ChatGPT

Local-First Tool Suite Implementation for VALIS Agents (Sprint 7)

Current Architecture Overview

PostgreSQL Memory Spine: VALIS already uses a structured memory system backed by PostgreSQL (introduced in Sprint 2). The database schema includes tables like `persona_profiles`, `client_profiles`, `canon_memories`, `working_memory`, and `session_logs` for long-term and short-term memory storage [1]. A `DatabaseClient` (in `memory/db.py`) manages connections to Postgres via psycopg2 connection pooling [2]. High-level access to memory is provided by `MemoryQueryClient` methods (e.g. `get_persona`, `get_client`, `get_top_canon`, etc.) which query these tables for the runtime [3]. This means VALIS can retrieve persona profiles, client info, and memory entries on the fly from the DB instead of flat files. The MCPRuntime uses these methods to inject relevant facts into the prompt without exceeding token budgets – e.g. it fetches top-ranked canon memories and recent working memory with limits based on the context mode [4] [5]. This ensures *persistent, queryable memory* for agents while avoiding prompt bloat (token management was a key goal of Sprint 2) [6].

Provider Cascade & Tool Routing: User requests are handled by a cascade of *providers* orchestrated by the `ProviderManager`. As configured after Sprint 6, the cascade order is: `mcp_execution` → `mcp` → `local_mistral` [7]. The first in line, `MCPExecutionProvider`, is the *tool/command execution layer*. It examines the user prompt for any action intents (like file or process commands) via regex patterns. For example, it will detect phrases like *"list files in …"* or *"read file …"* and map them to specific actions (`list_directory`, `read_file`, etc.) [8]. If a pattern matches, `MCPExecutionProvider` handles the request immediately; if not, it passes the prompt along to the next provider (the main AI model). This design allows VALIS to **intercept tool requests** before engaging the language model. In Sprint 6, this execution layer was implemented with basic functionality: listing directory contents is handled by calling Python's `os.listdir` (via `_call_list_directory`), but other actions were stubbed. For instance, `_call_read_file` and `_call_search_files` currently just return placeholder text indicating what would be done [9] [10] rather than performing the real file operations. The system also logs each command execution to an `execution_logs` table with details (intent, parameters, success, etc.) for audit trail [11] [12].

Persona Execution Flow: If no tool intent is detected (or after a tool action is executed), VALIS falls back to the AI persona response. The `MCPProvider` combines the memory-aware runtime with a local LLM. It uses `MCPRuntime` to **compose a prompt** that includes the persona's profile (role, system prompt, traits) and relevant memory from the database, then sends this composed prompt to the language model (Local Mistral 7B in the current setup) [13] [14]. The cascade ensures that if `mcp_execution` doesn't produce an answer, the query goes to `mcp` (which yields an AI-crafted answer with context), and if that fails, to a raw `local_mistral` fallback without added context [7]. This cascade approach, as documented in Sprint 6, means **VALIS agents can both "think" (via the LLM with memory) and "act" (via tool/command execution)** within a single conversation loop [15] [16]. The persona's reply to the user is normally generated by the LLM, except in cases where a tool was invoked – currently the execution provider returns a hard-

coded success message with the result of the command [17]. (In Sprint 7, we will aim to better integrate these tool results into the persona's voice, as discussed below.)

Tool Implementations for Local-First Execution

To empower VALIS agents with local-first capabilities, we will implement or refine the following tools within the VALIS codebase. Each tool allows controlled access to local resources or data, ensuring that the agent can retrieve information without relying on external APIs.

1. `query_memory(user_id, topic)` **– Structured Memory Query:** Leverage the Postgres "memory spine" to fetch relevant memories or facts related to a given topic or keyword. This tool will enable an agent to ask for information it "knows" (from the database) about a topic. Implementation-wise, we can add a method in the `MemoryQueryClient` or a new function in the tool suite that performs a database query for the given topic. For example, we might query the `canon_memories` table for entries matching the topic in their content or tags [18] [19]. A simple approach is to use a case-insensitive LIKE or full-text search on the `content` and `tags` fields (which are already part of the schema). We should scope the query by persona or user if appropriate – e.g. find facts that the current persona knows about the topic. The result can be a brief list of memory snippets. To avoid flooding the prompt, we'll **limit the number of results** (for example, top 3–5 matches by relevance score) and/or truncate each snippet to a reasonable length. This way, the agent gets the most relevant pieces of long-term memory related to the topic without exceeding token limits. (The groundwork for this exists – the `canon_memories` are tagged and scored [1], so we can reuse those attributes to filter and sort results.) The `query_memory` tool essentially gives the agent an API to its own knowledge base, which it can use to answer user queries more factually by retrieving stored info instead of relying purely on the LLM's parametric knowledge.

2. `read_file(path)` **– Secure File Reading:** Implement this tool to allow an agent to read the contents of a local text file and return it (or a portion of it) to the user. In the current code, `read_file` is identified as an intent, but the actual execution function simply returns a placeholder string ("[EXEC] Read file contents from X") without reading anything [9]. We will replace that stub with real file I/O logic. Key considerations for safety and performance:

3. **Path Whitelisting:** Only allow reading files from certain directories. We will enforce that the `path` is within an approved directory (for example, a workspace like `C:\VALIS\` or a configurable data folder). The Sprint 6 implementation already normalizes relative paths to `C:\VALIS` by default [20], which keeps most file operations in a contained location. We can enhance this by checking the resolved absolute path against a whitelist (and perhaps leverage the planned `command_allowlist` table for more dynamic control [21]). If a path is outside the allowed scope (e.g. pointing to system directories or user's private files not meant for VALIS), the tool should refuse with a safe error message ("Access denied").
4. **File Size Limits:** Reading an entire file could yield more text than we can safely put into the AI's context or response. To enforce token limits, the tool should only read up to a maximum size. For instance, we might read the first N kilobytes or first M lines of the file. This can be configured based on typical prompt token limits (OpenAI GPT-4 context or local model context). If the file is larger, we truncate the output and perhaps append a message like "<...content truncated...>". This ensures we

avoid prompt bloat or excessive response length, in line with VALIS's token management approach [22].

5. **File Type and Encoding:** Focus on reading text files (e.g. `.txt`, `.md`, `.py`). We should open files in text mode with proper encoding. If the file is binary or cannot be decoded as text, the tool should handle this gracefully – possibly returning an error "Cannot read binary file" or a sanitized summary (rather than gibberish). This prevents the agent from outputting unreadable data.

6. **Output Formatting:** The tool can return the file content as a plain text string. We might prepend a header like "Contents of `<filename>`:\n" for clarity. If truncation occurred, clearly indicate that. The idea is to provide useful content to the user while maintaining security. All such operations should be logged (the existing `_log_execution` will capture the action and a preview of the content returned [11] [23]).

7. `search_files(keyword)` **– Local File Search:** Allow the agent to search for files by name or content containing a given keyword, within allowed directories. In Sprint 6, a "search_files" intent was defined, but `_call_search_files` currently just returns a placeholder "[EXEC] Searched for files matching X..." without actually searching [10]. We will implement this properly in Sprint 7. There are two modes to consider:

8. **Filename search (wildcards):** If the `keyword` looks like a filename or pattern (e.g. contains `*.txt` or an extension), we interpret it as a filename search. We can use Python's glob or os.walk to find files that match the pattern in the permitted directory tree. For example, if the user says "find files named `*.py` in logs", the tool would search under the "logs" folder (if within allowed path) for any `.py` files.
9. **Content search:** If the keyword is a regular word or phrase (e.g. "error" or "Jane Doe"), the tool will search **within** text files for that substring. This can be done by scanning files in the allowed directories (perhaps recursively, but bounded to avoid huge spans). For each file, we can do a simple substring or regex search. Because searching the entire filesystem could be expensive, we restrict to an index or specific folders that are known and approved (the prompt mentions "indexed or pre-approved directories"). In practice, we might maintain a config list of directories that are small enough to search or have been indexed. (If performance is a concern, we could require the user to specify a directory to search in, or implement a cached index in the future; but initially, a straightforward scan of the VALIS workspace for matches is acceptable given likely modest scope.)

10. **Result formatting:** The output should be a concise list of results. For a filename search, we can list file paths (e.g. "Found files: `scripts/test.py`, `scripts/utils.py`"). For content search, we might list filenames and an excerpt of the line containing the keyword. We must also cap the number of results – for example, return the first 5 matches – to avoid overwhelming the response. If no files are found matching the query, return a polite "No matching files found." As with other file tools, ensure we only search within whitelisted locations and skip any files or folders that are off-limits. All searches should respect the same safety guardrails (no searching system directories, etc.). This tool effectively gives the VALIS agent a way to **locate information in its local file corpus** (logs, notes, config files, etc.) by keyword, functioning like a localized "grep" or search engine.

11. `list_directory(path)` **– Directory Listing:** This tool is essentially already in place – in Sprint 6, `list_directory` was implemented to list files in a given folder. The current implementation uses

`os.listdir` and then labels each entry as `[DIR]` or `[FILE]` in the output [24]. We will carry this forward with additional safeguards. The improvements will include:

12. **Path validation:** Similar to `read_file`, ensure the `path` is allowed. In fact, `MCPExecutionProvider._extract_path` already defaults to `C:\VALIS` if the prompt doesn't specify a path, and it tries to catch relative paths and anchor them safely [20]. We will extend this by blocking explicit disallowed paths. For example, if a user tried to list `C:\Windows\System32`, the tool should refuse because that's outside the permitted scope. We can implement a check like: if the normalized absolute path does not start with our allowed base directory (`C:\VALIS` or whatever is configured), then return an error or an empty result with a warning. This prevents the agent from snooping in areas it shouldn't.
13. **Output clarity:** The existing format (prefix "[DIR]" or "[FILE]" to each entry name) is user-friendly and we will retain it. We might also sort the listing (e.g. directories first, or alphabetical) for consistency.
14. **Large directories:** If a directory has a huge number of files, we should consider truncating the list or summarizing (e.g. "N entries not shown"). However, in many cases the agent will be dealing with its specific workspace or project directories, which likely won't be extremely large. We can implement a sensible limit (for example, list the first 100 entries) just in case.
15. The `list_directory` tool is useful for navigation and situational awareness of the file system. By keeping it constrained to safe locations and sizes, we preserve security. We will continue logging its usage in the `execution_logs` (including the path accessed and success/failure) as done in Sprint 6.

Integration into VALIS Runtime

Registering Tools in the Provider Cascade

The VALIS runtime uses the `ProviderManager` to initialize and manage providers (which include the tool execution layer) [25]. To incorporate the new tools, we will update the `MCPExecutionProvider` class in `valis2/providers/mcp_execution_provider.py`. This is the ideal place to register and implement our tools because it already handles routing of detected commands to specific functions. Concretely:

- We will replace the stub implementations of `_call_read_file`, `_call_search_files`, etc., with the real logic described above. For example, `_call_read_file(self, path)` will attempt to open and read the file from disk (with try/except for error handling) and return the actual content or an error message [9]. Similarly, `_call_search_files(self, path, pattern)` will perform the directory scan and return matches instead of a placeholder [10]. The `execute_desktop_command()` method already contains a dispatcher that calls these functions based on the action name [26] [27], so hooking in our new logic will be straightforward.

- If the tool implementations become too large, we might factor them into a separate module (for instance, a new `tools.py` in the `valis2` package) and have `_call_read_file` simply call `tools.read_file(path)` internally. This would keep the provider class clean and make it easy to maintain the tools or even reuse them elsewhere. However, initially, it may be simplest to implement within `MCPExecutionProvider` for cohesion.

- **Tool Registration:** The provider is already instantiated and added to the cascade as `"mcp_execution"` in the ProviderManager's `_initialize_providers` [25]. We don't need to add new providers; we are augmenting the existing one. We should, however, update any configuration or constants if needed (for example, if `providers.json` or similar config file lists available intents or if there's a command allowlist to update). Since Sprint 6 anticipated a `command_allowlist` table for security [21], we may integrate a check against that table in our tool functions (e.g., ensure the requested `path` is in an allowlist of safe paths). This would make the registration dynamic – only execute if the action and path are permitted by policy.

- After implementing, all these tools will still be invoked via the same flow: user input → `MCPExecutionProvider.ask()`. That method will detect the intent and call `execute_desktop_command()` which now executes the real tool and returns the result [28] [29]. If successful, the provider returns a response to the user indicating success and includes the tool's output [17]. If no intent was detected, it returns an error flag so that ProviderManager will move on to the next provider (the AI model) [30]. This integration means from the outside, the provider usage doesn't change – we're just making sure it does meaningful work when those tool intents occur.

Tool Awareness in Persona Prompts

Currently, the personas (Kai, Luna, etc.) have system prompts that define their role and style, but do not mention tool usage (e.g., Kai's prompt: "You are Kai, an energetic personal coach..." [31]). Since the tool execution is handled transparently by the system, the AI model might not fully realize that actions like file access are available. To align the model's behavior with the new capabilities:

- We will update the **prompt templates or system messages** for personas to include a note about tools. For example, we can append something like: *"You have access to certain tools such as memory lookup and file system browsing. When necessary, these will be used to help retrieve information or perform tasks."* This can be added in the persona's `system_prompt` (in the database) or in the static prompt template loaded by MCPRuntime for that persona. By doing this, the AI will be less likely to say "I cannot access files" or hallucinate an answer when a factual tool-based answer is possible. Instead, it will understand that such queries result in actions.

- The **bootstrapping of the conversation context** could also explicitly mention tool results when they are used. As it stands, if a user asks for a file to be read, `mcp_execution` handles it entirely and returns the content without involving the language model's persona response. This ensures speed and accuracy (since it doesn't have to go through the LLM), but the response might seem a bit robotic or out-of-character. In the future, we might change this flow so that after the tool fetches the data, the persona's LLM is invoked to present the result in a more conversational tone. For example, the system could inject the file's content into the persona's prompt (perhaps as: *"(The user requested to read file X. Its content is:\n* `\n<file text>\n` *\n)"* and then ask the LLM to continue the response). This would preserve the persona's voice and allow for follow-up questions or summaries. However, this kind of multi-turn tool usage would require more complex prompt management. For Sprint 7, as a first step, we ensure the persona knows tools exist and the rest can remain as a direct system response for simplicity.

- **Persona memory context:** The `query_memory` tool effectively pulls information from the same source the MCPRuntime would when composing a prompt. If the agent uses `query_memory` mid-

conversation (say the user explicitly asks the agent *"What do you know about XYZ?"* and the agent decides to call `query_memory`), we should be cautious not to double-inject the same info again in the prompt. One approach is to treat `query_memory` results as *on-demand info retrieval* and perhaps not include those same facts again in the next cycle's automatic memory injection (to avoid redundancy). This is an internal detail, but worth noting as we integrate tool awareness with persona reasoning.

In summary, by adjusting the persona's system prompt or templates, we make the AI agent cognizant of its new local-first tools. It will then either invoke them implicitly (through our regex triggers) or at least respond in a manner consistent with having those capabilities. This integration bridges the gap between the **procedural tool execution** and the **conversational persona**, so the overall experience remains coherent.

Enforcing Token Limits and Safety

Introducing these tools requires strict adherence to VALIS's safety and efficiency principles. We will enforce token and security constraints at multiple levels:

- **Token Limits:** As mentioned earlier, any content retrieved via tools must be trimmed to fit within the model's context limits. VALIS already uses context mode limits for memory insertion (e.g. only a certain number of memory entries based on "tight", "balanced", etc.) [32] [33]. We will apply similar limits for tool outputs. For example, we might decide that at most 200 tokens of file content can be returned in a single response. If a file is longer, the agent could inform the user and perhaps offer to send more in a follow-up if needed. This keeps responses concise and avoids swamping the conversation. It's also important for performance, especially with a local model that has a fixed context size.

- **Safety and Permissions:** Sprint 6's design included several safety checks – regex-based intent filtering, path sanitization, and an audit log for every command [34]. We will extend those. **Path whitelisting** has been emphasized for each tool: essentially treating a specific directory (or set of directories) as the sandbox that VALIS can operate in. The `command_allowlist` table (if already created in the DB schema) can be used to store these allowed paths or filename patterns [21]. Our tool functions will consult this allowlist: e.g., `read_file` will only proceed if the file path matches an entry on the whitelist (or at least doesn't match any known blacklist entry). During Sprint 7, if the allowlist feature isn't fully implemented yet, we will hard-code the allowed directory (like VALIS's working directory) as an interim solution – this is essentially what the code does now by defaulting to `C:\VALIS` directory [35].

- **Execution Isolation:** Since we are running local commands, we need to ensure they don't inadvertently cause harm. Reading files and listing directories are relatively safe (as long as we don't allow writing or deleting files at this stage). The `list_processes` tool (from Sprint 6, not explicitly requested in this sprint's list) should similarly be restricted to just reading process info. We will not execute arbitrary shell commands – the tools are narrow and controlled, which is good for security. In the future, if more powerful operations are allowed, we might execute them in a sandboxed subprocess with timeouts. For now, we continue using Python's built-in functions (like `os.listdir`, `open`) which run in the VALIS process. We must be mindful that if, say, `read_file` tries to open a huge file or a named pipe, it could block – to mitigate this, we can set timeouts or use

non-blocking reads where possible. However, typical usage should be fine, and we already have a 30-second timeout mechanism mentioned for commands [34] (likely enforced at the provider or API level).

- **Logging and Monitoring:** The existing logging to `execution_logs` will be very useful [36] [37]. After implementing the tools, we will test that each usage indeed creates an entry with the correct details (execution_id, persona_id, client_id, the action name, parameters, success flag, and a preview of results). This gives us a permanent record of what the agent accessed. In a multi-user or production scenario, an admin could review these logs to ensure no unauthorized file was read, etc. Moreover, if something goes wrong in a tool (exception occurs), the error is caught and logged by the provider, and a failure response is returned to the user safely [30]. We will maintain this behavior – e.g., if `search_files` encounters a permission issue or read error, it should not crash the system; it will log an error and politely inform the user it couldn't complete the request.

By combining these measures, we ensure the new local-first tools are used in a *secure, efficient,* and *controlled* manner, consistent with VALIS's design goals.

Routing Layer Recommendation

One architectural decision for implementing these tools is how the agent will invoke them – i.e., the "routing layer" between the AI and the local system. We have a few options, given the VALIS architecture:

- **Native Python Calls (in-process):** This is the current approach used by `MCPExecutionProvider` – it directly calls Python functions for listing directories, etc., within the same process that runs the VALIS backend. This approach is fast (no IPC overhead) and keeps everything local, which is aligned with "local-first." It also minimizes the complexity: the tools can share memory/state with the rest of the system (e.g., use the same database connection object `db`, or config). For Sprint 7, this is the recommended approach. We will continue using in-process function calls for `query_memory`, `read_file`, `search_files`, and `list_directory`. The code already demonstrates how seamless this can be – for instance, calling `os.listdir` and building the result in `_call_list_directory` is done directly with minimal fuss [24]. This means the agent's tool use does **not** require formatting a special request or parsing a response in text; it's handled as a function return value that we insert into the final output (or return directly to user). This approach has the lowest latency and avoids running up the token count (since we're not feeding the raw file content through an LLM just to parse it; we're responding with it directly).

- **FastAPI/HTTP Layer:** VALIS already uses FastAPI for its API endpoints (e.g., the chat endpoint that clients hit, and some admin endpoints). We could expose these tools as internal API endpoints as well (for example, `POST /api/tools/read_file`), and have the agent call those endpoints. However, since the agent's "brain" is server-side, calling an HTTP endpoint from the same server is unnecessary overhead – it would be simpler to just call the function. FastAPI would be useful if we envisioned the agent running in one process and a separate "tool server" running in another (or on another machine). At present, that separation isn't needed; everything runs within the VALIS service on the local machine. Therefore, introducing HTTP for tool calls would add complexity (serialization to JSON, networking stack) without clear benefit. It's better to keep tool invocations as function calls for now, and perhaps only use the FastAPI layer to expose results to a user interface if needed

(which is already the case: the final answer, whether from a tool or LLM, is returned via the existing API).

- **Custom RPC or Message Bus:** Another possibility is using a JSON-RPC mechanism or an IPC (inter-process communication) for tools. This could be considered if we want to **sandbox the tool execution**. For example, running file system operations in a lower-privilege process and communicating via JSON-RPC would enhance security – even if the agent went rogue, it couldn't directly execute OS calls, it would have to go through a controlled channel. This idea aligns with the "future optional remote expansion" mentioned in the objectives. We could design our tool interface such that it could later be backed by an RPC call instead of a direct call. For instance, today `read_file(path)` might open a file directly, but tomorrow it could send a JSON-RPC request to a dedicated file-reader service (which might even be running on a different machine, enabling remote file access if VALIS expands beyond local).

- **Recommendation Summary:** For **Sprint 7**, stick with the simplest approach: **native in-process tool execution**. It fits the current VALIS architecture (monolithic Python application with modular providers) and avoids unnecessary overhead or refactoring. We will implement the tools as Python functions and call them directly from `MCPExecutionProvider`. At the same time, we should design the code with an eye towards flexibility. Abstract the tool implementations behind a clear interface so that in the future, if needed, we could swap the underlying call to an RPC or HTTP service without changing how the provider works. For example, a `ToolManager` class could have methods `read_file`, `query_memory`, etc. that currently just call local functions, but could be updated to call external endpoints later. By doing this, we "prepare for future optional remote expansion" – meaning if we decide to offload tools to a microservice or cloud function, the change won't require a complete redesign.

In summary, **FastAPI** is not necessary for internal tool calls (it's better suited for client<->VALIS communication), and a **JSON-RPC/remote tool server** can be an enhancement for down-the-road security or scalability. The VALIS Sprint 7 scope can be achieved efficiently with in-process calls, keeping the system local-first and fast. This aligns with how the local LLM is integrated (notice that even the local Mistral model is called via an HTTP request to `localhost:8080` in `LocalMistralProvider` [38] – effectively treating it as a microservice). For tools, we don't even need a separate process yet – we can run them in the same process safely with proper checks. As VALIS grows, we'll revisit the routing layer, but the recommended starting point is the straightforward one.

Conclusion

Sprint 7 will enhance VALIS with a robust **local-first tool suite**, enabling agents to securely query memory and interact with local filesystems. We began by auditing the repository: VALIS already has a strong foundation with its PostgreSQL-backed memory (canon/working memory, persona and client profiles) [1] and a provider cascade that intercepts certain commands [15]. Building on this, we will implement the `query_memory`, `read_file`, `search_files`, and `list_directory` tools fully. These implementations will follow the principles of **minimal prompt bloat** (retrieving only necessary information, respecting token limits) and **maximum safety** (restricting file system access to whitelisted areas, sanitizing inputs, and logging all actions). The tools will be integrated into the existing VALIS runtime by extending the `MCPExecutionProvider` (the execution layer) [25], and we will update persona prompts to acknowledge these new capabilities, keeping the AI's behavior consistent with its actual abilities.

Overall, the best approach is to keep tool execution local and lightweight – as internal function calls – which leverages VALIS's current architecture and avoids performance overhead. We've ensured that each tool's design has considered future needs (for instance, how to swap in an RPC layer or how to handle very large outputs) so that VALIS remains scalable and secure. After these additions, VALIS agents will be able to **retrieve stored knowledge, read files, search through data, and navigate directories on behalf of the user** – all while logging their actions and staying within safe operational bounds [16] [34]. This local-first tool suite will significantly broaden what VALIS can do (moving it closer to an "agent with arms and hands" as the project envisions), without compromising the system's efficiency or integrity.

---

[1] [3] [6] [22] SPRINT2_COMPLETE.md
https://github.com/Loflou-Inc/VALIS/blob/65223cc3babbe912773bbc56c4660d66a4ddc8be/valis2/SPRINT2_COMPLETE.md

[2] db.py
https://github.com/Loflou-Inc/VALIS/blob/65223cc3babbe912773bbc56c4660d66a4ddc8be/valis2/memory/db.py

[4] [5] [32] [33] mcp_runtime.py
https://github.com/Loflou-Inc/VALIS/blob/master/valis2/core/mcp_runtime.py

[7] [15] [16] [21] [34] SPRINT6_COMPLETE.md
https://github.com/Loflou-Inc/VALIS/blob/65223cc3babbe912773bbc56c4660d66a4ddc8be/valis2/SPRINT6_COMPLETE.md

[8] [9] [10] [11] [12] [17] [20] [23] [24] [26] [27] [28] [29] [30] [35] [36] [37] mcp_execution_provider.py
https://github.com/Loflou-Inc/VALIS/blob/65223cc3babbe912773bbc56c4660d66a4ddc8be/valis2/providers/mcp_execution_provider.py

[13] mcp_provider.py
https://github.com/Loflou-Inc/VALIS/blob/65223cc3babbe912773bbc56c4660d66a4ddc8be/valis2/providers/mcp_provider.py

[14] [38] local_mistral.py
https://github.com/Loflou-Inc/VALIS/blob/65223cc3babbe912773bbc56c4660d66a4ddc8be/valis2/providers/local_mistral.py

[18] [19] [31] seed_data.py
https://github.com/Loflou-Inc/VALIS/blob/65223cc3babbe912773bbc56c4660d66a4ddc8be/valis2/memory/seed_data.py

[25] provider_manager.py
https://github.com/Loflou-Inc/VALIS/blob/65223cc3babbe912773bbc56c4660d66a4ddc8be/valis2/core/provider_manager.py