SPAR Project Report: Secure Developer Infrastructure and Local Agent System

Kesav Nagendra
SPAR Program
kesavnagendra3@gmail.com

***Abstract*—This report summarizes my SPAR work on secure developer infrastructure and a local agent system for repository analysis. The first part of the work focused on a hardened developer workstation direction: image customization, package cleanup, kernel experiments, signing workflow considerations, and internal access debugging. The second part focused on building a local agent system that can answer repository questions, review source code, audit dependencies, and post results back into the existing internal development workflow. The current prototype works end to end from my local machine through comment-based commands such as `-vc ask`, `-vc review`, and `-vc dependency`. I also started moving the service onto an internal always-on compute server so it can eventually run as a shared service rather than depending on my laptop. The main remaining deployment blocker is the approved embedding setup for server-side retrieval.**

***Index Terms*—secure development, local agents, retrieval augmented generation, dependency audit, internal LLM infrastructure, software supply chain**

I. Introduction

My SPAR work developed along two connected paths. The first path was secure developer infrastructure. This involved working on a hardened workstation image, reducing unnecessary software, experimenting with kernel options, and testing internal image workflow ideas. The second path was the Local Agents project, which focused on using internal LLM infrastructure to support code review, repository search, dependency analysis, and off-hours engineering workflows.

The common theme across both paths was reducing unnecessary exposure in the development environment. The workstation work focused on reducing the surface area of the developer machine. The Local Agents work focused on keeping code review and repository analysis inside the internal environment rather than relying on external services.

The Local Agents system is now functional as a local prototype. It can receive a comment-style command, route the task, retrieve relevant context or scan dependencies, call the internal LLM API for generation, and post the result back into the internal Git discussion thread. The next major step is completing server-side deployment and expanding from a single test repository to organization-wide support.

II. Secure Developer Workstation and Image Infrastructure

A. Image Build Setup

The first part of the infrastructure work focused on a custom hardened workstation image. The image was managed through a declarative build recipe so package additions, package removals, and file changes could be tracked in a reproducible way.

The main goals were:

- build a usable hardened developer workstation image;
- reduce unnecessary packages and services;
- test local image build and switch-over workflows;
- understand how signing and internal automation could fit into the image pipeline;
- keep the workstation practical for day-to-day development.

I tested local image builds with container tooling and worked through issues around image tags, rootless and root image storage, local image transfer, and repeated rebuild behavior. This helped clarify what would be needed for a stable shared image workflow.

B. Package Cleanup and Surface Reduction

A major part of the workstation work was reviewing installed packages and deciding what should be removed from the image. The cleanup work covered browser-related packages, printing and scanning components, virtualization tools, extra input methods, extra desktop packages, large language packs, and support bundles.

I did not treat package cleanup as simply removing anything that looked optional. For packages that could affect the desktop or developer workflow, I checked whether they were required by other components before deciding whether they should stay.

Some package areas required more careful reasoning:

- SSH-related packages;
- graphical toolkit libraries;
- printing-related dependencies;
- input method components;
- core desktop dependencies.

This helped reduce the workstation attack surface while keeping the system usable.

C. Kernel Experiments

I also worked on kernel-related experiments for the workstation image. This included testing an alternative kernel path, verifying the active kernel after reboot, exploring long-term support kernel options, and understanding how kernel package selection interacts with the image build workflow.

The purpose was not just to switch kernels. It was to understand how controlled kernel sourcing and kernel selection could fit into a secure developer workstation pipeline. Kernel selection affects trust, maintainability, hardware support, and the update story for a hardened workstation image.

D. Image Signing and Internal Build Workflow

I worked through signing-related questions for the image workflow. One important point was that private signing keys should not be committed to the repository. They should be stored as protected internal secrets, while public verification material can be kept in the repository where appropriate.

This connects to the longer-term direction of using internal automation to build, sign, and publish workstation images. I also worked through local Git setup, repository pushes, and the process of moving local image experiments toward shared internal infrastructure.

E. Internal Access and Network Debugging

A recurring part of the infrastructure work was debugging internal access. I worked through issues involving multiple VPN connections, internal DNS resolution, internal repository and registry access, authentication redirects, and direct IP testing when hostnames were not resolving correctly.

One issue I found was that both VPNs could appear connected, but the system could still use the wrong DNS server. That caused some internal names to fail even though the network connection itself was active. I used tools such as `nslookup`, `curl`, `ping`, `tracert`, and direct IP checks to separate DNS issues from routing issues.

This debugging was important because both the workstation image work and the Local Agents project depend on reliable access to internal services.

III. Local Agents Project

A. Goal

The Local Agents project was designed to make internal LLM infrastructure useful for development workflows. The system is intended to read local code and documentation, retrieve relevant context, answer repository questions, review source code, audit dependencies, post results back into the normal development workflow, and eventually run scheduled off-hours jobs.

The system was designed to stay inside the internal environment. It does not depend on external model APIs. The main user interaction model is through comments in the internal Git system, so developers can ask the agent to do useful work without opening a separate interface.

IV. Original Proposed Architecture

The original design used local data sources, preprocessing, retrieval, an agent controller, local LLM generation, and an output layer. This design guided the implementation, although some pieces were simplified to keep the prototype inspectable and practical.

V. Current Implemented Architecture

The current implementation follows the same high-level structure, but now the pieces are concrete and tested. It includes a local HTTP service, command routing, retrieval, dependency auditing, generation through the internal LLM API, and comment posting.

The current local prototype supports the full flow from command to result:

1) A user posts a `-vc` command.
2) The service detects the command.
3) The correct agent mode runs.
4) The system retrieves or scans context.
5) The internal LLM generates the answer, review, or summary.
6) The result is posted back into the Git discussion thread.

VI. Tooling Decisions

A. Agent Scaffold

I chose to build a lightweight custom Python agent loop instead of depending directly on a larger existing agent framework. I considered existing tools and frameworks, but did not use them directly for the first version because this project has a strong security and dependency-minimization constraint.

A custom loop was enough for the current needs: detect the task type, retrieve context, build a prompt, call the internal LLM, format output, save logs, and post results back into the internal Git system. This kept the implementation easier to inspect and reduced additional software dependencies.

B. Retrieval and Vector Database

I used FAISS as the vector index. FAISS was a good fit because it runs locally, can run on CPU, is simple to integrate with Python, does not require GPU memory, and is enough for the current repository-scale prototype.

Keeping FAISS on CPU is useful because GPU memory is better reserved for the LLM serving stack.

C. Model Usage

The current system separates embedding from generation. For generation, the system uses the internal LLM API for repository question answering, code review generation, dependency audit summary generation, and general response formatting.

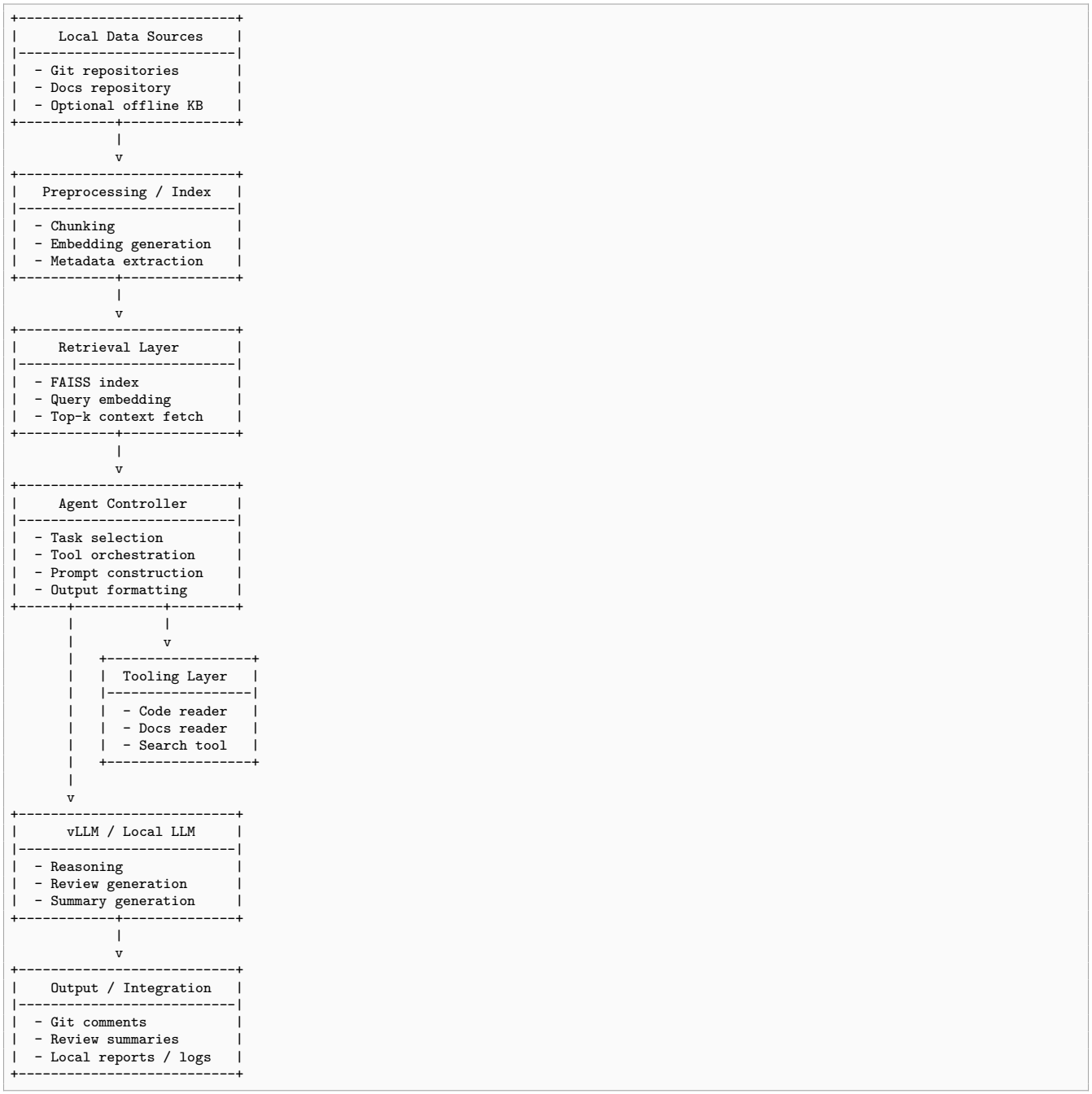

Fig. 1: Original proposed Local Agents architecture.

For embeddings, the local prototype uses a local embedding model. The embedding model is used to create query embeddings and search the FAISS index. One deployment issue remains: the internal LLM API currently supports generation but does not expose an embeddings endpoint. The deployment version therefore needs an approved internal embedding setup or an approved embedding service/image.

VII. Repository Indexing

I used a large editor-style repository as the main test case because it contains both documentation and Rust code. This made it useful for testing repository understanding, documentation Q&A, and source-code review.

The indexing pipeline walks through selected documentation and source-code files, splits files into chunks, records metadata for each chunk, generates embeddings, saves chunk embeddings, builds a FAISS index, and saves

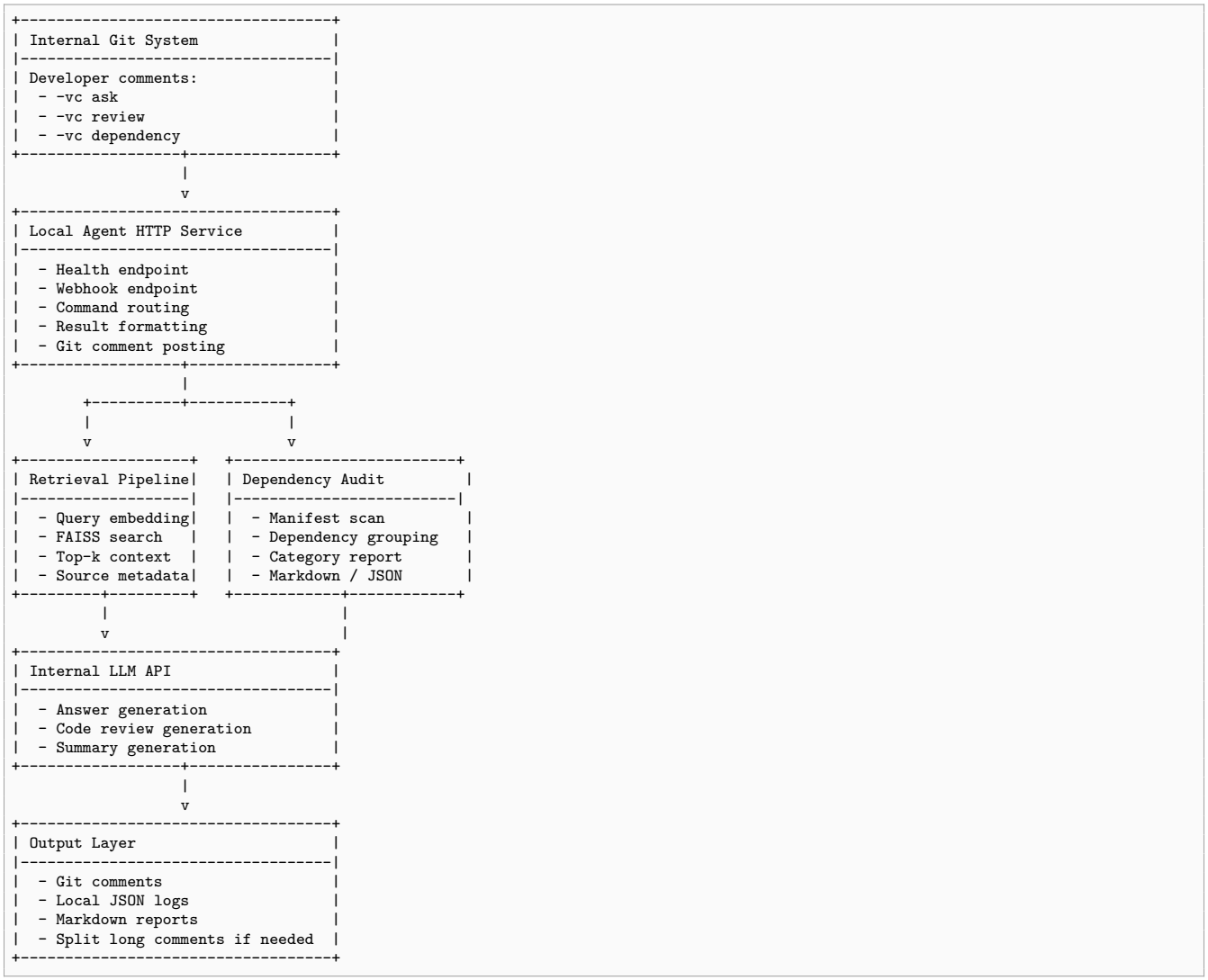

Fig. 2: Current implemented Local Agents architecture.

metadata next to the index.

Metadata includes chunk ID, file path, file name, chunk type, start line, end line, and text content. This metadata is important because the agent can cite repo-relative source paths and line ranges in the output.

A. Embedding Pipeline Improvements

The embedding process was initially slow because it was running locally and the repository was large. To make it practical, I updated the embedding script to support checkpointing and resume behavior. This meant the embedding job could continue from existing progress instead of starting over after an interruption or failure.

The final local run completed almost the full repository index:

- total chunks: 4618;
- embedded chunks: 4606;
- failed chunks: 12;
- code chunks in final index: 2622;
- documentation chunks in final index: 1984.

After rebuilding the FAISS index, retrieval quality improved significantly.

VIII. Agent Modes

A. Question Answering

The question-answering mode answers repository or documentation questions. A typical command is:

```
-vc ask How does the project handle diagnostics?
```

The question is embedded, relevant documentation or code chunks are retrieved, context is formatted with file paths and line ranges, the internal LLM generates a source-grounded answer, and the answer is posted back into the discussion thread.

B. Code Review

The code-review mode reviews retrieved source code and identifies possible risks, bugs, edge cases, or maintainability concerns. A typical command is:

```
-vc review editor
```

One issue I fixed was that code-review prompts sometimes retrieved documentation instead of source code. I changed the retrieval logic so code-review tasks retrieve a larger candidate set and then prefer chunks marked as code. This improved review quality because the model was given source-code context instead of documentation-only context.

The review output includes a short summary, potential issues, suggested next steps, and repo-relative source references. I also removed local machine paths from posted comments so the output is useful to other people.

C. Dependency Audit

The dependency-audit mode was added to support the dependency-minimization requirement. The audit scans dependency-related files and classifies dependencies into Specialized, Direct, Unnecessary, Test-only, and Internal Workspace categories.

The first three categories map to the dependency-minimization requirement. The two additional categories make the report more useful by separating internal workspace components and test-only dependencies from external runtime dependencies.

For the test repository, the audit produced:

- dependency files scanned: 245;
- total dependency entries: 844;
- unique dependencies: 260;
- Internal Workspace: 89;
- Specialized: 22;
- Direct: 138;
- Unnecessary: 6;
- Test-only: 5.

The output is saved as JSON and Markdown, and a summary can be posted back into the Git discussion thread.

IX. Dependency-Minimization Feature

An additional project requirement was to support dependency minimization. The goal is to reduce software supply-chain attack surface by identifying dependencies that can be reimplemented, consolidated, or removed.

The dependency-audit mode is the first step toward that. The current audit does not automatically rewrite or replace dependencies. Instead, it produces an inventory and classification so replacement work can be reviewed carefully.

The audit supports the requirement by:

- enumerating dependency files;
- separating internal workspace dependencies from external dependencies;
- identifying frequently used utility dependencies as Specialized candidates;
- identifying external direct dependencies for review;
- flagging small limited-use dependencies as possible Unnecessary candidates;
- separating test-only dependencies from runtime dependencies;
- generating Markdown and JSON reports for review.

This matters because dependency replacement must be done carefully. Reimplemented libraries need compatibility tests, architecture support, security review, and documentation.

X. Git Integration and User Interface

I used the internal Git system as the main user interface. Developers already work there, and adding a separate web UI would increase complexity for the first version.

The service exposes a webhook endpoint and supports the following commands:

```
-vc ask <question>
-vc review
-vc dependency
-vc audit
-vc
```

The command behavior is:

- `-vc ask <question>` runs question-answering mode;
- `-vc review` runs code-review mode;
- `-vc dependency` runs dependency-audit mode;
- `-vc audit` also runs dependency-audit mode;
- `-vc` defaults to code review.

I tested the complete local flow using a test repository and a test issue. The service successfully posted generated results back into the Git discussion thread.

A. Output Improvements

During testing, I made several output improvements. The output keeps repo-relative source paths, includes source line references, splits long comments so useful output is not cut off, keeps JSON logs locally for debugging, and keeps Markdown reports for dependency audits.

XI. Repo Mapping and Organization-Wide Direction

For a single local test, it was enough to manually pass the repository path. For real usage, that is not enough. A real webhook sends a repository full name, not a local filesystem path. To support that, I added repo mapping.

The basic idea is:

```
repository.full_name
        |
        v
repo mapping
        |
        v
local clone path + index path
```

For organization-wide use, the final design should support many repositories using a layout like:

```
local-agent/
  repos/
    ORG/
      XXX_repo/
      YYY_repo/
  indexes/
    ORG__XXX_repo/
    ORG__YYY_repo/
  outputs/
    agent_runs/
    dependency_audits/
```

This prevents one repository from accidentally using another repository's index. The proper final behavior is to read `repository.full_name` from the webhook, map it to a local clone, load that repository's own FAISS index, run the requested command, and post the result back to the same issue or pull request.

XII. Deployment Work on Internal Compute Server

After the local version worked, I started moving the service onto an internal always-on compute server. The reason was to avoid depending on my laptop and to reduce the local load from embedding and indexing work.

On the internal server, I confirmed that SSH access works, Git is available, Podman is available, the internal LLM API is reachable, and Python is not installed directly on the host. Because Python is not installed directly, I chose to deploy the agent service as a Podman container. This avoids modifying the host and keeps the deployment cleaner.

A. Container Build

I copied the agent scripts to the internal server and created a container image for the service. The first container build failed because the container could not reach external package mirrors during an `apt-get` step. Since the service did not need those extra packages inside the container for the first deployment, I removed that step.

After that, the container image built successfully with the needed Python dependencies:

- `requests`;
- `numpy`;
- `faiss-cpu`.

This confirmed that the service can run inside a container on the internal server.

B. Remaining Deployment Blocker

The main blocker for completing the internal-server deployment is embeddings. The internal LLM API works for generation, but it does not expose an embeddings endpoint. I also tried running an embedding service container, but pulling the external image was blocked from the internal server.

The next deployment decision is how embeddings should be handled internally. Possible options are to use an approved internal embedding service, use an approved internal image for the embedding service, run a separate embedding model through an approved internal service, or rebuild indexes with whichever embedding model is approved.

This needs to be resolved before question-answering and code-review modes can run fully from the internal server. Dependency audit can run without embeddings, but retrieval-based modes need query embeddings.

XIII. Current Status

The system is working end to end from my local machine. Completed work includes:

- secure workstation image experimentation;
- package cleanup and surface reduction;
- kernel experiments;
- internal access and DNS debugging;
- local repository chunking;
- embedding pipeline with checkpoint/resume support;
- FAISS index over documentation and source code;
- question-answering mode;
- code-review mode;
- dependency-audit mode;
- internal LLM generation integration;
- HTTP service;
- Git comment command interface;
- Git token setup and comment posting;
- repo mapping;
- removal of local paths from public output;
- long comment splitting;
- initial container deployment on an internal server.

Still in progress:

- approved embedding setup on the internal server;
- repo-specific indexes for multiple repositories;
- organization-level webhook;
- automatic local repo update before analysis;
- off-hours scheduled execution;
- final deployment documentation.

XIV. Remaining Work

A. Resolve Embeddings on the Internal Server

The agent needs query embeddings for retrieval. The query embedding model must match the model used to build the FAISS index. Once an approved embedding setup is available, I can complete the internal-server deployment for `-vc ask`, `-vc review`, and repo-specific retrieval.

B. Add Repo-Specific Indexes

The current full index is for the test repository used during development. For organization-wide usage, each supported repository should have its own index.

```
local-agent/
  repos/
    ORG/
      XXX_repo/
  indexes/
    ORG__XXX_repo/
```

Each repository should be indexed independently.

C. Add Organization-Level Webhook

Once the service is running from the internal server and repo-specific indexes are ready, the webhook can be configured at the organization level. The target flow is that a developer comments in any supported repo, the organization webhook sends the event to the agent service, the agent reads the repo name, loads the correct repo and index, runs the requested mode, and posts the result back to the same thread.

D. Add Local Repo Update Support

Before running analysis, the agent should update the local clone safely using a read/update sequence such as:

```
git fetch
git pull --ff-only
```

This keeps the local analysis copy current without introducing unsafe repository changes.

E. Add Off-Hours Runner

The original project goal included off-hours usage. The planned off-hours runner would read configured repositories, run dependency audits, run queued review or summary jobs, save Markdown and JSON reports, and optionally post summaries into the internal Git system. This can later run through a systemd timer, cron-like schedule, or another approved scheduler.

F. Final Documentation

The final handoff documentation should explain how the system is structured, how to run it locally, how to deploy it on the internal server, how to configure tokens, how to add a new repository, how to build a repository index, how to use `-vc` commands, known limitations, and next steps.

XV. Final Target Architecture

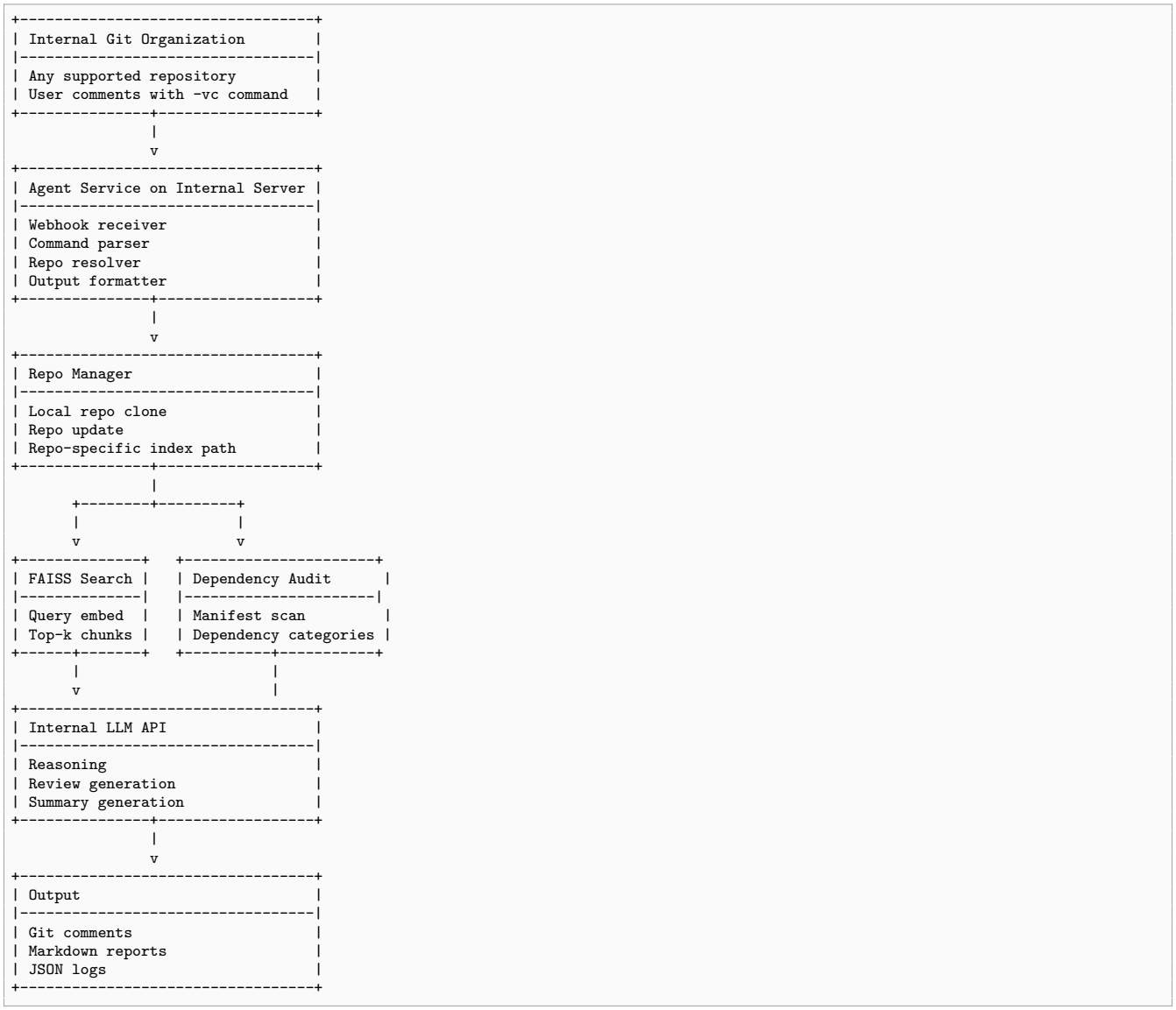

Fig. 3: Final target architecture for organization-wide deployment.

XVI. What I Learned

This project helped me understand the practical difference between a prototype and a usable internal service. The agent logic itself was only one part of the work. A lot of the real difficulty was in the surrounding infrastructure: internal networking, DNS behavior, token permissions, repository access, container deployment, embedding availability, output formatting, and avoiding local-machine assumptions.

I also learned that dependency minimization affects design choices. Using a large agent framework might have been faster, but it would have added more software dependencies. A smaller custom loop was easier to review and better aligned with the project constraints.

The dependency-audit feature also made the supply-chain part more concrete. Instead of only discussing dependency reduction, the tool now creates an initial inventory and classification that can support future replacement work.

XVII. Conclusion

During SPAR, I worked on both secure developer infrastructure and local agent tooling. On the infrastructure side, I worked on a hardened developer workstation direction through image customization, package cleanup, kernel experiments, image workflow debugging, signing direction, and internal access troubleshooting.

On the Local Agents side, I built a working retrieval and agent system that can answer repository questions, review code, audit dependencies, and post results back into the internal Git workflow. I also started moving the system onto an internal always-on server so it can eventually run as a shared service rather than from my laptop.

The current version is a working local prototype with Git comment integration. The next major step is completing the internal-server deployment by resolving the embedding setup, then expanding from one test repository to organization-wide support across multiple repositories.

References

[1] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *Advances in Neural Information Processing Systems*, 2020.

[2] J. Johnson, M. Douze, and H. Jegou, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535-547, 2019.

[3] W. Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention," *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, 2023.

[4] Open Container Initiative, "Open Container Initiative Image Format Specification," technical specification.

[5] The Git Project, "Git Documentation," software documentation.

[6] Sigstore Project, "Cosign Documentation," software supply-chain signing documentation.

[7] Microsoft, "Language Server Protocol Specification," software protocol documentation.

[8] Kiwix Project, "Kiwix Documentation," offline knowledge-base documentation.