Local Agentic AI — Phase■1 (MVP)

Full Technical Documentation & Deployment Manual

Platform: Windows 11 + WSL2 (Ubuntu 24.04 LTS)

Prepared by: Senior Systems Engineering Team

Date: October 2025

Index of Contents

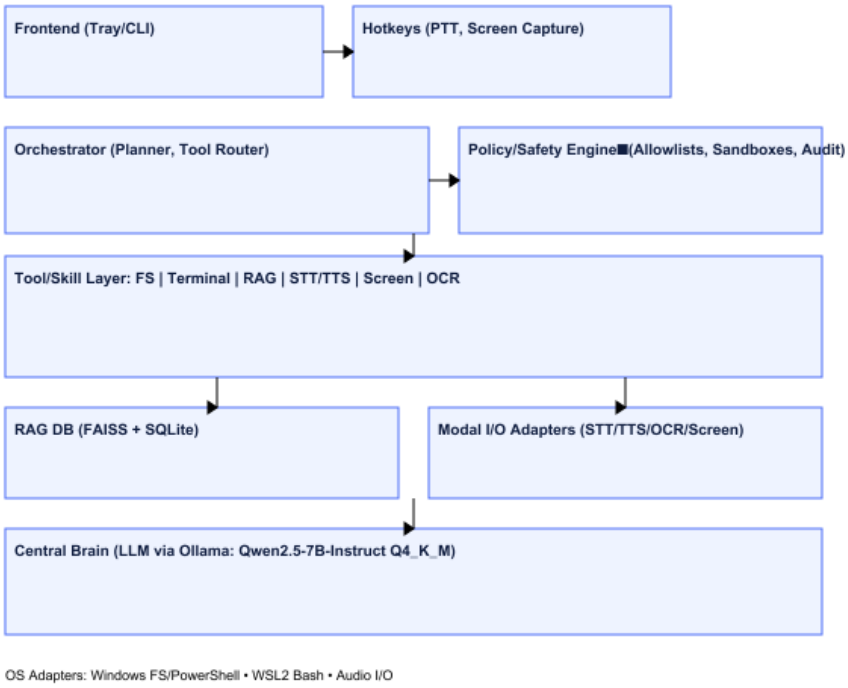
1. Project Definition & Objectives

The Local Agentic AI project is an offline■first, multimodal agent designed to operate entirely on a local Windows 11 system augmented by WSL2 (Ubuntu 24.04). Its purpose is to integrate voice recognition, natural language reasoning, terminal execution, and file/system awareness through a secure, sandboxed architecture. The MVP (Phase■1) focuses on establishing the foundational runtime pipeline — from voice input to intelligent local response — emphasizing security, auditability, and modular extensibility. It's targeted at developers, DevOps engineers, and researchers who require privacy■preserving, controllable AI agents for personal or enterprise environments.

The system's core achievement in this phase is the capability to receive spoken commands (push■to■talk), perform reasoning via a local LLM (Qwen2.5■7B■Instruct Q4_K_M), access and summarize local documents within user■approved sandboxes, and speak the summarized results. All tool interactions are recorded in detailed audit logs ensuring full transparency and traceability.

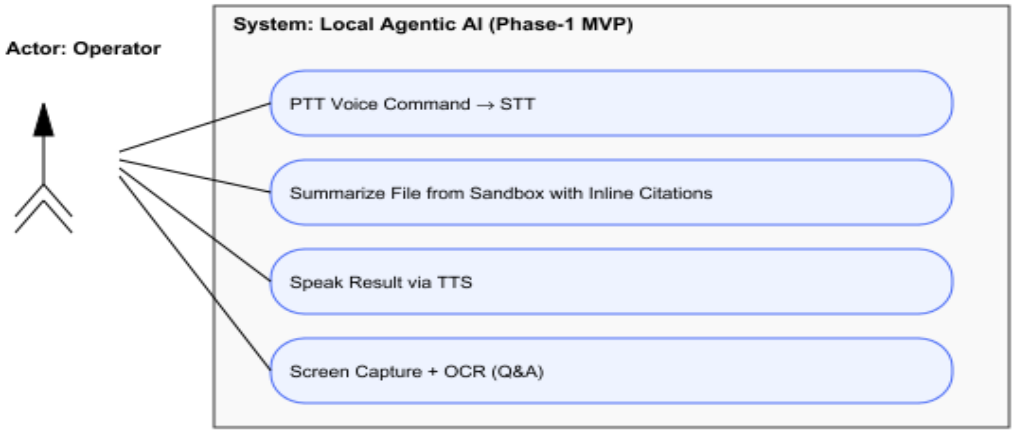
2. System Overview & Architecture

The architecture follows a layered model where an Orchestrator (planner) manages communication between the central LLM brain (running on Ollama) and tool adapters for specific modalities — file I/O, terminal commands, retrieval■augmented generation (RAG), audio, and visual inputs. Each layer is independently auditable, allowing granular policy enforcement. Below is the system's high■level architecture diagram.



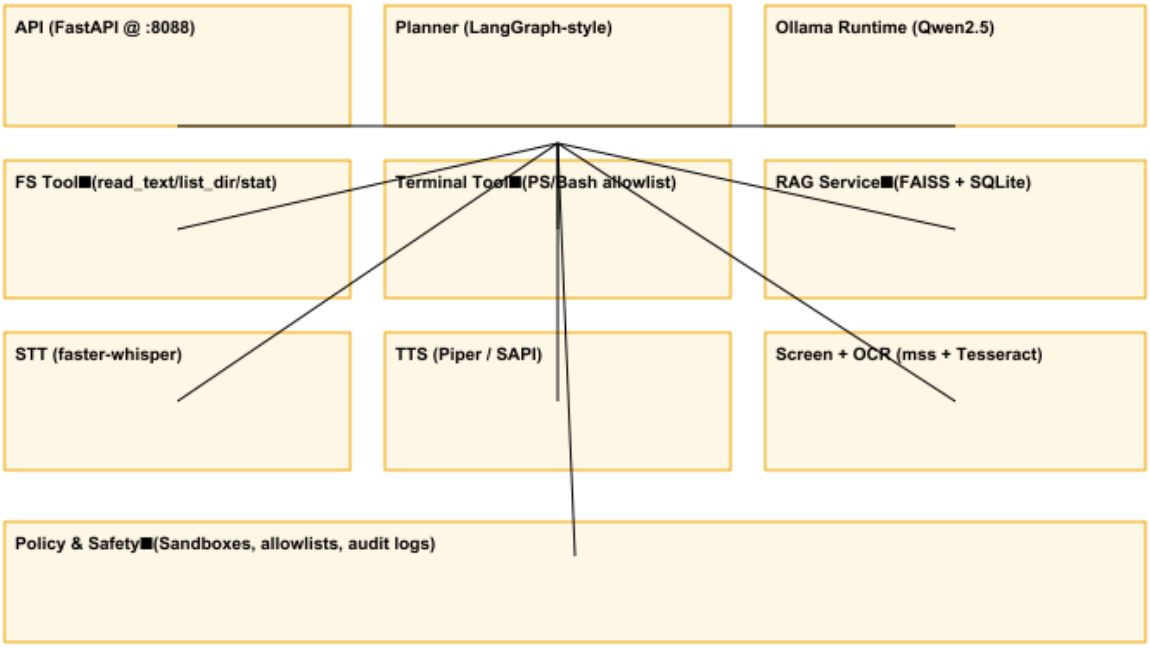
3. Use■Case Scenarios & Actors

Primary Actor: the local Operator. Secondary actors are system components that interact autonomously under predefined policies. The MVP covers the following principal interactions.



1. **Voice■Driven Summarization** — User issues a spoken command to summarize a local file. The agent transcribes, plans, reads, summarizes, and audibly returns the output. 2. **Screen Q&A;** — User captures screen content; OCR extracts text; the agent answers contextual queries. 3. **Allowlisted Terminal Operations** — Execute only safe, read■only commands (e.g., `git status`, `ls`, `Get■Content`) via policy enforcement.

4. Component Overview & Core Modules



Each component encapsulates a subsystem: • **API Layer (FastAPI)** — Serves as the REST interface for all tools and orchestrator actions. • **Planner (LangGraph■style)** — Handles intent recognition and tool invocation logic. • **Ollama Runtime (Qwen2.5■7B■Instruct)** — Central reasoning model with local inference. • **Tools** — Discrete modules implementing functions: filesystem, shell, RAG, STT/TTS, OCR, and screen capture. • **Policy Engine** — Governs safety: allowlists, sandbox boundaries, and audit trail creation.

5. Directory & File Structure Explanation

Below is a detailed breakdown of the directory hierarchy and the purpose of each core file or module in the codebase.

```
C:\Agent\
  ■■■ bin\
  ■   ■■■ agent.ps1       – Start/stop script orchestrating API & hotkeys.
  ■   ■■■ hotkeys.py      – Registers push■to■talk (Ctrl+Space) & screen capture hotkeys.
  ■   ■■■ reindex.py      – Manual RAG index rebuild utility.
  ■■■ config\
  ■   ■■■ config.example.json – Default configuration file.
  ■   ■■■ config.schema.json  – Validation schema for runtime configuration.
  ■   ■■■ terminal.allowlist.ps1.txt – Approved PowerShell commands.
  ■   ■■■ terminal.allowlist.bash.txt – Approved Bash commands.
  ■■■ models\           – Local storage for LLM, embeddings, and TTS voice models.
  ■■■ logs\             – JSON logs and temporary output (audio, screenshots).
  ■■■ audit\            – Immutable JSONL audit records per day.
  ■■■ ingest\           – Drop■folder for auto■indexing into the RAG vector store.
  ■■■ src\
  ■   ■■■ api\main.py           – FastAPI entrypoint.
  ■   ■■■ brain\runtime.py      – Ollama HTTP client for LLM inference.
  ■   ■■■ brain\planner.py      – Intent parser and flow router.
  ■   ■■■ common\logging.py     – Central structured logging/auditing.
  ■   ■■■ policy\safety.py      – Path normalization, traversal denial, allowlist validation.
  ■   ■■■ tools\*               – Independent modules (fs, shell, rag, stt, tts, ocr, screen).
  ■■■ tests\          – Unit & end■to■end verification scripts.
  ■■■ docs\           – Deployment and install documentation.
```

Each module is modularized for maintainability and clarity. The `src/tools` directory implements atomic tool functions so that each capability (e.g., OCR, STT) can be upgraded independently in future phases.

6. Security, Policy, and Safety Model

The system enforces least privilege by confining operations within user■specified sandboxes and disallowing any destructive terminal commands. Every function call is logged and audited with cryptographic hashes to ensure tamper evidence. Push■to■talk interaction replaces always■on listening, further minimizing exposure. All OCR, STT, and screen capture operations are gated behind explicit hotkeys.

7. Detailed Data & Control Flows

Flow 1: Voice Summarization → PTT triggers STT → Transcription parsed by planner → File read via FS module → LLM summarization → TTS audio output. **Flow 2: Screen OCR** → Capture active window → OCR text extraction → LLM contextual Q&A.; **Flow 3: Terminal Diagnostics** → Planner requests shell command → Policy verifies allowlist → Executes in subprocess → Returns stdout.

8. API Endpoints & Schemas

```
POST /chat – Route a free■form user message to the planner.
POST /tools/fs – Read or list files (operations: read_text | list_dir | stat).
POST /tools/shell – Execute allowlisted PowerShell or Bash commands.
POST /tools/rag/query – Query FAISS index for semantic document retrieval.
POST /tools/screen/capture – Capture full screen or region.
POST /tools/ocr – Perform OCR using Tesseract (eng+fas).
POST /tools/stt – Transcribe recorded audio via faster■whisper.
POST /tools/tts – Generate speech from text via Piper or SAPI.
```

9. Configuration Reference

The configuration JSON defines runtime parameters for model selection, safety, languages, and sandbox paths. Key parameters include context length, quantization level, terminal mode, and embedding model for RAG indexing. Configuration files reside under `C:\Agent\config`.

10. Deployment & Runtime Guide (0→100)

Follow this end■to■end procedure to deploy and run the Local Agentic AI MVP on a new Windows 11 system.

```
1. **Environment Setup**
   – Install Python 3.11 and create a virtual environment.
   – Install project dependencies (FastAPI, faiss■cpu, sentence■transformers, mss, pillow, pytesseract, sound

2. **OCR Engine**
   – Install Tesseract via Chocolatey and ensure `eng` and `fas` language packs.

3. **LLM Runtime**
   – Install Ollama and pull `qwen2.5:7b■instruct` (Q4_K_M quantization).

4. **TTS Setup**
   – Download Piper executable and a voice ONNX file; set `PIPER_VOICE` environment variable.

5. **Configuration**
   – Copy `config.example.json` to `config.json` and edit sandbox paths.

6. **Indexing**
   – Place documents in `ingest` or sandbox folders and run `bin/reindex.py`.

7. **Run Agent**
   – Start with `pwsh -File C:\Agent\bin\agent.ps1 start`.

8. **Operation**
   – Press Ctrl+Space to issue voice commands; Ctrl+Alt+S for screen OCR.

9. **Verification**
   – Check logs under `C:\Agent\logs` and audits under `C:\Agent\audit`.

10. **Shutdown**
   – Execute `pwsh -File C:\Agent\bin\agent.ps1 stop` for clean termination.
```

11. Operation & Monitoring

During runtime, all actions are visible in structured JSON logs. Operators can monitor sessions for errors, policy denials, or latency spikes. Audit logs are immutable daily JSONL files containing every executed tool call, its parameters, and results.

12. Testing, Validation & Quality Assurance

Automated tests validate sandboxing, policy enforcement, OCR accuracy, and retrieval integrity. End■to■end tests simulate real operator flows to ensure speech, summarization, and TTS output are coherent. Chaos scenarios cover missing GPU, corrupted indices, and denied permissions.

13. Performance Benchmarks & Tuning

Typical inference rate on GTX■960 (4 GB VRAM): 2–3 tokens/s for Qwen2.5■7B Q4_K_M. CPU fallback (i7■6700HQ) achieves ≈1 token/s. Whisper base.int8 transcribes 5 s audio in ≈1.2 s. Keep context ≤8 k to maintain stability with 12 GB RAM. For faster indexing, batch documents before reindex runs.

14. Risks, Mitigations, and Roadmap

Potential issues include VRAM exhaustion, audio device conflicts, OCR misreads, or sandbox misconfiguration. Mitigation strategies involve reducing GPU offload layers, explicit device enumeration, and nightly re■indexing. Phase■2 will introduce confirmed write■ops, hybrid RAG (BM25 + dense), improved context management, and wake■word activation.

15. Appendix A – Representative Code Snippets

```
# src/policy/safety.py
def check_ps_allowlist(cmd: str, allow: list[str]) -> bool:
    if _BAD.search(cmd): return False
    head = cmd.strip().split()[0].lower()
    return any(head.lower() == a.lower() for a in allow)
```

```
# src/brain/planner.py
if 'summarize' in text.lower():
    path = extract_path(text)
    if not path: return {'ok':False,'error':'no_path'}
    return summarize_file(flow, path, sandboxes, rag, llm, voice_model)
```

End of Document