ChatGPT

Layer 4 – Code Ingestion (Audit Findings & Plan)

Identified Layer-4 Components (WIP)

During the audit, two unpublished script files were found related to “Layer 4: Code Ingestion.” These appear as work-in-progress stubs not yet merged into the main branch. No production code or CLI tools for code ingestion exist on `main` – the functionality is effectively paused in a stash/local state. The table below lists the WIP components, their intended phase roles, and where they currently reside:

Component (File / Tool)	**Intended Layer-4 Phase**	**Current Status & Location**
`scripts/ingest/stub_ingest.py`	Phase 1 – *Code Discovery*	**Uncommitted WIP** (only in local working copy). This stub script was created as a harness for code ingestion but contains no logic yet [1]. Not present on any remote branch.
`scripts/ingest/mappers.py`	Phase 2 – *Fragmentation* Phase 3 – *Classification*	**Uncommitted WIP** (local only). Placeholder module for code parsing and tagging. No functions implemented; it’s a skeleton to hold code fragmentation and classification logic [1].
Code Ingestion CLI/ Make Tool	Phase 1 – *Discovery interface*	**Not present.** *No* user-facing command or Make target exists for code ingestion. (E.g. there is no `pmagent` subcommand or `make code.ingest` – the feature isn’t wired into any tool on `main`.)

Evidence: These files appear in the audit as new/untracked additions (with only the standard header) [1]. The absence of any “code ingest” command in the help menus confirms no integrated tooling yet on main. Phase 4 also does not appear in the completed phase list of the Master Plan [2], indicating it was left unfinished.

Phase Breakdown and Implementation Status

Phase 1 – Code Discovery: *Goal:* identify and catalog relevant code assets (e.g. scanning repository for source files/functions). **Status: Not implemented.** The `stub_ingest.py` file suggests intent to create a discovery harness, but currently it’s essentially empty [1]. No mechanism runs through the codebase to build a “code registry” yet. In short, the discovery phase exists only as a named stub, with no actual file-scanning logic or data structures in place.

Phase 2 – Fragmentation: *Goal:* break code into ingestible fragments (e.g. functions, classes, docstrings). **Status: Missing.** There is no code yet to perform fragmentation of code files. The `mappers.py` stub is presumably where such logic would live [1], but it currently contains no functions or code. Thus, no fragments are being generated or stored. (Notably, the existing document fragment model in the control-plane could be extended for code – see overlap discussion below – but this hasn't been done.)

Phase 3 – Classification: *Goal:* label/categorize code fragments (e.g. type of fragment, subsystem, or other taxonomy). **Status: Missing.** No classification or tagging functionality for code has been written. We expect this would be implemented in the same `mappers.py` module or a similar place, but as of now it's just a plan. There are no classification rules or data outputs yet for code fragments.

Additional phases (embedding/indexing): In analogy to document ingestion, a phase to embed code fragments into a vector index for search may be needed (let's call it Phase 4 of code ingestion). **Status:** not started. There is no code to vectorize or store code semantics. This would need to follow once fragments exist, using a similar approach as the docs RAG ingestion (storing embeddings in `control.doc_embedding`, etc.), but nothing is implemented yet.

Overall, **Layer 4 is essentially at the "scaffolding" stage** – the files and placeholders exist, but none of the core functionality (discovery, split, classify, etc.) has been developed. This aligns with the Master Plan's omission of a completed Phase 4; the work was postponed and never integrated into main [2].

Overlap/Duplication with Layer 3 (Patterns)

Layer 3 involved **pattern mining and document ingestion**, which introduced infrastructure that Layer 4 should leverage rather than duplicate:

- **Fragmentation logic:** The control-plane already has a `control.doc_fragment` table and an `upsert_fragment()` utility for storing document fragments (developed for proof-of-read and docs ingestion) [3] [4]. Rather than creating a parallel mechanism, code ingestion should reuse or extend this schema for code fragments. The audit notes this in context of docs: *"We already have* `control.doc_fragment` *... This can be reused... to avoid duplicating fragment logic."* [5]. In other words, Layer 4 should piggyback on the existing fragment model (perhaps by adding a fragment type like `"code"` or linking to a code registry) instead of reinventing a new store for code pieces.

- **Classification patterns:** Any categorization schemes from pattern analytics (Layer 3) could inform code classification. For example, pattern mining may have categorized patterns by type or cluster; similarly, code fragments might be classified by type (function, test, config, etc.). If Layer 3 introduced taxonomy or clustering methods, those could be adapted for code. However, since pattern mining dealt with **numerical/textual patterns** in data rather than code structure, direct overlap is limited. The main duplication risk is in the chunking and storing of content – which, as noted, should be unified with the existing approach for docs/patterns rather than done separately.

- **Infrastructure:** Layer 3 established a pipeline for ingestion (discovery → fragmentation → embedding of documents). Layer 4's pipeline is conceptually similar, so it should follow the established conventions (hermetic CI mode, use of pgvector for embeddings, etc.) rather than creating a wholly separate process. For instance, the plan for doc content ingestion (Phase 8C) set out how to chunk text and store embeddings in Postgres [6] [7]. A code ingestion pipeline would mirror that, preventing duplicate solutions for things like chunk storage or vector search.

In summary, the only potential duplication identified is **structural**: the fragmenting and storing of code could duplicate what was done for docs/pattern patterns if done naively. The remedy is to integrate with the **existing Layer 3 ingestion framework** (doc_registry, doc_fragment, etc.) rather than parallel it. Fortunately, since Layer 4 hasn't progressed, no actual duplicate implementation exists yet – we have the opportunity to design it correctly from the outset.

Phase Completion Status & Promotion Readiness

Summary of Layer 4 status: No Layer 4 phase is fully implemented at this time – all are either absent or just skeletal. Phase 1 (discovery) has a stub but no functionality; Phases 2 and 3 (fragmentation & classification) are completely unimplemented stubs; any further phases (embedding, integration) have not been started. This means **the entire Code Ingestion feature is essentially inactive** – nothing from it affects the current system, and there are no failing tests or broken integrations related to it (since it's not wired in).

What can be promoted safely: The good news is that because the code is isolated and not in use, we can promote pieces of it incrementally without regressing existing functionality. The PM can create a focused feature branch to begin **Phase 1: Code Discovery** implementation without impacting `main`. For example, we recommend a branch named `layer4-phase1-code-discovery` to develop the first step in isolation. On this branch, the team can implement the code scanning logic (e.g. traverse the repository for `.py` files, perhaps populate a new `control.code_registry` or reuse `control.doc_registry` for code) and ensure it runs hermetically (no external side effects, respecting CI guards).

Once Phase 1 is working and passes all checks (no duplicate entries, no broken rules), it can be merged into main. At that point Layer 4 would have an initial foothold in the codebase (e.g. a command or script to list/record code files). Subsequent phases – fragmentation (Phase 2) and classification (Phase 3) – can then be developed in their own feature branches (e.g. `layer4-phase2-fragmentation`, etc.), building on the Phase 1 foundation.

Minimal promotion strategy: Start with **Phase 1 on a clean branch** (`layer4-phase1-code-discovery`) implementing just the discovery logic and any minimal schema changes (if needed) [6]. Keep it read-only and observational at first (for example, generate a report of code modules or a JSON registry in `share/` for PM to review) so it doesn't disrupt anything. Once that is stable, incrementally introduce Phase 2 on another branch, reusing the doc fragment mechanism to store code fragments (ensuring no interference with existing doc fragments) [5]. Each phase should be merged only after its tests/guards pass in HINT and STRICT modes. By staging the promotion in small, self-contained PRs (one phase at a time), the Layer 4 features can be integrated cleanly into the mainline.

In conclusion, **Layer 4 (Code Ingestion) is largely a blank slate** – the groundwork exists in name only. We have identified the placeholder files and confirmed they're confined to a local WIP state [1]. None of the phases have substantial code yet, which means we can proceed to implement them methodically. The first step is to resume Phase 1 (code discovery) on a dedicated branch and get that merged, establishing the pattern for subsequent phases. Adhering to the patterns from Layer 3 (for fragmentation and storage) will ensure consistency and minimize duplication as we bring Layer 4 from stash into a fully tracked, functioning part of the system.

Sources:

- Audit evidence of Layer 4 WIP files [1]
- Doc content ingestion plan (Phase 8C) – fragment/embedding strategy to emulate [5] [8]
- Master Plan excerpt (Phase 4 not completed in mainline) [2]

---

[1] pm_system_introspection_evidence.md
https://github.com/iog-creator/Gemantria/blob/d01041033f6cf822ccafbd62ad24258e4578a111/share/pm_system_introspection_evidence.md

[2] MASTER_PLAN.md
https://github.com/iog-creator/Gemantria/blob/d01041033f6cf822ccafbd62ad24258e4578a111/share/MASTER_PLAN.md

[3] [4] doc_fragments.py
https://github.com/iog-creator/Gemantria/blob/d01041033f6cf822ccafbd62ad24258e4578a111/agentpm/control_plane/doc_fragments.py

[5] [6] [7] [8] DOC_CONTENT_VECTOR_PLAN.md
https://github.com/iog-creator/Gemantria/blob/d01041033f6cf822ccafbd62ad24258e4578a111/docs/SSOT/DOC_CONTENT_VECTOR_PLAN.md