ChatGPT

Converting a Prompt Library into a Proactive MCP Workflow Assistant

Converting Prompt Playbooks into an MCP Server

To turn a Markdown-only prompt library into an MCP server, you can expose each prompt file as an **MCP tool**. For example, the open-source *Prompts MCP Server* demonstrates this pattern: it loads prompt templates (Markdown files with YAML frontmatter) from a `prompts/` directory and exposes operations like adding, retrieving, listing, and deleting prompts via MCP [1] [2]. In practice, you'd package your prompt files and a lightweight server program (e.g. in Node or Rust) that registers each prompt as a tool (e.g. `/planning-process` becomes a tool callable by the AI). Once installed and configured in an MCP client (like Claude Desktop or Cursor), the server advertises these prompt-tools on initialization. This approach essentially wraps your docs as a **tool library** the AI can invoke by name. It requires writing minimal glue code (to read files, handle JSON-RPC requests, etc.), and there are templates and examples available – for instance, the prompts-mcp-server above is published on NPM for quick use [3] [4]. You would follow a similar structure: load your Markdown playbooks, perhaps parse any frontmatter (titles, descriptions, tags), and implement MCP *tool handlers* that simply return the prompt content (or perform any slight processing needed) to the client.

Maintaining Workflow State and Context

To go beyond static prompts and make the assistant **workflow-aware**, your MCP server should maintain some **project state** across calls. MCP servers support long-lived, stateful sessions with the client – the JSON-RPC connection stays open and can carry context between requests [5]. This means your server can "remember" where the user is in the workflow (for example, which phase or step has been completed) and adjust the next suggestions accordingly. In fact, some MCP servers are specifically designed to keep persistent project context. For example, a *Project Context MCP Server* can track details like the project's name, *current phase/status*, task list, and decision history in a local store [6]. By exposing tools to update and query this state (e.g. `get_project_status`, `advance_phase`, or `add_task`), the server and AI together can ensure the conversation always considers what's been done and what's next.

Crucially, an MCP server can include a **Session Orchestrator** component that links multi-step operations together. This allows implementing a DAG or flowchart of actions as a guided sequence of tool calls. For instance, if the user's request triggers a multi-step workflow (say Plan → Code → Test → Deploy), the orchestrator can carry information from one tool call to the next and decide when the sequence is complete [7] [8]. In a web-browsing MCP server example, the orchestrator tracks which page is open so that a "next page" command knows where to resume [7]. By analogy, your server can track which **milestone or gate** the project is at (e.g. "Scope defined", "App scaffolded", "Tests passing") and only allow or suggest tools relevant to the current stage.

Using Diagrams to Plan Next Actions

Since your repository already includes a **Mermaid workflow diagram** mapping out phases P0–P9, the MCP server can leverage this as a source of truth for the process flow. In practice, you might parse the Mermaid **graph** (which is a DAG of steps and decisions) to know the dependencies and order of tasks. Some MCP servers even specialize in diagram analysis – for example, an *MCP Mermaid Server* provides tools to **analyze an existing diagram's structure and provide insights or improvement suggestions** [9]. Leveraging such capabilities, your server could include a custom tool (or integrate an existing one) to read the `workflow.mmd` Mermaid file and determine "what comes next." For instance, if the project is in phase P3 (Data & Auth), a diagram analysis tool could find the next node after P3 and suggest or auto-trigger the commands under P4 (Frontend UX). Even without a dedicated diagram parser, you can encode the workflow logic in a simple state machine or lookup table: e.g. after **Scope Gate** is passed, the server knows the next phase is App Scaffold (P2) and could notify the user or proactively call the `/scaffold-fullstack` prompt. The key is that by **modeling the workflow** (via the Mermaid diagram or an equivalent DAG in code), the server can drive the assistant to follow a structured game plan rather than just reacting blindly. This yields a more *systematic, proactive assistant* that guides the developer through each stage in order.

Enabling Autonomous Step Execution

Finally, to have the MCP server *autonomously advance implementation steps*, you can take advantage of MCP's support for **agent-initiated actions**. Unlike a stateless API, MCP's persistent connection allows the server to send **notifications or prompts back to the client** asynchronously [10]. In practical terms, once the server detects that a certain condition is met (say, the "Test Gate" milestone achieved all green tests), it could emit a notification or result indicating "Gate cleared – ready to proceed to next phase." Modern AI hosts (Claude Code, Cursor, etc.) are built to handle such server-initiated messages, enabling what the MCP spec calls *agentic workflows* [10]. With this, your MCP server doesn't have to passively wait for the user to invoke the next slash command; it can proactively suggest or even trigger the next tool.

For example, you might implement a tool like `next_step_recommendation` that returns the next recommended command based on the stored state (phase, pending tasks, any "Gate" criteria). The assistant could call this at the end of each phase, effectively asking "what should I do next?", and your server would respond with something like: *"Proceed to P5: run /e2e-runner-setup to set up end-to-end testing"*. Alternatively, the server can be configured to push a notification to the client as soon as a phase's exit criteria are met. Because the MCP server can maintain a *live context* and even initiate calls, it serves as the **planner/foreman** in the workflow: tracking progress, updating the state store after each tool action, and guiding the AI on what tool or prompt to invoke next.

By drawing on existing patterns – a prompts library server for the base commands [1], a context-tracking layer for project state [6], diagram-driven logic for sequencing [9], and MCP's event capabilities for proactivity [10] – you can transform your docs-only playbook into an intelligent MCP server. Such a server will not only respond to slash commands but also **act as a workflow orchestrator**, interpreting the project's design map (your Mermaid flowchart) and proactively advancing the development process step by step. This ensures the AI assistant always knows *where you are* in the project and what to do next, essentially becoming a hands-on project co-pilot rather than just a passive text generator.

References and Examples

- *Prompts MCP Server* – Open-source MCP server that exposes Markdown prompt templates as tools [1] [2]. Helpful as a blueprint for wrapping a docs catalog into a running MCP service.
- *Project Context MCP Server* – Example MCP server that **maintains project state** (current phase, tasks, decisions) between sessions [6], illustrating how to persist and query workflow context.
- *MCP Mermaid Server* – Demonstrates tools for **interpreting and generating flowcharts**; e.g. it can analyze a Mermaid diagram's structure and suggest improvements [9], useful for building diagram-aware assistants.
- *MCP Protocol Discussions* – MCP is stateful and supports **server-initiated messages** (notifications, streaming) to enable agent-driven workflows [10]. This is the basis for making the assistant proactive in executing a sequence of steps.
- *MCP Architecture Guides* – Overviews of MCP server components and session orchestration, showing how multi-step actions can be handled internally [7] [8] (e.g. sequential tool calls managed by the server). These can guide the design of your "DAG-aware" planner logic within the server.

---

[1] [2] [3] [4] GitHub - tanker327/prompts-mcp-server: MCP server for managing and providing prompts with TypeScript, caching, and comprehensive testing
https://github.com/tanker327/prompts-mcp-server

[5] [7] [8] How MCP servers work: Components, logic, and architecture — WorkOS
https://workos.com/blog/how-mcp-servers-work

[6] MCP Project Context Server | MCP Ser... · LobeHub
https://lobehub.com/mcp/aaronfeingold-mcp-project-context

[9] MCP Mermaid Server | MCP Servers · LobeHub
https://lobehub.com/mcp/kayaozkur-mcp-server-mermaid

[10] State, and long-lived vs. short-lived connections · modelcontextprotocol modelcontextprotocol · Discussion #102 · GitHub
https://github.com/modelcontextprotocol/modelcontextprotocol/discussions/102