Project P5 Report: Multi-Agent Knowledge Navigator

Architecture:

1.1 System Overview

Project P5 integrates the capabilities developed across P1–P4 into a unified "Multi-Agent Knowledge Navigator" with a conversational interface in Open WebUI. Instead of treating each capability as a separate standalone program, P5 composes them into one assistant that can (1) answer questions using an internal document knowledge base, (2) retrieve up-to-date information via web search, (3) run verified terminal commands on the host machine, and (4) control on-screen UI actions using coordinate-based automation.

1.2 Components

(A) Open WebUI (Conversation + Orchestration Layer)

Open WebUI provides the chat interface and tool execution environment. It supports:

- Selecting a cloud LLM profile (P2 replacement),
- Creating and referencing Knowledge Bases (P3 replacement),
- Enabling built-in web search (for time-sensitive queries),
- Registering MCP servers (P1 + P4 tools) and exposing them as callable tools in chat.

(B) Cloud LLM (P2 Replacement)

Instead of a fine-tuned local model, P5 uses a powerful hosted model (e.g., gpt-oss:20b-cloud) configured via Open WebUI credentials. The model:

- Interprets user intent (question vs. action vs. retrieval),
- Chooses the correct capability (KB, web, terminal, UI automation),
- Converts natural language into tool-appropriate inputs (commands or action keys),
- Summarizes tool outputs in human-readable form.

(C) RAG Knowledge Base (P3 Replacement via Open WebUI)

The built-in Knowledge Base stores project PDFs, lecture notes, documentation, and any reference material required to answer questions accurately. Documents are uploaded via Workspace → Knowledge and can be referenced during chat using the #<KBNAME> syntax. When referenced, Open WebUI retrieves relevant text segments, injects them into context, and the model produces grounded responses based on those retrieved chunks. RAG is used for stable information contained in the uploaded documents (rubrics, project requirements, setup steps, and internal notes).

(D) Terminal Tool MCP Server (P1 as Tool)

P1 is implemented as an MCP server over Streamable HTTP. It exposes the terminal as a tool that Open WebUI can call. The key design requirement is that the terminal tool must maintain persistent state (directory changes, environment variables) across multiple user requests.

For P5, we expose three tools:

- initiateterminal() – starts a persistent bash session
- executecommand(cmd: str) – executes a bash command inside that same session
- terminateterminal() – closes the session cleanly

A marker-based output capture strategy is used to collect command output deterministically and to provide stable fields such as exit code and working directory after the command.

(E) UI Automation MCP Server (P4 as Tool)

P4 is implemented as a second MCP server exposing coordinate-based automation. It reads a JSON mapping file (e.g., coordinatemap.json) that assigns element names to screen coordinates. Two core tools are exposed:

- uiclick(elementname: str) – clicks at the coordinate mapped to the given key
- uitype(elementname: str, text: str) – clicks/focuses and types text at the mapped location

This approach avoids runtime OCR for the demo workflow and achieves low latency once coordinates are calibrated.

(F) Web Search (Built-in Open WebUI)

Web search is enabled in Open WebUI for information that may be outdated or time-sensitive (e.g., current weather, breaking news, rapidly changing APIs). When used, the model summarizes results and avoids guessing when verification is possible.

1.3 Communication and Message Flow

P5 uses Open WebUI as the hub, so all interactions are mediated through the conversation:

User → Open WebUI Chat: Natural language request.

LLM Decision: Determine whether the request is best answered by:

- RAG (if it depends on documents),
- Web search (if time-sensitive),
- Terminal tool (if needs verification/commands),
- UI automation (if user explicitly requests screen action).

Tool Invocation (if needed):

- For terminal: executecommand("...")
- For UI: uiclick("...") / uitype("...", "...")

Tool Result → LLM: Output returned as structured JSON + stdout content.

LLM → User: Summarized response, next steps, and any error explanations.

Performance: Latency Measurements and Optimizations

2.1 Measurement Method

Latency was measured at two levels:

Tool execution latency for terminal commands (P1): the server returns durationms for each executecomm and call.

User-perceived end-to-end latency: from user message to final assistant response, including tool calls.

For UI automation (P4), latency can be measured by:

Timestamped logs (if implemented) or

Video timestamps (tool call moment → observed click effect)

- executecommand("pwd"): fast; minimal output
- executecommand("ls -la"): depends on folder size
- uiclick("browsercloselasttab"): includes OS event + click delay
- RAG query with KB reference: retrieval + generation
- Web search question: depends on network/provider

2.2 Optimizations Applied

(1) Persistent terminal session (major improvement)

Instead of spawning a new shell per request, the terminal tool maintains a single bash process, allowing directory state (cd) and environment to persist. This reduces repeated process startup overhead and enables multi-step workflows across messages.

(2) Marker-based output capture (stability improvement)

A common failure mode for terminal tools is partial output capture or "hanging" reads, especially with interactive commands or uncertain termination. The server prints explicit markers after each command:

A "CWD marker" containing the post-command $PWD

A "DONE marker" containing $? (exit code)

The reader collects output until DONE is observed, guaranteeing deterministic completion and enabling the assistant to trust the returned state.

(3) Output truncation (reliability + UI stability)

Large outputs can cause tool/UI slowdowns, token explosion, or timeouts. The server limits total captured output by line count and character count. On the assistant side, commands are written with head -n ... where appropriate to reduce response size.

(4) Single-command chaining (&&)

Instead of multiple tool calls, the assistant uses compound commands: cd ~/Desktop && pwd && ls -la

This reduces tool overhead and decreases the probability of repeated "View Result…" loops in the UI.

(5) Coordinate automation (P4) for low-latency UI actions

Once coordinates are recorded, UI clicks/types are constant-time operations with minimal computation, enabling a fast demo loop.

Challenges & Limitations: Issues Encountered and Solutions

3.1 Inconsistent / "Garbage" Terminal Listings

Issue: Sometimes directory listings appeared incorrect or unrelated to the user's real filesystem.

Root Causes:

- The model sometimes answered without calling the tool (hallucination).
- Open WebUI could point to a different environment (e.g., Docker container filesystem) if the MCP URL was misconfigured (localhost inside container ≠ host machine).
- Multiple terminal tools enabled at once caused ambiguous tool routing.

Solutions:

- Enforced a strict system prompt rule: never list files unless tool stdout is present.

- Verified correct MCP endpoint configuration (use host.docker.internal when Open WebUI runs inside Docker on macOS).
- Disabled duplicate terminal tools so only the intended MCP server is used.

3.2 “cd does not persist”

Issue: User requested cd Desktop, but subsequent commands behaved as if still in the old directory.

Root Cause: Stateless execution (new process per call) or tool output not properly confirming state.

Solution: Persistent shell session + returning cwdafter ensures directory changes persist and are verifiable.

3.3 Repeated “View Result from execute_command” spam

Issue: The UI sometimes displayed repeated tool result entries or the model retried calls.

Root Causes:

- Empty stdout from commands like cd made the model “think nothing happened”.
- Streaming UI + tool event display behavior in Open WebUI.

Solutions:

- Server returns (ok) for success with empty stdout.
- Prefer chained commands that include pwd for confirmation.
- Reduce output size to avoid timeouts.

3.4 UI Automation Click Inaccuracy

Issue: Coordinates were correct but clicks sometimes missed the expected UI element.

Root Causes:

- Retina scaling / display scaling mismatch.
- Multi-monitor coordinate offsets.
- Window movement after capturing coordinates.

Solutions:

- Calibrate coordinates on the same display configuration used for demo.
- Keep the target window position consistent (maximized, fixed layout).
- Use a coordinate capture helper during setup.

3.5 Limitations

- Coordinate-based automation is not robust to UI layout changes; it requires re-calibration if resolution/scale/layout changes.
- Terminal tool is powerful and must be restricted to avoid destructive commands.
- Web search can fail due to network/provider; assistant must degrade gracefully without guessing.

Conclusion:

P5 successfully composes terminal execution (P1), cloud LLM reasoning (P2 replacement), built-in RAG knowledge management (P3 replacement), and coordinate-based UI automation (P4) into a single assistant in Open WebUI. The resulting system demonstrates a practical agentic workflow: it can retrieve knowledge from documents, verify facts via terminal commands, act on the desktop via UI automation, and use web search for current information. The persistent terminal session and marker-based output capture significantly improve reliability, while coordinate mapping enables low-latency UI control for demo requirements.