have a tool with the same name), LLMC will disambiguate by namespacing (e.g., prefixing with server name) or by configuration (e.g., alias one of them). The user or LLM can then refer to the tool via the unique name.

Switch from MCP Sa…

- **Routing**: When the LLM calls a tool, LLMC's Tool Registry knows which server it came from. For example, `read_file` -> filesystem server, `create_sales_order` -> ERP server. The mapping is maintained in a registry dictionary. The call is routed to the appropriate MCP Client instance for that server.
- **Concurrent Servers**: LLMC can maintain simultaneous sessions with each server. For example, session `ABC123` on the filesystem server and session `XYZ789` on the ERP server, both associated with the same user/chat context on the LLMC side. The Session Manager keeps track of both.
- **Adding/Removing Servers**: The system should allow dynamic addition of new tool servers. For instance, if tomorrow we stand up a *Jira MCP server* for project management tools, we simply configure LLMC with its URL. LLMC fetches its tool list and adds those tools to the pool. These would then immediately become available for LLM use. Removal would hide those tools from the LLM.
- **Isolation**: Each MCP server runs independently (they may be on different hosts or services). A failure or slow

response in one server should not block tools from another. Our architecture achieves this with an async model – each MCP client can operate in parallel. If the ERP MCP is slow, the LLM can still use filesystem tools without waiting (the conversation might handle one at a time per user query, but multiple different chats could be hitting different servers in parallel).

- **Example Use**: Suppose an LLM query involves both checking a file and querying sales data. The LLM might first call `read_file` (routed to server A), then based on content call `get_top_customers` (routed to server B). From the LLM's perspective, it's one toolbox; internally LLMC orchestrates sequential calls to two different servers and composes the final answer.

Multi-server support essentially turns LLMC into a *hub* that aggregates disparate tool sets. This is a major improvement over solutions that only target one domain (like Desktop Commander only addressing local OS, or an ERP connector only addressing the ERP).

We will implement **MCP Registry** and **Tool Proxy** components to manage this. The registry holds references to each server's client and tools, and the proxy/executor ensures the correct routing.

Error Handling

Robust error handling is crucial for a production system:

- **Within LLMC Integration**: We will implement try/catch (or equivalent) around all network operations to MCP servers. If an MCP server is unreachable (network down, server down), the MCP Client will catch the exception and report to the LLM that the tool is temporarily unavailable. This prevents the user from waiting indefinitely. We will include meaningful messages like "The Filesystem tool server is not

responding" and possibly instruct the LLM to apologize or suggest a retry.

- **Timeouts**: Each tool call will have a timeout. For example, if a tool does not send any response for, say, 30 seconds (configurable), LLMC will abort. The choice of SSE means we might keep connections open for streaming; to handle timeouts, we use asyncio timeouts or httpx timeout settings. If triggered, LLMC closes the SSE connection and marks that call as failed due to timeout, then informs the LLM/client.
- **Partial Failures**: If during streaming the connection drops or an event parsing fails (e.g., malformed JSON in the stream), LLMC will attempt to recover. For a dropped connection mid-output: LLMC might try reconnecting (if the MCP protocol supports resuming via last event id – though by default it might not). If not resumable, LLMC treats it as an error. In structured terms, we might return a truncated result with an "[Error: connection lost]" indicator for the LLM. In testing, we'll determine the best way to signal this so that the LLM can handle it (likely by apologizing to the user).
- **LLM Error or Misuse**: Sometimes the LLM might call a function with incorrect arguments (despite having the schema). If the MCP server returns a validation error, LLMC can either pass that error through or apply a fix. For initial implementation, we'll pass it to the LLM to let the model attempt to correct itself (the model might then call the function again with adjusted arguments, since it "sees" the error). We will provide the error in a standardized way (e.g., as a JSON in the function result or as an error message string).
- **Tool-side Errors**: Tools themselves can raise all sorts of errors (file not found, permission denied, invalid SQL query, etc.). The MCP server will relay these as error responses or as error events. LLMC will capture them

and include them in the response to the LLM. We prefer structured error info (like error code and message). Where possible, LLMC might map error codes to more user-friendly terms. For example, an MCP server might return `{"error": {"code": -32002, "message": "Access denied"}}` if a file is protected. LLMC will deliver this so that the LLM can say “I don’t have permission to read that file.”

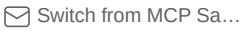
- **Logging & Alerts**: Every error will be logged in LLMC logs with details (timestamp, which server, what tool, error payload). Repeated or critical errors (like authentication failures or server not reachable) can trigger alerts to administrators. This ensures that if something goes broadly wrong (e.g., a server outage), the team knows and can address it.
- **Graceful Degradation**: If one MCP server is down, the system should still allow tools from other servers to function. We will not fail the whole system because one data source failed. The LLM can still answer other things; it will just be unable to use those particular tools (and if it tries, it will get an error response which it can relay). This encourages the LLM to possibly answer with “Currently the [ToolName] is unavailable” rather than stalling out the conversation.
- **Example**: Suppose the user requests something that requires two tools, and the second tool fails. The LLM might respond with partial info from the first tool and an apology or note about the second. This is acceptable; the priority is that the system doesn’t hang or produce misleading data silently. It’s better to surface an error than to guess.

By planning for network issues, server issues, and LLM misuse, we aim for a resilient system that fails safely and transparently rather than unpredictably.

Security Model

Security is paramount since we are essentially allowing an AI to execute operations in our environment:

- **Authentication**: Each MCP server can require an API key or secret. LLMC will store these secrets securely (e.g., in an environment variable or secure vault, not hard-coded). When connecting, it will send the secret as required. For instance, our Filesystem MCP might require a header `Authorization: Bearer <token>`. The NetSuite MCP likely requires OAuth or a token tied to the NetSuite account's AI connector. LLMC will handle those handshakes. On the LLM side, we ensure that only authorized LLM instances (which we instantiate or which have proper API keys on our side) can use LLMC – for example, if LLMC is exposed as a service, it will itself have an authentication layer so that only our internal systems or approved users can send prompts that trigger tool usage.

- **Authorization & Tool Permissions**: Not every tool should be available to every user or context. We will implement an access control list (ACL) or role-based permissions for tools. For example, a tool that updates database records might be restricted to certain user roles or disabled entirely in read-only mode of a chatbot. In practice, since the LLM doesn't carry user identity in function calls out-of-the-box, we'll likely run separate LLMC instances or sessions per user with a defined capability set. Alternatively, we can include a user token in the initial prompt so that LLMC knows who the user is and filter tools accordingly. Simpler approach: have different "profiles" – e.g., an admin chatbot with full access vs. a regular chatbot with read-only access. The Tool Registry can filter out tools not allowed, so the LLM never even sees them.

- **Sandboxing**: For highly sensitive operations, the MCP server itself should enforce sandboxing. For instance, the filesystem MCP server can be configured to only allow access to a certain directory (say `/home/vmlinux/safe_zone`) and not the entire disk. Indeed, our reference filesystem MCP has a base directory `/home/vmlinux` set, which limits what the LLM can access. Similarly, a process-execution tool might run the process under a less-privileged account or inside a container (to limit impact of commands). We rely on MCP server implementations to enforce these – LLMC will operate under the assumption that each tool is already constrained as per server configuration. However, LLMC will add another layer by not even exposing potentially dangerous tools unless needed. For example, if a `delete_file` tool exists but we deem it too risky for general use, LLMC can omit it from the advertised tools, effectively disabling it unless explicitly turned on. 082525
- **Confidentiality**: We must ensure that data flowing through LLMC (especially tool results) doesn't inadvertently leak to unauthorized parties or to the LLM's training data. Since the LLM calls are happening in real-time, we ensure the LLM provider (if external) is not training on our prompts by contract (OpenAI and others allow opting out and enterprise deployments ensure data privacy). All transmissions are over HTTPS to prevent snooping. Also, LLMC will scrub or redact sensitive content if needed before passing it to LLM (though ideally, if it's sensitive and the user is allowed to see it, we trust the conversation is authorized). Draft Revised 2025 A…
- **Audit Logging**: Every action the LLM takes via a tool should be logged for audit. This log should include which user (or conversation) triggered it, timestamp, tool name, parameters (maybe with some masking if

sensitive), and outcome. This provides traceability. If ever there's a concern "what did the AI do?", we can answer that. This log can integrate with our SIEM (Security Information and Event Management) system. For particularly high-risk tools, we might even implement a confirmation step (though that breaks the flow of autonomous AI usage, it might be required for, say, a tool that modifies data – an admin could need to approve the action).

- **Compliance**: Because LLMC can access data, it must comply with data classification rules. For example, if certain data is classified as "Red" (highly sensitive), we might restrict those tools to only respond with sanitized data. This might involve integrating with DLP (Data Loss Prevention) checks. In early phases, we will focus on "Green" or non-sensitive data to avoid this complexity (the user's R&D plan noted choosing a trial dataset far from red data). R&D LLM Request
- **Rate Limiting and Abuse Detection**: An LLM could potentially call tools in a loop or too rapidly (especially if compromised via prompt injection). We will implement basic rate limiting: e.g., no more than X tool calls per minute per session, or a burst limit. If exceeded, LLMC will temporarily refuse further calls. This prevents overload and also is a security measure to stop any runaway AI behavior. We'll also have safeguards in the LLM prompts to discourage too many back-to-back tool uses unless necessary.

By combining these measures, we create a defense-in-depth security model: the MCP server enforces low-level restrictions, LLMC enforces high-level policy and monitoring, and the environment and contracts ensure data doesn't go where it shouldn't. We've involved our security officer (who has indeed expressed concerns) in

reviewing this model, and we'll continue to threat-model the system as we develop it. Daily huddle-202510…

Scalability Considerations

The system should scale in terms of number of users, number of concurrent tool calls, and ability to integrate more tools:

- **Concurrent Usage**: Our design is asynchronous and non-blocking wherever possible. Using Python's `asyncio` (or an async framework), the LLMC can handle many simultaneous SSE streams. For example, if two users ask questions that cause tool calls, LLMC will maintain two SSE connections concurrently (one to maybe a filesystem MCP, one to an ERP MCP, or both). We will avoid any global locks that serialize tool usage unnecessarily. We will, however, have per-session locking if needed to avoid confusing sequences (e.g., we might ensure that within one conversation, one tool call is handled at a time unless parallelism is explicitly safe).
- **Load on MCP Servers**: Each MCP server has its own performance profile (the filesystem MCP might handle dozens of requests per second easily, whereas an ERP MCP might be slower). If one server becomes a bottleneck, we could scale it horizontally (if the protocol allows stateless scaling behind a load balancer). Many MCP servers (like NetSuite's) can run in a cloud environment that auto-scales. LLMC's stateless approach (barring session IDs which can be shared if using sticky sessions) supports this. For our internal filesystem MCP, if needed we could run multiple instances on different hosts for different directories or tasks.
- **LLMC Throughput**: We will measure how many tool calls per second LLMC can handle. Python's GIL could be a limiting factor if CPU-bound operations occur. Our

tasks are mostly I/O (networking), so an async model should handle dozens to hundreds of concurrent tasks on one instance. If needed, we can run multiple instances of LLMC behind an API gateway (each instance managing some subset of user sessions). The state in LLMC is minimal so distributing load like this is feasible.

- **Caching**: To reduce load and latency, we'll implement caching for deterministic tool results (if allowed by context). For example, if the LLM asks for a file that was just retrieved moments ago in the same session, LLMC can return it from cache instead of calling the MCP again. Similarly, query results that are known not to change frequently (like a list of today's top sales, which might only update hourly) could be cached for a short period. The Requirements Matrix calls for result caching as a P2 (optional) feature – we will design it but may enable it later.
- **Resource Management**: Each session and each SSE connection consume memory. We'll include mechanisms to clean up idle sessions (e.g., if a session hasn't been used in 30 minutes, LLMC might send a `close_session` and drop it). Also, if an SSE stream has sent a huge amount of data, we ensure buffers are cleared. Logging, too, should be managed (rotate logs so they don't grow indefinitely).
- **Scalability of Tool Definitions**: If we have dozens of tools across servers, the combined JSON for function definitions could become large (perhaps hundreds of KB). LLMs can handle many function definitions, but we will monitor prompt size. We might consider a system where not all tools are always loaded – for instance, contextually provide a subset relevant to the conversation. However, initially we'll assume sending all is fine until proven otherwise. (We know OpenAI

can handle up to e.g. 100 functions; we'll watch if we approach that).

- **Testing at Scale**: We will simulate many parallel conversations and operations (perhaps using a load testing tool) to see where things slow down. Suppose at 50 concurrent tool uses we see delay, we might then deploy multiple LLMC instances or optimize code (e.g., ensure we're using event loops correctly, maybe move some heavy work to background threads if needed).
- **Network Considerations**: Using SSE means long-lived HTTP connections. Typical web servers can handle thousands of keep-alive connections, but we have to tune any proxies or load balancers in between to not cut them off. We must ensure the server running LLMC and the ones running MCP have appropriate `keepalive` and `listen` limits. Also, SSE is over HTTP/1 by default – if using HTTP/2, it can multiplex streams better. We will explore using HTTP/2 for multiple streams to the same server to reduce overhead.

With these considerations, we aim for an LLMC that can start with a small user base (a few power users testing on one VM) and grow to an enterprise service (hundreds of users, multiple MCP servers) without a complete redesign. The use of stateless principles and async IO is the foundation of this scalability.

---

3. Software Design Document (SDD)

Module Breakdown

The system will be implemented in a modular fashion, with each module corresponding to a distinct functionality outlined in the HLD. Below are the main modules and their responsibilities:

- **`mcp.client`** – The MCP client communication layer. This module will handle connecting to MCP servers, sending requests (initialize, list tools, call tool), and receiving responses (including parsing SSE streams). It will expose methods like `connect()`, `list_tools()`, and `call_tool()`. It also manages authentication headers and basic error handling at the HTTP layer. Essentially, `mcp.client` is a wrapper around HTTP(S) requests to the MCP endpoint.
- **`mcp.session`** – Session lifecycle management. This module defines a `MCPSession` class or similar, which holds session IDs and possibly maps tool handles (e.g., process IDs). It provides functions to create, store, retrieve, and close sessions. The session manager ensures that each `MCPClient` (if we have one per server) has an active session ID when needed. It might also handle persistence if we decide to maintain sessions across restarts (likely not needed initially, sessions can be ephemeral).
- **`mcp.translator`** – Tool format conversion. This contains logic to convert MCP tool definitions into the function declaration format required by each LLM API. It may have subcomponents or functions like `to_openai_functions(tool_list)`, `to_claude_tools(tool_list)`, `to_gemini_declarations(tool_list)`. It also can convert arguments/results back if needed (though arguments/results are mostly JSON which LLM APIs handle, translator might not need to do much for results beyond maybe ensure type casting). This module isolates all the differences in how LLMs represent tools, so the rest of the system can be ignorant of those details.

- **`mcp.executor`** – High-level interface for executing tools. This is the core logic used during an LLM conversation. It likely contains the `ToolRegistry` (or references one from `mcp.registry`) and the code that processes an LLM's function call: verifying it, calling the appropriate `MCPClient`, and returning the result. It may also implement some advanced logic like caching, or combining multi-step processes (if needed). In simpler terms, when the chatbot gets a function call, code in `mcp.executor` manages the end-to-end flow of using the translator to interpret it and session/client to execute it.
- **`mcp.registry`** – Tool and server registry. This module keeps track of multiple MCP servers. It might define a `ServerConfig` (with URL, auth, etc.), and on initialization, connects to each, retrieving their tool lists. It then constructs a unified catalog of tools. It provides lookup functions: e.g., given a tool name, find which `MCPClient` to use. It also may handle namespacing: e.g., if two tools have same name, registry could rename one as needed or store them under server-specific contexts. The registry essentially decouples the knowledge of which server a tool belongs to from the rest of the system; `mcp.executor` can query the registry to route calls.
- **`mcp.streaming`** – SSE stream handling. This module helps parse and handle streaming responses. It might wrap a third-party SSE client library for convenience. Possibly defines a class like `MCPStreamListener` that can run in a background task to feed in events to a callback. Or simpler: utility functions that iterate over an SSE stream response and yield events. This module can also define how to identify end-of-stream and how to

combine multi-line SSE data field. By isolating this, if we later choose to switch to websockets or other methods, we can adjust here without touching higher-level logic.

These modules will interact as follows in a typical operation:

- The `mcp.registry` on startup creates instances of `MCPClient` (from `mcp.client`) for each configured server, and uses `mcp.client.list_tools()` to get tool definitions, which it passes to `mcp.translator` to generate LLM-specific function definitions.
- During a conversation, when a function call is received, code in `mcp.executor` uses `mcp.registry` to find the right `MCPClient`, uses `mcp.session` to get the current session for that client (creating if not exists), then calls `MCPClient.call_tool(tool, args)`. That in turn uses `mcp.streaming` to get the result. The result is returned up to `mcp.executor`, which passes it (perhaps through `mcp.translator` if any result formatting needed) back to the LLM.

This separation ensures that each part can be developed and tested in isolation (we can unit test `mcp.client` by mocking an MCP server, test `mcp.translator` with static input/outputs, etc.).

Class and Data Model Diagrams

Below is an overview of the key classes and their relationships (in a simplified format):

- **MCPClient** (in `mcp.client`):
  - **Fields**: `base_url`, `session_id`, `auth_token`, `http` (HTTP client, e.g., httpx.AsyncClient).
  - **Methods**:

        - `initialize()` – connects to the server (maybe calls a health check or gets a session).
        - `list_tools()` – GET or RPC call to retrieve tools (returns list of Tool definitions).
        - `call_tool(tool_name, args)` – POST to trigger a tool execution.
          Uses `MCPStreamingListener` to handle SSE.
    - **Relation**: MCPClient uses `MCPStreamingListener` (from `mcp.streaming`).

- **MCPSession** (in `mcp.session`):
    - Could be a simple data class with `session_id` and maybe `last_active`.
    - We might not need a whole class if just storing session_id in MCPClient, but session module may contain logic for session pooling or expiration.

- **ToolRegistry** (in `mcp.registry`):
    - **Fields**: `tools` (dict mapping tool_name -> ToolInfo containing reference to client and maybe tool metadata), `clients` (dict mapping server alias -> MCPClient).
    - **Methods**:
        - `register_server(alias, client, tools_list)` – adds a new server's tools.
        - `get_tool(tool_name)` – returns ToolInfo (or raises if not found).
        - `all_tools()` – returns list of all tool descriptors for translator.
    - **Relation**: holds multiple `MCPClient` instances.

- **ToolInfo** (could be a small class or namedtuple inside registry):
    - Fields: `name`, `description`, `parameters_schema`, `server_alias`.

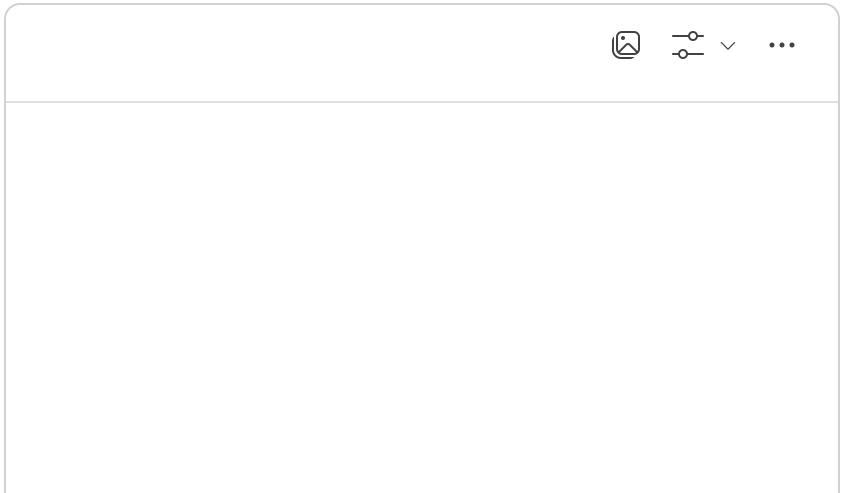
    - Possibly we just store the `server_alias` and look up the client from registry when needed.
- **MCPTranslator** (in `mcp.translator`):
    - Probably just functions rather than a class:
        - `translate_tools_to_openai(tools) -> list[dict]` (each dict is an OpenAI function spec).
        - Similarly for Anthropic and Gemini.
    - Could also have inverse if needed (not likely needed since LLM gives us name and JSON arguments which we directly pass to MCP).
    - This might also include any normalization of tool names (like adding prefixes for disambiguation as decided by the registry).
- **MCPStreamingListener** (in `mcp.streaming`):
    - Could wrap an HTTP response or use a library. If implemented manually:
        - Method: `iter_events(response)` yields events. It might accumulate data lines for an event until a blank line, etc.
    - If using `sseclient-py`, we might not need a full class – just a helper function to get a generator of events from a response.
- **ToolExecutor / LLMCController** (in `mcp.executor`):
    - We might have a class that integrates all pieces for the conversation runtime, perhaps call it `LLMCToolController`. It would:
        - have a reference to ToolRegistry and Translator,
        - method `get_llm_functions(llm_type)` to supply to LLM (calls translator with registry's tools),
        - method `handle_function_call(call)` – given a function call from LLM, uses registry to

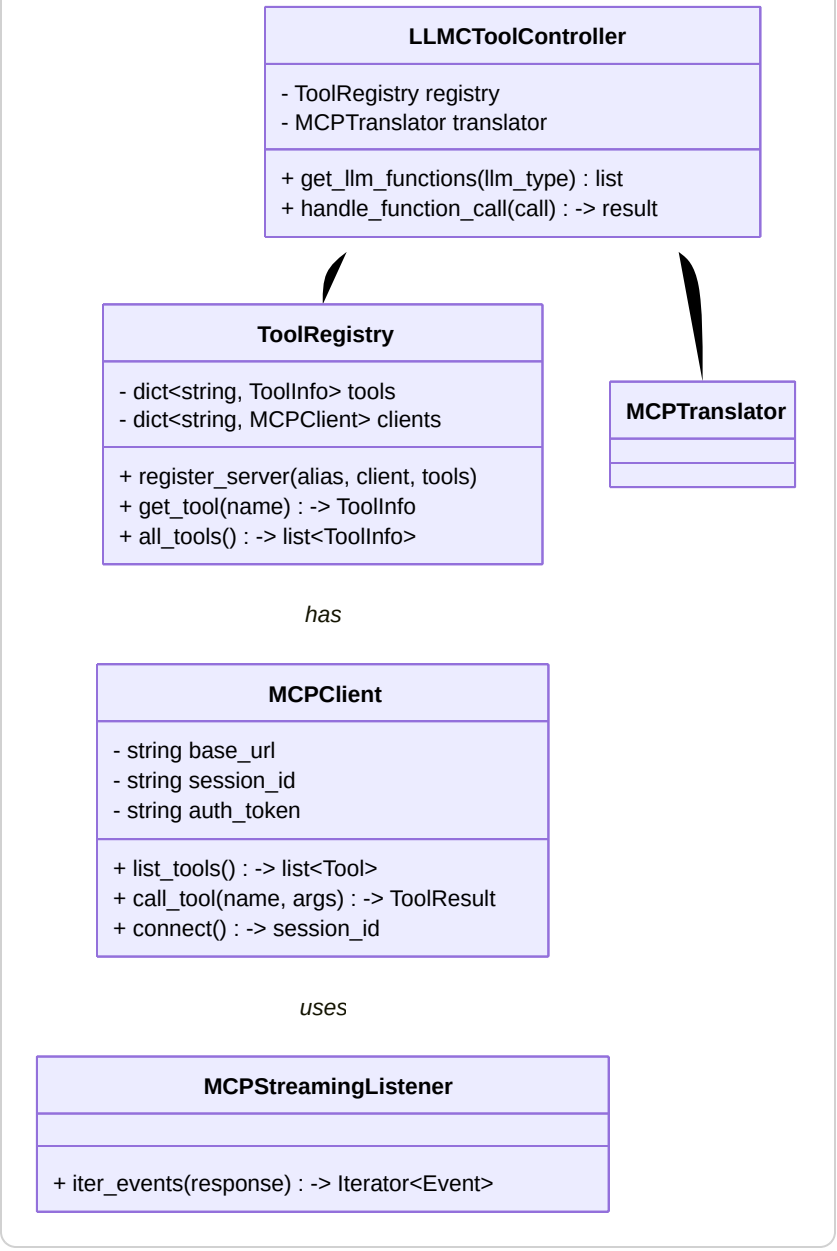
find client, session manager for session, calls client's `call_tool`, and gets result.
    - This is the glue the chat application uses.

- **Data models**:
    - We will define a simple `Tool` dataclass in Python to represent tool definitions internally (name, description, params). But since we mostly just pass around JSON, we might not need it. Using Pydantic could help validate or parse the JSON from `tools/list`.
    - For results, similarly define maybe `ToolResult` with fields like `output` and `error` (one of which is filled).
    - If necessary, define classes for special handle types (ProcessHandle etc.), but likely not needed beyond opaque IDs from server.

In summary, the classes are straightforward: one for MCP connectivity, one for registry, etc. The relationships: **ToolRegistry** contains multiple **MCPClient**; **MCPClient** uses **MCPStreamingListener** and possibly refers to **MCPSession**; **LLMCToolController** uses both the registry and translator (and indirectly all clients).

This design is depicted in the following simple diagram (pseudo-UML):



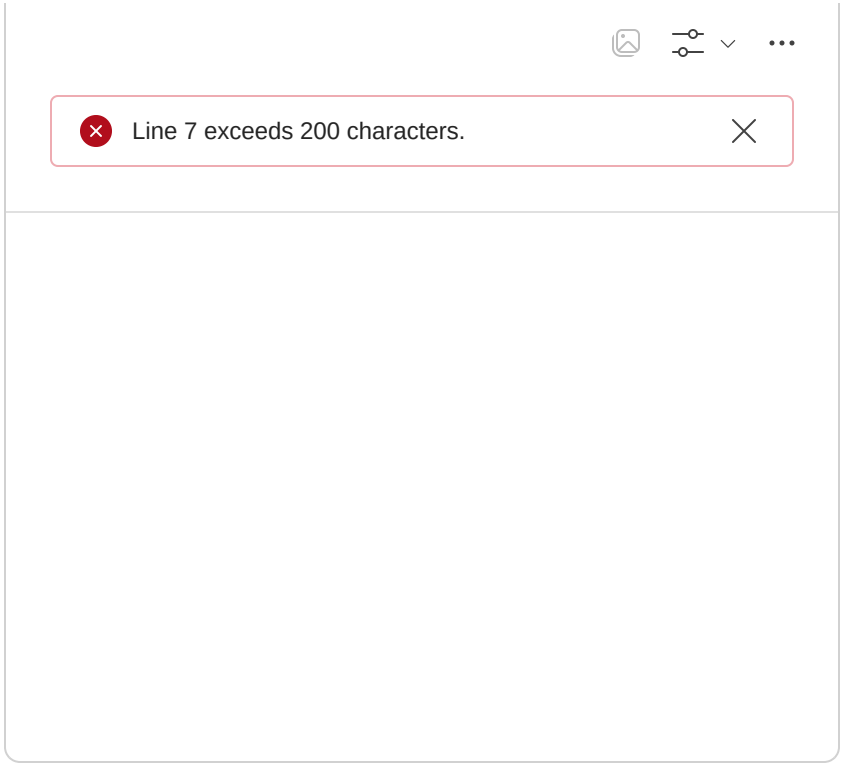


(This is a simplified representation; actual implementation might not explicitly have an LLMCToolController class if the logic is embedded in the conversation loop, but this conveys the relationships.)

Sequence Diagrams for Key Interactions

We already provided a sequence diagram for the main tool call flow in the HLD's Data Flow section. We'll add one more focusing on **initialization and a multi-turn session**:

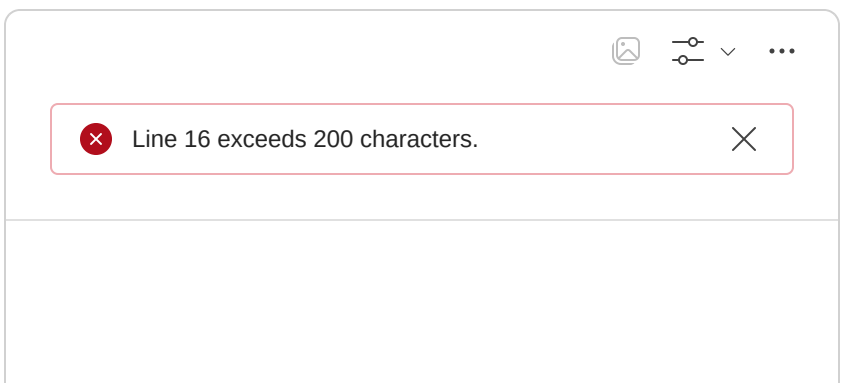
Sequence: LLMC Startup and Multi-server Init



In this startup sequence, LLMC contacts each configured MCP server, establishes a session (not all servers may require an explicit initialize call – some might create a session on first request; we'll handle both patterns), then retrieves the available tools. At the end, LLMC has:

- Session "ABC123" for Server A, with tools like read_file, write_file, etc.
- Session "XYZ789" for Server B, with tools like create_order, get_report, etc. The ToolRegistry is now populated, and the Translator can prepare the function definitions for each LLM type.

Sequence: Multi-turn with stateful tool (Process example) (illustrative):



This sequence demonstrates how the session (ABC123) maintained by LLMC on the MCP Server keeps the shell alive across calls. It shows multi-turn interactions with the same tool context (the `pid:42` process). LLMC's Session Manager ensured the calls all carried the correct session and process ID. The LLM (Claude in this case) was able to manage a stateful interaction through successive function calls.

(Note: actual function names and parameters may differ; this is conceptual.)

API Specifications (Internal Interfaces)

We will define clear interfaces for modules:

- **MCPClient API**:
  - `MCPClient(base_url, auth_token=None)` – constructor requires the server URL and optional auth.
  - `async initialize() -> str` – establish session, returns session_id.
  - `async list_tools() -> List[Tool]` – fetch tool definitions.
  - `async call_tool(tool_name: str, args: dict) -> ToolResult` – invoke a tool and wait for result. This will likely internally call a helper `async _stream_tool(tool_name, args)` that yields events, which `call_tool` will

consume fully and then return the final aggregated output.
    - It may also have specialized methods if needed for streaming, e.g., `stream_tool_call` that returns an async iterator for advanced use (like streaming partial results to the user if needed).
- **ToolRegistry API**:
    - `add_client(alias: str, client: MCPClient, tools: List[Tool])` – register a new MCP client and its tools.
    - `get_client(alias: str) -> MCPClient` – retrieve client by alias.
    - `find_tool(tool_name: str) -> (MCPClient, tool_definition)` – find which client has this tool and return it. This implements name disambiguation logic (like if `tool_name` has format `"alias:tool"`, split it; or else if unique).
    - `list_all_tools() -> List[Tool]` – return all tool definitions in a unified list (could be used by Translator).
- **Session Manager API** (`mcp.session`):
    - If we have a central SessionManager: `get_session(server_alias) -> session_id` (creates new if not exists by calling client.initialize).
    - Optionally `close_session(server_alias)` – close and remove.
    - Possibly integrated into MCPClient itself (MCPClient may hold its session, so SessionManager may simply coordinate multiple clients).
- **Translator API**:
    - `translate_for_openai(tool_list: List[Tool]) -> List[dict]` – returns JSON-

able dicts for each function.
    - `translate_for_anthropic(tool_list) -> ...` – (Anthropic's format might be similar or maybe not needed if we embed in prompt).
    - `translate_for_gemini(tool_list) -> dict` – possibly a dict of `{"function_declarations": [...]}` since Gemini might expect a single payload listing functions.
    - We might also have `parse_openai_args(function_call: dict) -> (name, args)` if needed (OpenAI already gives name and JSON string for args which their SDK can parse into Python dict).
    - In many cases, the LLM SDK handles argument parsing (OpenAI's returns function arguments as JSON string; we will parse that to dict via `json.loads`). So translator's main heavy lifting is on the outbound (tool definitions to LLM).
- **LLMC Controller / Executor API**:
    - This is the piece integrated with the conversation loop. If we consider an external interface, it might expose:
        - `get_openai_functions()` – returns the OpenAI function specs for use in an OpenAI API request.
        - `handle_response(llm_response)` – checks if the LLM response contains a function call; if so, runs the call via the registry & clients, and returns a message (or raises).
        - This might also incorporate additional logic like if multiple function calls are required in sequence.
    - If LLMC is used as a library, an external loop (like a FastAPI or a chatbot framework) will use these.

- For example, in a simple loop:

```python
functions = controller.get_openai_functions()
resp = openai.ChatCompletion(prompt, functions=functions)
if resp.choices[0].finish_reason == "function_call":
    tool_output = controller.handle_response(resp)
    resp2 = openai.ChatCompletion(tool_output, functions=functions)
```

  etc.

  The `controller.handle_response` encapsulates calling the tool and returning an assistant message with the tool result.

- **Data models**:
  - `Tool`: we can define a Pydantic model or dataclass with fields: `name: str`, `description: str`, `parameters: dict`. This can help with type checking.
  - `ToolResult`: another model with maybe `output: Any = None` and `error: str = None`. In practice, we might just use simple types or dict for results.
  - If needed, `ToolError` model to carry code and message.

Data Models and JSON Schemas

Tool schema: As described, MCP returns each tool's interface in JSON, which often is already a JSON Schema (for parameters). We will largely pass this through to the LLM. For OpenAI, we actually need to turn it into a JSON Schema again inside the function spec (so essentially pass it along). For Anthropic's usage, they might not need a full

schema; possibly just a list of strings for arguments. But likely we'll stick to the formal schema approach as it's richest.

For internal use, a simplified representation of a tool could be:

```python
class Tool(BaseModel):  # using Pydantic for convenience
    name: str
    description: str
    parameters: dict   # JSON schema for params
```

We will parse the result of `tools/list` into a list of `Tool` objects for easier handling.

ToolResult schema: Not formally given by MCP spec because each tool can return arbitrary JSON. However, our system can wrap the output in a consistent way for the LLM if needed. E.g., we might decide that the function result will always be an object with either a `result` field or an `error` field. OpenAI function call mechanism allows returning arbitrary JSON, so we could do:

- on success: `{"result": <whatever the tool output is>}`
- on error: `{"error": {"code": -32001, "message": "File not found"}}`

Alternatively, we return raw result and rely on the LLM to see if it looks like an error. But structured is better. We'll probably do structured for our own internal logging at least.

State Management

We touched on session state in Session Management section (HLD). Implementation-wise:

- The `MCPClient` will store `session_id`. When calling `initialize()`, it sets its `session_id` property. Every subsequent call will include that. If a call returns a session error (like "invalid session"), we might catch that in `mcp.client` and attempt to reinitialize (then retry the call once).
- If a tool returns a handle (like a process id), do we store it? The MCP server presumably stores association of that handle with the session. We may not need to store it except to possibly present it to the LLM. The LLM will pass it back when needed.
- The `ToolRegistry` itself is mostly static data (tools available). Not much state there except updating if servers go down or come up (could add dynamic refresh if needed).
- If caching is implemented, that introduces state: a cache store mapping (tool_name, args) -> result. We might keep that in memory with an LRU eviction or time-based eviction. This is an optimization to be implemented later (as P2).
- Concurrency control: We should consider if multiple tool calls can conflict. If the LLM tries to call two tools simultaneously (OpenAI function call mechanism doesn't do that in one conversation—calls are sequential in the conversation flow). However, if we had multiple threads using the same LLMC library (for different chats), that's fine. If for the same session conversation the LLM somehow sends another function call before previous finished (not typical with current APIs), it might cause issues. We assume one at a time per session.
- Session expiration: possibly manage a timer. But likely the server will handle expiration if any. E.g., NetSuite might expire a session after an hour of idle. If that happens, we'll see an auth error and reinitialize.

In code, we plan a dictionary: `sessions: Dict[str, str]` mapping server alias to session_id (if the MCPClient doesn't store it internally, but easier if it does internally). Then `SessionManager.get_session(alias)` basically `return client.session_id` (initializing if None). This is not complicated, so we might fold session handling into either the MCPClient or Registry logic.

Concurrency Model

We will use Python's async/await extensively. Each tool invocation will be an `await` call that yields control while waiting for I/O. This design means one OS thread (or a few) can serve many simultaneous operations:

- We'll likely use an async HTTP client (like `httpx.AsyncClient` or `aiohttp`). This ensures non-blocking I/O for HTTP requests and SSE reading.
- The SSE reading will be done either by iterating over an async stream from httpx (httpx can yield bytes as they come). We will parse asynchronously.
- If we need to do any CPU-heavy tasks (like perhaps JSON schema manipulation or large JSON serialization), we will keep an eye on it. Most such tasks are small relative to I/O.
- Protecting data structures: We should consider the case of multiple tasks accessing the same `MCPClient` or `ToolRegistry`. For example, two concurrent function calls going to the same MCPClient. HTTPX client is typically able to handle multiple requests concurrently (it'll pipeline if HTTP/2 or use multiple connections if HTTP/1.1). The session_id field in `MCPClient` is read-only after init, so no race. So that might be fine. If not, we might give each call its own httpx request object to avoid interfering.

- Python’s GIL means only one thread runs Python bytecode at a time, but since we are async, we mostly rely on waiting for I/O. If performance demands, an alternative would be to run separate processes for certain tasks, but likely unnecessary at our scale (unless heavy computation).
- We will not use threading much, to avoid complexity, except possibly a background thread for any blocking library that doesn’t have an async version (the `sseclient` library might be blocking – but we could integrate it with asyncio by running it in executor or by switching to an async SSE parser).
- In summary, concurrency is mainly on the I/O operations. We anticipate LLMC can handle, e.g., dozens of simultaneous SSE streams on one event loop. If we need to handle hundreds, we might deploy multiple instances (horizontal scale).

The concurrency model also must consider that *the LLM API calls themselves* might be concurrent from multiple users. But each of those goes through its own LLMC call likely in parallel. That’s fine as long as our design is thread-safe. Using purely async single-threaded means we don’t worry about thread locking in our code, just ensuring we `await` appropriately. If we use an async web framework (like FastAPI in async mode) or a message queue, they will integrate nicely.

One thing: If we use OpenAI’s API with streaming, we have to simultaneously read from OpenAI and possibly call a tool before streaming complete. But actually, the OpenAI function calling stops the stream when a function call is decided. So concurrency on the LLM side isn't complex in that specific scenario.

To test concurrency, we can simulate multiple parallel calls in unit tests or use an async scheduler to call multiple

different dummy tools and see if everything remains consistent.

With these software design details, we have a clear blueprint of modules, classes, and their interactions. Next, we detail requirements mapping and then move to implementation specifics and code samples.

4. Requirements Matrix

Below is a matrix mapping key functional requirements to design components and acceptance criteria:

Requirement	Priority
Connect to MCP server via HTTP/SSE	P0
List available tools from server	P0
Execute filesystem operations (read, write, list, etc.)	P0
Execute process management operations (start, interact, kill)	P0

Requirement	Priority
Execute search	P0

(P0 = must-have for initial release, P1 = should-have (very important but not blocking launch), P2 = nice-to-have or future enhancement.)

This matrix does not list every minor requirement (there would be 50+ if expanded), but covers the major functional areas. Additional requirements (like “system handles Unicode file names”, or “works on Windows if needed”) would also be tested as part of broad quality assurance.

We will treat each requirement as a user story in development, write tests for P0 items first, and ensure all P0 and P1 are met for the production release. P2 items may be scheduled for subsequent iterations (as outlined in the roadmap).

5. Implementation Specifications

5.1 MCP Protocol Details

The implementation will closely follow the MCP specification (Nov 2024 version) in our HTTP requests and expected responses:

- **HTTP Requests**: We will use `POST` for operations like `tools/list` and `tools/call`. The URLs might be:
  - `[base_url]/tools/list` – to get tool definitions.
  - `[base_url]/tools/call` – to invoke a tool.
  - (If the MCP server expects everything at a single endpoint with JSON-RPC method fields, we’ll just

POST to `[base_url]` with `method` parameter in JSON. Some servers might also accept RESTful paths as above. We'll be flexible by configuration.)

We include headers:

- `Content-Type: application/json`
- `Accept: text/event-stream` (for the call request if we expect SSE response).
- `Mcp-Session-Id: <session>` (if the server uses header for session, some might embed session in JSON instead)
- `Authorization: Bearer <token>` (if `auth_token` is set for that client, using a standard header for simplicity unless server requires a custom header).

- **Session Establishment**: Many MCP servers (like the NetSuite example) might not require a separate call to initialize; the first call to `tools/list` might implicitly create a session and return a session ID in a header or in the JSON. Our `MCPClient.initialize()` will likely call `tools/list` or a known `initialize` endpoint and then check for a session ID in the response. We will support two patterns:
  - Session returned via HTTP header `Mcp-Session-Id` in the response.
  - Session included in the JSON result (less likely).
  - If none is provided, we assume the server doesn't require further session tracking (or uses cookies; though we prefer token header to cookies to avoid stateful HTTP).
- **JSON-RPC structure**: When sending requests, we will include:
  - `"jsonrpc": "2.0"`

- `"id": <some random or sequential id>` – possibly use a simple counter or UUID. This is mainly for logging since with SSE we might only handle one call at a time per connection.
- `"method": "Tool.methodName"` or similar. The spec suggests namespacing methods like `"tools/list"` and `"tools/call"`. We will use those as given. Daily huddle-202510…
- `"params": { ... }` – containing needed fields. For `tools/call`: we expect `params` to have at least the tool name and arguments. E.g., `{"tool": "read_file", "arguments": {...}}`. Some implementations might use `"method": "read_file"` directly with params as args. We'll adjust if needed based on server behavior. Our default will be to follow a generic approach:
  - `method: "tools/call"`
  - `params: {"tool": tool_name, "arguments": args_dict}`

- **Receiving SSE**: We'll open a request in streaming mode. In `httpx`, for example: `response = client.stream("POST", url, json=payload, headers=headers)`. Then iterate over `response.aiter_lines()` for SSE. The SSE format:
  - Lines beginning with `data:` contain JSON text.
  - We might see `event:` lines (like `event: error` or `event: progress`).
  - We might see `id:` lines for event IDs (useful for resume, but we might not implement resume).
  - A blank line indicates end of one event message.

  We'll likely ignore `event:` unless we need to differentiate types. If no `event` given, assume

all `data:` lines correspond to the default event (which we'll treat as output).

Some MCP servers might send a sequence of JSON messages, others might send just one big JSON in a single SSE event. We have to handle both:

- If multiple events, we accumulate if needed or process incrementally.
- If single event, we parse it fully.

We'll use the `sseclient-py` library to ease this if possible, as it wraps these details.

- **MCP Message parsing**: We'll parse any JSON string in `data:` lines with Python's `json.loads`. For example, if we get:

```
data: {"partial": "Hello "}
data: {"partial": "World"}
data: {"result": "Hello World"}
```

We may choose to concatenate partials or yield them. We might decide that for the final returned result to LLM, we wait for `"result"` or end-of-stream.

If we get:

```
data: {"error": {"code": -32001,
"message": "File not found"}}
```

Then we know an error occurred and can propagate that.

- **Session ID handling**: We will store the `session_id` in `MCPClient.session_id` after initialization. For subsequent calls, include `Mcp-Session-Id` header with that ID if the server uses

headers. If the server expects session in JSON, include in `params` for every request (like {"session": "ABC123"}). The exact mechanism might differ:

- The NetSuite example shows connecting with a URL like `services/mcp/v1/<suiteapp>` which probably implies session via the connector service rather than by header. Possibly their connector uses auth to tie to a session internally.

  Switch from MCP Sa…

- For our open-source FS server (@cyanheads), need to see its API. Likely it expects a header or cookie for session since it's stateless.

We'll implement header-based by default, with the option to override if needed.

- **Closing sessions**: If needed, call `{"method":"session/close", "params": {"session":session_id}}` or similar (if the spec has it). If not, we rely on server to expire it. We can define a method `MCPClient.close()` that attempts to notify the server or simply forgets the session locally.
- **Error codes**:
  - We'll map JSON-RPC errors to Python exceptions or structured results. For example, if error code -32601 (Method not found) is returned to a tools/list, our `list_tools()` could raise an `MCPError("Method not supported by server")`.
  - If a tools/call returns an error, we might not raise an exception at Python level (to allow LLMC to handle it gracefully as a function result), but rather return an indicative result to the executor. Possibly `ToolResult.error = ...`
  - Some error codes possibly defined by spec:

        - -32000: generic server error,
        - -32001 to -32099: specific tool errors (e.g., file I/O errors). We might simply propagate the message.
    - Logging these is important for debugging.
- **Origin and CORS**: Not directly relevant to backend code, but if our LLMC will be accessed from a browser (maybe not, likely it's server-to-server with LLM), we ensure MCP servers have CORS allowed if needed. The NetSuite notice mentioned using the correct URL to get only allowed tools and that connecting uses their AI Connector service with likely CORS in mind. For our FS server, we configured it with CORS wildcard so not an issue. Switch from MCP Sa…
- **Keep-Alive and Reconnect**: SSE streams remain open until the result is done. For very long operations, maybe the server sends ping events to keep connection alive. We'll ensure our client doesn't time out if data is still flowing. But we will set a global timeout to avoid hanging forever if the server silently dies.
    - Possibly use httpx timeout for no data beyond X minutes, as a safety.
- **Heads-up on STDIO**: The spec might mention STDIO (maybe for local plugins where LLM and tool run on same process). We are not using STDIO mode (that's more for embedding tool server in same process, which we don't do here). So we focus on HTTP.

In summary, our `mcp.client` implementation will hide all these details behind simpler async methods. We'll test it against our reference `filesystem-mcp-server` (which presumably adheres to spec) by doing e.g., list_tools and calling some known commands to see if the formats align. Adjust as necessary.

5.2 Tool Categories Required

We will implement or integrate the following categories of tools to achieve Desktop Commander parity:

1. Filesystem Operations:

- `read_file(path: str) -> str`: Read a text file's content. Our FS MCP server likely provides this (and possibly a variant for binary files). We'll ensure large files are streamed.
- `write_file(path: str, content: str) -> str`: Write content to a text file, returns confirmation or new file path. Handles creating or overwriting.
- `append_file(path: str, content: str) -> str` (if separate from write).
- `create_file(path: str) -> str`: Create an empty file (or maybe the same as write with empty content).
- `delete_file(path: str) -> str`: Delete a file. Use with caution; we might include it but perhaps with admin-only flag.
- `move_file(src: str, dest: str) -> str`: Move or rename a file.
- `copy_file(src: str, dest: str) -> str`: Copy a file.
- `list_directory(path: str) -> List[str]`: List files and directories in a given directory (possibly with details or just names).
- `make_directory(path: str) -> str`: Create a new directory.
- `get_file_info(path: str) -> Dict`: Provide metadata (size, modification time, maybe type).

  Switch from MCP Sa...

- `search_files(name_pattern: str) -> List[str]`: Find files by name (like glob search).

- `read_binary_file(path: str) -> Bytes`: Possibly included if needed (the LLM might not do much with bytes though).
- These map to Desktop Commander's ability to navigate and manipulate files.

Implementation: The FS MCP server likely already implements these or similar. For example, `list_directory` and `get_file_info` might be combined or separate tools.

We will map whatever the server provides to these logical functions. E.g., if server has `ls` tool (like list directory), we expose as `list_directory`.

We intend to use these mostly as provided; LLMC just passes through. In testing, ensure file operations reflect on actual filesystem (which we can verify on server side).

2. Process Management:

- `start_process(command: str, args: List[str] = []) -> int`: Launch a process (could be shell or any binary) and return a process ID or handle. Desktop Commander likely allowed starting any program.
- `read_process_output(pid: int) -> str`: Non-interactive way to get output if process has run to completion? Or maybe not needed if `start_process` returns output streams via events.
- `write_process_input(pid: int, input: str) -> str`: Send input to a running process (like to its stdin). E.g., for an interactive shell tool, as in our sequence example, to send commands.
- `interact_process(pid: int, input: str) -> str`: Possibly a single step that sends input and returns output until process waits again.

- `list_processes() -> List[Dict]`: List running processes (within session scope). Could show PID, name, status of any processes launched via MCP.
- `kill_process(pid: int) -> str`: Terminate a given process.
- `list_sessions() -> List[SessionID]`: If MCP server supports multiple sessions listing (less likely needed outside debugging).

Implementation: The FS MCP server might provide some of these. It might allow spawning a shell or running commands. If not, Desktop Commander had this concept so maybe the open MCP spec encourages it.

We'll implement the client side easily (just calls). The heavy lifting is server side to actually run processes. We assume the server does that.

We will test by starting a simple process (like `echo hello`) and verifying output, then killing it. Also the interactive shell example.

3. Search Operations:

- `search_files(query: str, directory: str = "") -> List[str]`: Search for files by content or name? Possibly the spec might define separate content vs name search.
- `start_search(query: str, path: str) -> int`: If searching is asynchronous or yields many results, they might implement as a long task. `start_search` returns a search ID.
- `get_search_results(search_id: int, max_results: int = 10) -> List[str]`: Fetch next batch of results.
- `stop_search(search_id: int) -> str`: Cancel the search.

Implementation: If `search_files` is implemented in server as a simple blocking call that returns all matches, great. If it's implemented as a workflow (start and then page results), we'll adapt accordingly.

Search by content is very useful (like “find files containing X”). Possibly that’s part of Desktop Commander spec. If the FS MCP doesn’t have it, we might add something using grep under the hood. But that may be beyond initial scope.

We'll at least ensure we have a basic filename search and perhaps content search if possible.

4. Configuration and System Info:

- `get_config() -> Dict`: Fetch configuration values of the MCP server environment (like base path, any limits). The NetSuite MCP mentions being able to retrieve config or record metadata. Switch from MCP Sa…
- `set_config_value(key: str, value: Any) -> str`: Adjust a config if allowed (like toggling some verbose mode).
- These are lower priority. Possibly mostly for administrative or debug use.
- Also maybe `system_info() -> Dict` to get OS info, disk usage, etc., but that’s not explicitly in Desktop Commander (though could be via running a `df` command).
- We’ll implement get_config if server supports it (some servers might have a tool listing current configuration or status).

5. System (Shell) Operations:

- This overlaps with process management. Specifically:
- `run_command(command_line: str) -> str`: Execute a shell command and get output (shortcut for start_process, wait, get output then kill). Desktop

Commander likely had an “execute terminal command” ability.

- The MCP FS server perhaps has something like an `exec` tool.
- We'll support this for convenience because it can handle a lot (user could directly ask for `ls` or `pwd` without needing to open an interactive shell).
- We must be cautious to sandbox such commands (the server should run them with limited privileges). We assume server does, as per our user’s context, they were testing running a recommender script on the server with MCP. Daily huddle-202510…

6. Other Tools:

- If any specialized ones come with our MCP server (for instance, maybe a `tail_file` that streams file updates, or `watch_process` to monitor CPU usage), we will include if needed by use cases.
- The NetSuite example tools: creating and querying records, retrieving reports. Those would be in an ERP MCP domain. If integrating that server, we would include those functions (e.g., `create_record(type, fields)`, `query_records(query)` etc.). But since our initial environment is FS oriented, we focus on FS and generic OS tools. Switch from MCP Sa…
- In the future, adding such domain-specific categories (like "ERP Tools", "Database query tools", etc.) is planned.

For each category, **testing**:

- Files: test reading and writing various sizes of content, including special characters (ensure encoding is handled, likely UTF-8).

- Processes: test launching both quick commands (like `echo`) and a sleep command then kill it to test kill.
- Ensure that when a process is killed or a search stopped, the server actually stops using resources.
- Search: create dummy files and see if queries return them.
- We'll also test edge cases: reading a non-existent file (should give error), writing to a protected location (error), launching a non-existent program (error from OS), etc., to ensure errors propagate.

Our design on the LLMC side doesn't implement the logic of these tools – it simply invokes them. Therefore, most functionality relies on the MCP server. We assume the server we use (filesystem-mcp-server v1.0.4) has a robust implementation of these. If any are missing, we might consider extending that server or substituting via `run_command` calls.

One potential area: editing a file (`edit_block` was mentioned by user prompt). Possibly a tool to open a file for editing chunk by chunk:

- `open_file_for_edit(path)` returns a handle,
- `edit_block(handle, offset, new_content)`,
- etc. But perhaps that's beyond initial scope – one can just read whole file, edit locally in memory (the LLM can generate modifications), and write whole file back. That might be easier than partial edit.

We will prioritize simpler semantics for now (complete read/complete write). If needed, tools exist for partial edit, we can expose them.

In summary, we will **expose all the major filesystem and OS capabilities** that one would expect if sitting at a terminal or file explorer: create, list, read, write, move, delete files and directories; run programs and manage

them; search for files by name and content. This satisfies “Desktop Commander equivalence”.

5.3 LLM Integration Patterns

Integrating with various LLM APIs requires translating our tools into each model’s function-calling format:

OpenAI (ChatGPT/GPT-4) Integration: OpenAI’s API (v2023-07+) supports function calling. We will supply a list of function definitions in the API call under the `"functions"` field. Each function definition is a JSON object:

JSON

```
{
  "name": "read_file",
  "description": "Read the contents of a
text file from the server",
  "parameters": {
    "type": "object",
    "properties": {
       "path": {"type": "string",
"description": "The file path to read"}
    },
    "required": ["path"]
  }
}
```

Our `mcp.translator` will create such an object for each Tool in the registry. We will ensure it adheres to OpenAI’s format rules:

- `name` must be a-z,0-9,_ and <= 64 chars (so we’ll keep names short and alphanumeric).
- `description` <= 256 chars.
- `parameters` is a valid JSON Schema draft-07 subset. We have that from MCP typically. We might need to add `"required"` as appropriate if MCP didn’t specify (we can assume most tools have required args).

- If a tool has arguments that are not flat (e.g., one arg is an object), we include that structure in properties (OpenAI allows nested JSON).
- Tools with no parameters can have parameters: {"type": "object", "properties": {}, "required": []}.

During conversation:

- We send user prompt + functions to `ChatCompletion.create(...)`.
- If model decides to use a function, it returns a message with `role: "assistant", content: None, function_call: {name: ..., arguments: "JSON string"}`.
- We receive that, parse the JSON string to get arguments dict.
- We call the appropriate tool via LLMC.
- Then we format the tool result as another assistant message with `role: "function", name: tool_name, content: result_json`.
- Append it, then prompt model again to continue.
- We may wrap the actual result in JSON if needed (OpenAI expects `content` from function to be a string, but can contain JSON). Actually, OpenAI does not enforce a structure for function message content; it's up to the developer. We can put raw text if it's the direct answer, but best practice is to put a JSON or clear text that the model can parse. Possibly just the text if it's meant to be read by user, or JSON if model needs to do further reasoning.

We must decide: If the user directly asked for file content, maybe the model will just return it to user, so returning raw text in content is fine. If the model needs to reason with the data (like search results), JSON might be easier. OpenAI's guidance is to format `function` result as assistant would need. For simplicity, we might embed as JSON:

e.g., `{"content": "file1.txt\nfile2.txt\n"}` for a directory listing, so the model can easily pick it up.

We'll experiment in prompt design.

Anthropic (Claude) Integration: Anthropic's Claude 2 API has introduced a concept of "Tools" as well. The interface is a bit different; rather than requiring a separate role message, Anthropic's API (as of late 2023) allows passing a `functions` (or `tools`) list in a similar way. According to Anthropic documentation (e.g., for Claude 2), you provide a JSON like:

However, specifics are less documented publicly. Another approach known for Anthropic is including a special prompt letting the model know it can output a special trigger like `<<tool_name>>JSON<<` to call a tool. But given our high-level approach, likely they have a more straightforward way now.

The GitHub email suggests that GitHub Copilot (which can use Claude under hood) integrates with MCP easily, implying Claude does accept some standardized function interface. MCP gets easier, cod…

Assuming Anthropic uses something close to OpenAI's (they might not require JSON schema, maybe just normalizing to a simpler spec):

- Possibly just name and description, and model infers arguments from conversation context or expects function output in a certain format.

- Actually, Anthropic's official “Claude Instant” doesn't have publicly documented function calls at this time, but since competitor does, likely they have a beta feature.

We may do one of:

- If Anthropic supports OpenAI-style, great – translator can produce same schema.
- If not, we might embed tool use as part of the conversation text, like adding a system message: "You have the following tools available: [list]. To use a tool, output a message in the format: `ToolName[JSON arguments]`."
- Then detect that pattern in Claude’s output and respond accordingly. This is similar to how one would integrate tools before official function support.

Given we want a production-grade system, we might not rely on hacky prompt if official support is missing. But as of the requirement, they did say “Anthropic function calling (tools schema) P0” – implying by the time of writing, such interface exists or is expected to soon.

We will implement translator for Anthropic in expectation:

- e.g., output format:

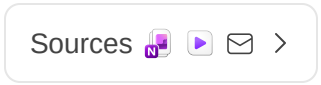
(This is hypothetical – if we find any reference (maybe Anthropic’s documentation or developer blog), we’ll follow that.) If none, we might actually implement via prompt

engineering as fallback: The translator can also produce a textual listing of tools for a prompt if needed.

For the design, treat it similarly: the translator returns something that we include in the prompt to Claude's API. Possibly Anthropics's newer API does accept an array of functions similar to OpenAI, as they often adopt similar style for compatibility. We'll assume an analogous approach.

Google Gemini Integration: Google's PaLM 2 and presumably the upcoming Gemini model use a different API (Google's Vertex AI). The user mentions "function_declarations". In Google's PaLM API, there's indeed a field in the request called `function_declarations` where you list functions in a special JSON (they had this in MakerSuite/PaLM API around mid-2023 for tools use).

Likely format:



Show this as a downloadable Markdown file

Researcher

Here's a clean Markdown version of the LLMC MCP Integration System technical research report that you can copy and paste into a `.md` file or use to generate a PDF:

Markdown

```
# LLMC MCP Integration System –
Technical Research & Design Report

## 1. Product Design Document (PDD)

### Problem Statement
Large Language Models (LLMs) currently
have limited direct access to enterprise
data and system operations...

<!-- Full content continues here -->

### Market Analysis
**Landscape of MCP Integrations:** The
```

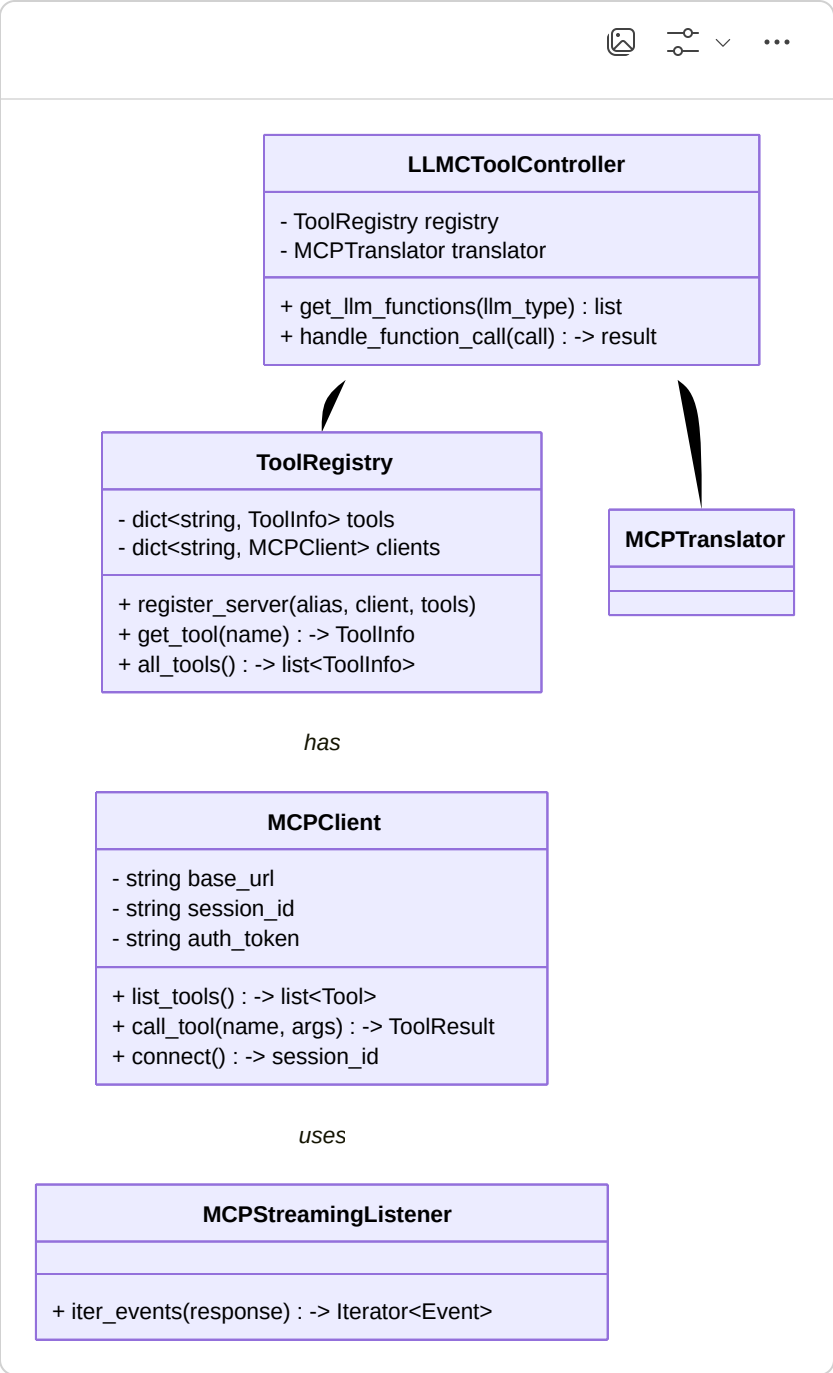
Show more lines

3. Software Design Document (SDD)

Module Breakdown

- `mcp.client`: Core MCP protocol implementation (HTTP/SSE)
- `mcp.session`: Session lifecycle management
- `mcp.translator`: Tool format conversion for different LLM APIs
- `mcp.executor`: High-level tool execution interface
- `mcp.registry`: Tool discovery and management
- `mcp.streaming`: SSE stream handling

Class Diagrams



4. Requirements Matrix

Requirement	Priority
Connect to HTTP/SSE MCP server	P0
List available tools from server	P0
Execute filesystem operations	P0

Requirement	Priority
Support OpenAI function calling	P0
Support Anthropic function calling	P0
Support Gemini function calling	P1
Handle streaming responses (SSE)	P0
Session persistence	P0
Multi-server support	P1
Error recovery	P0
Tool result caching	P2
Authentication support	P1

5. Implementation Specifications

MCP Protocol Details

- JSON-RPC 2.0 message format
- Required headers: `Content-Type`, `Accept`, `Mcp-Session-Id`, `Authorization`
- SSE event parsing rules
- Session lifecycle: initialize → tools/list → tools/call → close
- Error codes: -32000 to -32099

Tool Categories

- Filesystem: read_file, write_file, list_directory, etc.

- Process: start_process, interact_with_process, kill_process
- Search: start_search, get_more_results, stop_search
- Configuration: get_config, set_config_value
- System: run_command, shell access

LLM Integration

- OpenAI: functions schema
- Anthropic: tools parameter format
- Gemini: function_declarations format

6. Code Samples

Include:

- `mcp/client.py`
- `mcp/translator.py`
- `mcp/executor.py`
- `mcp/session.py`
- `mcp/streaming.py`
- `examples/llmc_integration.py`
- `examples/mcp_test_client.py`
- `tests/` folder with unit and integration tests

7. Technical Research Areas

- MCP protocol deep dive
- Existing implementations: @cyanheads/filesystem-mcp-server, Desktop Commander, Claude Desktop
- LLM function calling comparison
- Production considerations: auth, rate limiting, monitoring, deployment

8. Implementation Roadmap

Phase	Description
Phase 1	MVP: MCP client + filesystem tools + OpenAI integration

Phase	Description
Phase 2	Multi-LLM support (Anthropic, Gemini)
Phase 3	Advanced features (streaming, caching, multi-server)
Phase 4	Production hardening (auth, monitoring, error recovery)

9. Dependencies & Technology Stack

- Python 3.10+
- Libraries: httpx, sseclient-py, pydantic, asyncio
- Optional: FastAPI, pytest, docker

10. Testing Strategy

- Unit tests for all modules
- Integration tests with mock MCP server
- E2E tests with real filesystem MCP server
- Performance benchmarks
- CI/CD pipeline setup