ChatGPT

Architectural Improvements for Ahma MCP

Ahma MCP is already a powerful async tool adapter, but its maintainability can improve with a more modular architecture and clearer alignment to its guiding requirements. Below we present key recommendations – from splitting the project into subcrates to refining documentation – along with a prioritized refactoring plan. These changes will reduce context size for AI coding assistants, isolate features, and make adding future components (like a web UI or Okta auth) much easier.

1. Modularize into Subcrates for Clarity and Expansion

Currently, `ahma_mcp` is a single crate that houses both the core library *and* the CLI/server executable [1]. This monolithic design means all code (tool execution logic, config handling, CLI parsing, MCP protocol integration, etc.) lives in one place. Breaking the project into a **Cargo workspace** with focused subcrates will significantly improve manageability:

- `ahma_core` – a library crate containing the core functionality: tool definition structures, config loading/validation, the adapter engine, shell pool, async execution, and the MCP service logic. This crate would encapsulate all tool-agnostic logic and business rules, essentially what's now in `src/lib.rs` and related modules.
- `ahma_shell` – a binary crate for the command-line interface. This would include `main.rs` (CLI argument parsing via Clap, logging setup, and launching the server) [2]. By separating it, the core library isn't cluttered with CLI concerns. The `ahma_shell` binary can depend on `ahma_core` to start the server or run a one-shot tool call.
- **Future Extensions** – the workspace can easily grow to include new crates:
- `ahma_web`: a crate providing a secure HTTPS/web interface for the MCP server. It could embed a tiny web server that exposes similar tool functionality over HTTP, reusing `ahma_core` internally.
- `ahma_okta`: a crate (or module) handling authentication (e.g. via Okta). This might integrate with `ahma_web` to ensure only authorized users can invoke tools, or provide an auth layer for the server. Keeping it separate means the core remains auth-agnostic.

Why subcrates? Each subcrate keeps related code and dependencies scoped, making it easier for both humans and AI assistants to navigate. For example, all asynchronous execution and shell management lives in `ahma_core`, while all CLI argument handling (e.g. Clap definitions) lives in `ahma_shell`. This minimizes the "context size" for each part of the system, so tools like Copilot won't be distracted by unrelated code when working on a specific feature. It also forces clear interfaces between components (e.g., `ahma_shell` would call into `ahma_core::AhmaMcpService` instead of internal functions), reducing coupling.

Implementation notes: Converting to a workspace involves creating a top-level **Cargo.toml** (workspace) and individual `Cargo.toml` for each crate. You would move files: most of `src/` into `ahma_core/src/`, while `main.rs` and any CLI-specific code goes to `ahma_shell/src/`. Public APIs (functions, structs) in `ahma_core` will need to be exposed so that `ahma_shell` can use them. For instance, `ahma_shell` will call `AhmaMcpService::start_server()` from the core crate instead of having that logic inline. This split

is mostly along the lines already hinted in comments (the lib documentation notes the crate provides both core and CLI functionalities [1], which we can now formally separate).

Additionally, ensure that common types (e.g. config structs, `ToolConfig`, etc.) are accessible from `ahma_core`'s public API so the shell and other crates can use them. Once done, **adding** a new component like `ahma_web` becomes straightforward: it would depend on `ahma_core` and perhaps share some config or adapter usage, but it won't clutter the other crates. Each crate can even have its own tests focused on its domain (e.g. core logic tests vs. integration tests for the CLI).

2. Align Code Structure with the Requirements

The **requirements.md** document is the project's "single source of truth" [3], enumerating core principles (R1–R7) and expected behaviors. The codebase should be organized in a way that each major requirement is clearly implemented by a distinct component or layer – this makes it easier to maintain and extend without breaking those rules:

- **Configuration-Driven Tools (R1)** – The code already honors this by loading tool definitions from JSON files at runtime [4]. To keep this maintainable, isolate all config parsing and validation into its own module (or even sub-module in `ahma_core`). Currently, the `config` module handles JSON deserialization and schema enforcement. Consider keeping *all logic related to tool definitions* (JSON schema, validation, default behaviors) in one place (e.g., an `ahma_core::config` namespace). This way, when new options or fields are added to the tool format, changes remain localized. It also reinforces R1.3: no recompilation needed to add tools – the Rust code should **never** contain tool-specific branches. Double-check that this holds true (the requirements explicitly forbid adding tool logic in Rust [5]). If any stray tool-specific condition is found in code, refactor it out so that all tool behavior comes from data. A well-structured `config` module (with schema definitions, e.g. using `schemars`) makes abiding by R1 and R5 (schema validation) straightforward.

- **Async-First & Sync Override (R2 & R3)** – The architecture for async execution (via a pool of shells and background tasks) is implemented in modules like `adapter` and `shell_pool`. Ensure these concerns are cleanly separated from others. For example, the `Adapter` is stateless with respect to tool configs [6], which is good – it just executes commands either through the shell pool (async) or direct process spawn (sync). This separation means you can modify how async tasks are handled (say, use a different async runtime or add cancellation features) without touching config parsing or CLI code. It might be beneficial to introduce interfaces/traits for execution vs. planning. For instance, a trait for "ToolExecutor" that `Adapter` implements, and which the service calls – this could make it easier to swap out execution logic or test it in isolation. Overall, preserve the principle that asynchronous orchestration is the default: any code path that introduces a blocking operation should be clearly marked (or confined to the synchronous override logic when `"synchronous": true` in config). Keeping async handling code concentrated (in `adapter`, `operation_monitor`, etc.) helps when tuning performance (R4).

- **Performance (R4)** – The `shell_pool` is a key component to meet the 5-20ms startup goal [7]. The current design (maintaining a pool of zsh processes) should remain in the core crate. Consider if the shell pool logic can be further modularized – e.g., a sub-module just for shell management. This will ease future improvements like supporting different shells or adjusting pool parameters. Because

performance is critical, you might add more metrics or logging around the shell pool usage; having it encapsulated in one module makes this easier. Also ensure that `shell_pool` usage is abstracted behind a trait or simple API in `Adapter`, so that if one day an alternative execution mechanism is needed, the impact is local.

- **JSON Schema Validation (R5)** – The project uses `schemars` and has a `schema_validation` (MTDF validator) component. This is great for catching config errors early. One improvement: *integrate the schema validation step firmly into the startup flow.* For example, when the server loads tools at startup (`main.rs` scanning `.ahma/tools`), it should invoke the schema validator on each JSON file before proceeding. If this isn't already done, implement it so that invalid definitions yield clear errors and are skipped or abort startup. Keeping the validator usage centralized (perhaps within the config loading function) ensures all tools are validated uniformly. Since R5 is a core requirement, consider making the schema validator a required step in tests as well (maybe a test that all sample tools in the repo pass validation). In terms of architecture, `schema_validation` can be part of `ahma_core::config` or a sibling module. Just ensure the design is such that adding new schema rules (e.g., for future fields) is straightforward – possibly by generating the schema from Rust structs (via schemars) to avoid divergence.

- **Sequence Tools (R6)** – The requirements mention sequence support (`"command": "sequence"` with steps) [8]. If the implementation for this exists, verify it's cleanly integrated. Likely, the `Adapter` or `mcp_service` handles detecting a sequence config and running multiple commands in order. This logic should be encapsulated (perhaps in the adapter or a dedicated `sequence_executor` module) to keep it from complicating the normal code paths. A possible refactor is to treat sequences as just another tool execution, but where `Adapter` sees the `"sequence"` command and then iterates through the steps. Make sure any special cases (like adding delays between steps, per R6.3 [9]) are implemented in one place. If sequence handling code is scattered, consider consolidating it. This will make future enhancements (like conditional sequence steps or parallel steps) easier to implement without touching unrelated code.

- **MCP Service Interface** – The `AhmaMcpService` struct ties everything together by implementing `ServerHandler` trait methods (`list_tools`, `call_tool`, etc.) [10]. **This file is quite large**, as it must handle a variety of request types and orchestrate between config data, adapter execution, and client callbacks. To improve maintainability:

- Break down the `mcp_service.rs` logic into helper functions or even sub-modules. For instance, you might factor out the `list_tools` implementation into a separate function or module that deals with translating loaded configs into the MCP `Tool` schema. Similarly, `call_tool` handling could be split by the type of tool call (regular tool vs. special commands like `await` or `status`). This would reduce the length of the `AhmaMcpService` implementation and make each piece more focused.

- Ensure that `AhmaMcpService` remains a thin coordinator. It should delegate heavy lifting to the `Adapter` (for execution), to the config structures (for knowing defaults, timeouts, etc.), and to callback senders for notifications. If any of the trait methods contain very complex logic, ask if that can live in `ahma_core` modules instead. For example, formatting the progress notifications from `ProgressUpdate` to MCP protocol is already handled by `McpCallbackSender` [11] – a good

separation of concern that keeps protocol specifics out of core logic. Continue this pattern for anything similar.

- **Decouple Protocol from Core** – On a related note, notice that `McpCallbackSender` bridges internal progress tracking with the external RMCP protocol [12] [11]. This is a smart design: it allows the core to generate generic progress events, and only at the boundary do they get translated to JSON-RPC notifications. Continue to enforce this boundary. For instance, if in the future `ahma_web` provides an HTTP API, you could write a different callback sender that sends updates via SSE or WebSockets, while the core `ProgressUpdate` and `OperationMonitor` remain unchanged. Therefore, **keep protocol-specific code (RMCP, JSON-RPC details)** in either the `ahma_shell` (if it's purely for VS Code/Copilot integration) or in clearly separated modules of `ahma_core` (like the `mcp_callback` module now). This way, the core engine doesn't hard-code assumptions about how the client communicates, making it flexible for new frontends.

By structuring the project along these lines, each new **requirement** or feature can be mapped to a specific part of the codebase. An AI assistant (or a human) reading the requirements.md should be able to find the corresponding code section without wading through unrelated functionality. Likewise, when a new requirement is added to the doc, it will be clear *where* in the architecture it should be implemented.

3. Improvements to the Requirements Document

The **requirements.md** file itself is a great asset – it clearly lists goals and even dictates the AI maintainer workflow. However, as the project evolves, this document should evolve too, to remain the "source of truth." A few suggestions:

- **Add an Architecture Overview**: Currently, requirements.md focuses on functional requirements (what the system must do) and workflow/process. It would be beneficial to include a section (or an appended **Architecture** section) describing the intended project structure. For example, outline the separation between core logic and interface, and mention the use of subcrates if you adopt that. This helps future contributors (or AI agents) understand the high-level design before diving into R1…R7. It could be as simple as: "**Project Structure**: The project is organized into multiple crates – e.g. `ahma_core` for core functionality, `ahma_shell` for the CLI, etc. – to enforce modularity. Each new feature should fit into this structure rather than producing monolithic changes." This provides context that complements the specific requirements.

- **Reflect Recent Changes and Plans**: If we implement the subcrate split and plan for `ahma_web` and `ahma_okta`, update the requirements to reflect that plan. Possibly add new requirements or principles, such as a security requirement if Okta auth is in scope (e.g., "All web endpoints must be protected by authentication."). If the project's scope grows to web access, explicitly stating non-functional requirements like *security, concurrency limits,* etc., in the doc will guide development. Since the requirements.md drives all tasks, including such high-level goals ensures they aren't forgotten.

- **Critique and Clarify**: The current requirements are well-chosen. One minor critique is that some details might need clarification as the implementation is fleshed out. For instance, R2.4 says tool descriptions *must guide the AI to continue working while async ops run*. This implies a documentation or UX requirement. Ensure the code actually injects such guidance (perhaps via the `tool_hints`

module). If it doesn't yet, either update the requirement if it's no longer needed or implement the feature. Another example: the document refers to `.ahma/tools/` JSON files, which matches the code, but also the lib documentation referenced "TOML" erroneously [13] – that's a small inconsistency to fix in comments or docs for accuracy. Go through each requirement and double-check if the code honors it; if any gaps exist (like maybe R4.1's 5-20ms shell startup target – is that measured and achieved?), note them and consider adding acceptance criteria or test cases.

- **Living Document Process**: Since this project is AI-maintained, the requirements.md is essentially the spec for every change. Encourage a practice of updating the requirements **first** for any significant refactor or feature (like the ones we propose). For example, before splitting into subcrates, add a note in the doc's architecture section about the multi-crate layout. This keeps the AI agent (or any developer) in sync with the intended architecture. The README already states that all changes are driven by requirements.md [3], so maintaining that document is crucial. We can improve it by not just listing requirements, but also giving *rationale* or context for them. This helps an AI understand *why* a rule exists, which can be useful for problem-solving. For instance, explicitly state the reason behind R1.3 (tool-agnostic core): "This ensures the system can adapt to new tools without code changes, reducing maintenance and preventing hard-coded biases."

In summary, treat requirements.md as a continually refactorable document. After making the architectural changes, **revise the doc to match the new reality**. A concise section on project architecture, plus any new requirements for web/auth, will guide future development effectively.

4. Identify and Clean Up Problematic Areas

As the code has "grown organically" with AI assistance, there are likely sections that need cleanup or rework for smooth future development. Here are the areas to target first:

- **Large or Complex Modules**: The `mcp_service.rs` file, at ~1600 lines, is a prime candidate. Such a large file can be hard to navigate and maintain. Splitting it into smaller pieces (as discussed) or at least grouping related functions with clear comment separators will help. Look for internal duplication or overly complex functions. For example, if `call_tool` has a big `match` handling various tool names or special cases (`await`, `status` commands for monitoring), consider breaking those out. Simpler code is easier for AI to reason about. The same goes for `adapter.rs` (which is ~800 lines) – although it's manageable, ensure its functions are focused. If any function in these modules does "too much," break it into helper functions.

- **Testing Gaps and Incomplete Implementations**: There is evidence of partially-implemented tests (e.g., an `adapter_comprehensive_test.rs.backup` file exists [14]). This suggests some tests were disabled or in progress. It's important to finalize or remove such half-done pieces:

- **Revive or Remove Disabled Tests**: If `adapter_comprehensive_test` was meant to cover edge cases (as its comments indicate [14]), it would be valuable to complete it. It covers crucial scenarios (security of path handling, error propagation, etc.), which align with requirements (like validating path safety per R5.3 and R6). Finishing this test (and others like it) will ensure the system truly meets its requirements and remains stable as new changes are introduced. If certain tests were failing due

to unimplemented features, implement those features or adjust the requirements if they were aspirational.

- If a test is no longer relevant, remove it to avoid confusion. But given the critical nature of what that comprehensive test covers, it's likely worth completing.

- **Documentation and Comments**: Clean up inconsistencies in inline docs. For instance, update the lib.rs module list comment that mentions deserializing from TOML [13] to say JSON, reflecting the actual design [4]. Ensure all public items in `ahma_core` have clear Rust doc comments, since an AI assistant will use those for context. Comments should be accurate and not outdated by previous refactors. If you find any "// TODO" or commented-out code blocks, address them. They either need to be implemented or removed. Residual commented code can mislead AI suggestions, so it's best to clear them out once they're not needed.

- **Adherence to Rust Best Practices**: Do a pass for idiomatic improvements. Since AI generated parts of the code, there might be non-idiomatic patterns (though from a glance, it looks quite solid). Examples to watch for:

- Error handling: ensure errors use thiserror or anyhow consistently (looks like anyhow is used widely, which is fine for an application).
- Clippy warnings: Run `cargo clippy` and fix warnings to keep code quality high (the constitution or principles likely already require no clippy warnings).
- Performance or concurrency gotchas: e.g., are there any `.unwrap()` that could be graceful error handling? Any potential for deadlocks (Arc<Mutex> usage in adapter is common; just be mindful of lock ordering or long-held locks).

- If any functions are too long or deeply nested, refactor them into smaller ones. This not only helps maintainability but also helps AI tools by providing more modular, named steps to follow.

- **Modularity and Extensibility**: After splitting into subcrates, identify any residual tight coupling. For example, does `ahma_core` directly read files from disk (like the tools JSON)? It likely does. Perhaps abstract the filesystem access so that in the future a different source of tool configs could be used (just as a thought). Also, consider using feature flags for optional components (e.g., compiling `ahma_core` with or without `shell_pool` for debug vs performance modes, or to exclude Okta integration unless needed). Keeping things optional and modular ensures that adding a new requirement doesn't bloat the core for those who don't use it.

By addressing these areas, you'll reduce technical debt and friction. The goal is that when a new requirement comes in, the path to implementation is smooth: tests guide expected behavior, docs are up to date, and the code structure makes it obvious where to make a change. Each cleanup improves the chances that an AI agent can correctly implement future tasks without human intervention.

5. Prioritized Refactor Plan with AI Prompts

Finally, to put these recommendations into action, here's a step-by-step plan of **high-level refactoring tasks**. Each item is a big-picture change, listed in logical order. For each, we include a suggested prompt you could give to an AI coding assistant (like GitHub Copilot in chat mode) to help execute it:

1. **Convert to a Cargo Workspace with Subcrates** – First, restructure the repository into a workspace with `ahma_core` and `ahma_shell`.

2. *AI Prompt Example: "Refactor the project into a Rust workspace. Create a library crate* `ahma_core` *for the core logic (move relevant modules there) and a binary crate* `ahma_shell` *for the CLI. Update Cargo.toml files accordingly and fix imports so that the binary uses* `ahma_core`*. Ensure* `cargo build` *and tests run after the split."*

3. **Isolate Core vs CLI Code** – After creating subcrates, make sure all CLI and I/O related code is in `ahma_shell`, and core logic (tool execution, config, service) is in `ahma_core`.

4. *AI Prompt Example: "In* `ahma_shell`*, implement the CLI using Clap (migrate the* `Cli` *struct from the old main). Have it call into* `ahma_core` *for starting the server or running a command. Remove any CLI argument parsing from* `ahma_core`*. In* `ahma_core`*, expose an API like* `AhmaMcpService::run_server()` *that the shell can invoke."*

5. **Update and Enhance** `requirements.md` – Once the structure is in place, revise the documentation.

6. *AI Prompt Example: "Open the* `requirements.md`*. Add a section under 'Core Principles' about project architecture (e.g. R8: Modular architecture with subcrates for core, shell, etc.). Include rationale that core logic is separate from interface for maintainability. Also update any outdated info (e.g., confirm all references to config files are 'JSON', not 'TOML')."*

7. **Refactor Large Modules for Clarity** – Tackle the `mcp_service.rs` (and similar big files) by breaking out pieces.

8. *AI Prompt Example: "Refactor* `AhmaMcpService` *implementation to reduce its size. For example, move the code for generating tool list (*`list_tools`*) into a helper function or new module* `tool_listing.rs`*. Do the same for progress notification handling and any long match statements in* `call_tool`*. The goal is that* `mcp_service.rs` *becomes shorter and primarily orchestrates calls to these helpers."*

9. **Finalize and Fix Comprehensive Tests** – Address the incomplete tests and any failing cases.

10. *AI Prompt Example: "Complete the* `adapter_comprehensive_test.rs` *(currently a .backup file). Ensure it covers edge cases: path escaping, error propagation, sequence tool execution, etc., as hinted by its comments. Use existing functions (like* `Adapter.execute_*`*) to simulate those scenarios. Fix any bugs in the core that these tests uncover. Enable the test (remove* `.backup` *extension) and ensure it passes."*

11. **Code Cleanup and Consistency** – Do a thorough pass for minor improvements.

12. *AI Prompt Example: "Run* `cargo clippy` *and fix all warnings. Ensure all public items in* `ahma_core` *have proper doc comments (especially if the code moved around after refactoring). Remove any unused code or leftover TODO comments. Verify that the crate documentation in lib.rs and README.md reflect the new structure (e.g., mention how to run the CLI after refactor)."*

13. **Prepare for** `ahma_web` **and** `ahma_okta` – Lay groundwork for upcoming features.

14. *AI Prompt Example: "Create a new crate* `ahma_web` *(library or binary as appropriate) that depends on* `ahma_core`*. Scaffold a basic HTTP server (perhaps using Warp or Axum) with one placeholder endpoint, and ensure it can call into* `ahma_core` *to list tools or execute a tool (for now, maybe just returns a static response or version info). Also, design an interface in* `ahma_core` *for authentication hooks (even if not implemented yet). Document in requirements.md that web access should be authenticated, so when adding* `ahma_okta` *we have a place to integrate it."*

Each of these steps should be done in sequence, verifying that the project builds and tests pass at each stage before moving to the next. By following this ordered plan, you will systematically transform **Ahma MCP** into a cleaner, more maintainable system.

Importantly, the prompts given are starting points – when working with Copilot (or another AI), you may need to guide it through the changes, file by file. After each prompt's changes, run the build/test suite to catch issues early. The end result will be a robust architecture aligned with the project's requirements and much easier to extend with new capabilities.

---

[1] [13] lib.rs
https://github.com/paulirotta/ahma_mcp/blob/cd75804b0a88d9609b9c52e2019adb973675f7d9/src/lib.rs

[2] main.rs
https://github.com/paulirotta/ahma_mcp/blob/cd75804b0a88d9609b9c52e2019adb973675f7d9/src/main.rs

[3] README.md
https://github.com/paulirotta/ahma_mcp/blob/cd75804b0a88d9609b9c52e2019adb973675f7d9/README.md

[4] config.rs
https://github.com/paulirotta/ahma_mcp/blob/cd75804b0a88d9609b9c52e2019adb973675f7d9/src/config.rs

[5] [7] [8] [9] requirements.md
https://github.com/paulirotta/ahma_mcp/blob/cd75804b0a88d9609b9c52e2019adb973675f7d9/requirements.md

[6] adapter.rs
https://github.com/paulirotta/ahma_mcp/blob/cd75804b0a88d9609b9c52e2019adb973675f7d9/src/adapter.rs

[10] mcp_service.rs
https://github.com/paulirotta/ahma_mcp/blob/cd75804b0a88d9609b9c52e2019adb973675f7d9/src/mcp_service.rs

[11] [12] mcp_callback.rs
https://github.com/paulirotta/ahma_mcp/blob/cd75804b0a88d9609b9c52e2019adb973675f7d9/src/mcp_callback.rs

14 adapter_comprehensive_test.rs.backup

https://github.com/paulirotta/ahma_mcp/blob/cd75804b0a88d9609b9c52e2019adb973675f7d9/tests/adapter_comprehensive_test.rs.backup