ChatGPT

Gemantria Project Master Reference

Project Overview

The Gemantria project is a comprehensive system for analyzing Biblical texts through **gematria** (the numerological values of Hebrew words) combined with modern AI-driven insights. Its core goal is to build a **deterministic, resumable LangGraph pipeline** that produces verified gematria data and visualization-ready artifacts. In practice, this means leveraging both structured Biblical text data and LLM-based analysis in a governed pipeline. The system prioritizes **correctness and reproducibility** – strict rules ensure that code and database facts take precedence over AI outputs. Every stage's output is treated as a **Single Source of Truth (SSOT)** for the next stage, conforming to a fixed JSON schema and never altered downstream. This design makes the pipeline deterministic and auditable: given the same input (and model versions), it will produce the same results, and any change in output can be traced to an upstream change. The project integrates domain-specific data (like gematria values of words) with semantic analysis. For example, it computes the gematria of each significant word and identifies connections between words that share numeric values or appear in similar contexts. All original scripture data is kept in a read-only Bible database, while derived results are stored in a separate application database to ensure safety and integrity of source texts. Overall, Gemantria's mission is to **uncover meaningful patterns in sacred texts without compromising accuracy or integrity**, treating the pipeline as a governed, enterprise-grade system.

Governance Model

Gemantria employs a rigorous governance model to ensure that all contributions and pipeline executions adhere to project rules and quality standards. The **"4-source" framework** is used for automated governance agents (like the CI assistant "Cursor"), meaning the agent's instructions are derived from four authoritative sources: the **AGENTS.md** governance document, the **RULES_INDEX.md** (index of all internal rules), and two critical rule files (Rule-050 and Rule-051). Together these define the operating policies that the automation must follow. Enforcement policies are **fail-safe ("fail-closed")**: if something is off (e.g. a required file is missing, or not enough data), the process halts rather than proceeding with partial or unverified outputs. For example, the pipeline will abort if fewer than 50 nouns are extracted from a book (unless an override flag is explicitly set for a test). All critical files and operations are verified before use – any missing artifact triggers a loud failure instead of silent creation. This guarantees that errors or oversights surface immediately rather than corrupting downstream data.

Rules and Scope: Project policies are formalized as a series of numbered rules (see **Rules Inventory** below). These cover everything from data integrity and pipeline stages to code style and CI process. Certain rules act as an **"authority chain"** for automated operations – for instance, Rule-039 (execution contract), Rule-041 (merge policy), Rule-046 (hermetic file handling), Rule-050 (ops agent format), Rule-051 (PR checks advisory), and Rule-052 (tool priority) govern how the CI assistant behaves. Each rule has a defined scope (some apply to pipeline data, others to PR process, CI, security, etc.), and they are enforced by both automated guards and human reviewers. For example, **merge gates** are in place to ensure code is integrated only when all policy conditions are met. The **PR merge policy** (Rule-041) requires that all required CI checks pass, a conventional commit message and signed commit are present (policy gate), and at least one human approval is given before a PR can be merged. Even though branch protection may be relaxed for automation, the project's internal guardrails demand those conditions for every merge. Automated PR reviews (from bots) are considered **advisory only** once the required checks are green – the bot will explicitly state that its feedback is non-blocking per Rule-051. This ensures that human judgment remains in the loop for final approval, upholding the governance standards.

HINTs Envelopes: To facilitate governance and traceability, the system emits **HINTs** – structured log messages – which are collected into an envelope at runtime. The pipeline's state carries a `hints` log, and just before export, a special node wraps these hints into a JSON **hints_envelope** (with a type, version, count, and list of hint items). This envelope is stored alongside other outputs (embedded in the final graph JSON) and is validated by the guard agent or CI to ensure it's well-formed. The HINTs envelope provides an audit trail of important events or decisions (for example, noting if a fallback was used, or if a partial run was allowed with a reason) without breaking the deterministic pipeline flow. It serves as an **enforcement bridge** (as per Rule-026) between the automated pipeline and the governance layer, allowing the CI or reviewers to verify that the pipeline declared any unusual conditions. In summary, the governance model is a blend of code and policy: a network of rules and automated checks (the "forest") guides every action, and any deviation must be explicitly surfaced (via HINTs or PR references) for human oversight.

Pipeline Architecture

Overview: Gemantria's pipeline is implemented as a **directed acyclic graph of stages** using the LangChain **LangGraph** framework for orchestration. Each stage of the pipeline is an "agent node" with a clear responsibility, and edges define the flow of data and conditions between stages. The entire pipeline (the graph of agents) is coordinated by a unified **orchestrator** script, which ensures the stages run in order with proper hand-offs and error handling. Thanks to LangGraph's design, the pipeline can be introspected and even visualized as a DAG, which makes the AI workflow transparent and debuggable rather than a black-box prompt chain. State is passed immutably from node to node in a shared `PipelineState` object, and a checkpointing mechanism records progress so that if the run is interrupted it can be **resumed deterministically** without losing data or repeating work. The orchestrator uses a **Postgres-backed Checkpointer** in production (with an in-memory option for testing) to snapshot state at key points. This ensures that even long multi-chapter jobs can be paused or recovered safely, which is critical for reproducibility and resumability.

Stages: The **Main Pipeline** comprises several stages that transform the data step by step (each stage producing a JSON output that becomes input for the next). The high-level flow is as follows:

1. **Noun Extraction** – Identify significant Hebrew nouns in the text and calculate their gematria values.
2. **Batch Validation** – Check that enough nouns were found (>= minimum batch size) or fail-closed if not.
3. **Enrichment** – Use an LLM to add theological context to each noun (annotations, references).
4. **Confidence Validation** – Verify quality/confidence of LLM output, ensuring it meets thresholds.
5. **Graph Building** – Create a network graph of nouns (nodes) and their relationships (edges).
6. **Graph Scoring** – Compute similarity metrics for edges and classify their strengths.
7. **Analysis** – Run graph analytics (clusters, centrality, patterns) and compile summary statistics.
8. **Guard Compliance** – Rigid QA check of all outputs (schema validation, invariants, rule checks).
9. **Release/Export** – Package the verified outputs and prepare final artifacts (for DB or UI).

Some of these stages are combined in implementation (for example, Graph Building and Scoring are part of a continuous process, and schema validation is applied at multiple points). In the current code pipeline (as documented in **AGENTS.md**), the orchestrated sequence of nodes is: **collect_nouns → validate_batch → enrichment → confidence_validator → network_aggregator → schema_validator → analysis_runner** . Together, these cover the flow above (with `schema_validator` and internal guards handling the compliance checks). The pipeline is **deterministic** – given the same input (and random seeds), it will follow the same path and produce identical outputs, which is vital for debugging gematria computations. It also enforces a "**fail-fast**" philosophy at each step: if an agent produces an unexpected result (e.g., too few nouns, or an edge referencing a non-existent node), that condition is detected and the pipeline stops rather than emitting bad data. This guarantees that downstream stages never operate on corrupt or partial data.

Orchestrator & Make Targets: All pipeline stages are wired together in `pipeline_orchestrator.py` (and the LangGraph `graph.py` definition) which serve as the **single entry point** for pipeline execution. The project provides convenient **Makefile targets** to run different parts of the pipeline. For example, `make orchestrator.full BOOK=Genesis` will execute the entire pipeline end-to-end on the book of Genesis. One can also run subsets: `make orchestrator.pipeline BOOK=<X>` runs only up through graph construction, or `make orchestrator.analysis OPERATION=all` runs just the analysis steps on existing graph data. There are targets to validate schemas (`make schema.validate`) and to run evaluation or smoke tests as well. The **Book Processing** workflow (`scripts/run_book.py`) allows orchestrating by chapter: `make book.plan` to plan a full book run, `make book.go` to execute it, `make book.stop N=<n>` for a stop-loss after N chapters, and `make book.resume` to continue an interrupted run. This capability is useful for large texts or iterative development. The pipeline's design favors modularity: each major step corresponds to an **idempotent make target** that can be run independently as long as its prerequisites (earlier stage outputs) are available. For instance, one can re-run just the graph scoring via `make graph.score` (which depends on `graph.build` having been run) or regenerate the analysis report via `make analytics.export` without re-extracting nouns. This modular design not only aids development and testing (one stage at a time) but also ensures that if a particular stage fails, a developer can fix the issue and resume from that stage's input rather than start over from scratch. The entire pipeline can also be visualized and stepped through using **LangGraph Studio**, a visual DAG debugger, which further helps in diagnosing issues by allowing inspection of state at each node in real time.

Agent Contracts (Stages and Responsibilities)

Each stage of the pipeline is implemented by a specialized **agent** (or function) with well-defined inputs and outputs. This section describes all agents, their responsibilities, and the **contract** they fulfill – including any prompt or logic they use, the files they read/write, and how they interact via Make targets.

Ingestion Agent (Text Sharding)

Purpose: Pre-process raw scripture text into manageable chunks ("shards") for analysis. (This stage is **optional** – it runs if the pipeline is provided plain text instead of using the database directly.)

Responsibilities: The Ingestion agent reads the full text of a book and splits it into logically coherent segments about ~1000 characters each, without cutting through verse boundaries. It produces a stream (e.g. JSONL) of shards, each with a reference to its source (book, chapter, verse range) and the text content. The agent ensures the chunking is **deterministic** – given the same text and settings, it will always produce the same shards with the same boundaries. Each shard record includes metadata like `"ref": "<Book X:Y-Z>"`, an offset or index, and the text snippet. Sharding is critical for very large texts so that downstream LLM-based agents can process input in pieces without exceeding context limits. In cases where the Bible is already stored in a structured database (as is true here), this step can be skipped and verses read directly from the DB; otherwise, `make ai.ingest` will run this agent to generate `shards.jsonl`.

Make Target & I/O: `make ai.ingest` – reads raw text (or DB verses) and outputs `shards.jsonl` in the `exports/` directory. Each line in `shards.jsonl` is a JSON object representing one shard of text. This agent doesn't use an LLM for heavy reasoning (just splitting text), but it still follows the contract of not introducing any extraneous data – it only outputs the structured shards.

AI Noun Discovery Agent (Noun Extraction)

Purpose: Identify significant nouns (names, places, key terms) in the input text and calculate their gematria values, forming the initial nodes of the concept graph.

Responsibilities: The Noun Extraction agent (AI-Noun Discovery) takes either the output of ingestion (text shards) or reads verses directly from the Bible DB, and uses an LLM prompt to find **important Hebrew nouns** in the text. It focuses on nouns with "clear lexical evidence of significance" – e.g., proper names or unique terms likely to be meaningful in analysis. For each identified noun, the agent gathers: the surface text of the word, its constituent letters (Hebrew characters), its **gematria value** (the sum of letter values), a classification (e.g. person, place, thing), and the references (source verses and positions) where it occurs. The agent is instructed to include a brief reasoning or comment for each noun in an `analysis` field for traceability, but to keep all factual fields strictly correct. Notably, the LLM is expected to actually perform or replicate the gematria calculation for each word and provide the numeric sum in the output. However, the project's philosophy is **"code gematria > bible_db > LLM"**. This means that while the LLM attempts the calculation, the system cross-verifies those values using deterministic code or known database rules to ensure accuracy. (If the LLM's output is inconsistent, a separate Math Verification function can recalc the sum of the Hebrew letters.) The agent must also obey a rule that **at least 50 nouns** should be found for a full book run (default batch size is 50) – if it finds fewer, it triggers a failure unless a partial run was explicitly allowed by setting `ALLOW_PARTIAL=1` with a rationale. This prevents proceeding on trivially small outputs. In essence, this stage creates the **initial node list** for the knowledge graph, anchored in the scripture text.

Output Contract: The Noun agent outputs a JSON file `exports/ai_nouns.json` that conforms exactly to the **AI Nouns schema** (`gemantria/ai-nouns.v1`). This JSON has a top-level object with fields: `"schema": "gemantria/ai-nouns.v1"`, `"book": "<Name>"`, `"generated_at": "<timestamp>"`, and `"nodes": [...]`. Each element in the nodes array is a noun object with the required fields: - **noun_id** (a UUID for the noun),
- **surface** (the word in Hebrew or transliteration),
- **letters** (array of the constituent Hebrew letters),
- **gematria** (integer gematria value),
- **class** (classification like person/place/thing),
- **sources[]** (list of occurrences with scripture reference and offset), and
- **analysis** (textual commentary or reasoning).

No fields may be added or omitted – the LLM or code populating this must follow the schema exactly. For example, a node might look like:

```
{
  "noun_id": "UUID1234",
  "surface": "In the beginning",
  "letters": ["ב", "ר", "א", "..."],
  "gematria": 2701,
  "class": "phrase",
  "sources": [ {"ref": "Genesis 1:1", "offset": 0} ],
  "analysis": "Appears as the opening phrase of Genesis."
}
```

(This is an illustrative example; actual output would have the correct Hebrew letters and values.) The strict schema is enforced by validation after this stage. If the output JSON doesn't match (e.g., a field missing or extra), the **schema validator** will catch it and fail the pipeline.

Make Target: `make ai.nouns` – runs ingestion (if needed) and noun extraction. Internally, this corresponds to the `collect_nouns` node in the LangGraph pipeline. The environment variables `BATCH_SIZE` (default 50), `ALLOW_PARTIAL`, and `PARTIAL_REASON` can influence this stage. By default, `ALLOW_PARTIAL=0`, so <50 nouns triggers a stop; if set to 1 (for testing smaller texts), a `PARTIAL_REASON` must be provided to record why a short output is acceptable.

Enrichment Agent (Theological Context Enrichment)

Purpose: Enhance each noun with theological context, links, and commentary, without altering the noun's core factual data. This provides depth to each node in the graph (beyond just a name and number).

Responsibilities: The Enrichment agent takes the `ai_nouns.json` from the previous stage and for each noun adds additional context in an **analysis.subfield** (often a sub-object like `analysis.theology`). It uses a **theology-focused LLM** (a domain-specific large model tuned for biblical text) to generate a brief commentary on each noun. This could include key themes or motifs associated with that noun, cross-references to other parts of scripture where the noun or concept appears, and any notable theological insights or symbolic meanings. The agent is instructed **not to fabricate references** – any scripture citations or claims must be verifiable in the actual text. It appends this information under a designated field (ensuring that the original data from noun extraction remains unchanged). In practice, the enrichment might produce something like: for a noun "Light", add an `analysis.theology` object with fields like `themes: [...]`, `cross_refs: [...]`, `notes: "Light is a symbol of knowledge and divine revelation..."` (for example). The key point is that **core fields (noun_id, gematria, etc.) are never modified** by this stage; it only adds new structured subfields for context. This keeps the nouns as SSOT data intact while layering interpretation on top. The use of a specialized expert model for this step ensures the info is relevant and higher fidelity than a general LLM might produce.

Output Contract: The Enrichment agent produces an updated nouns file, often named `ai_nouns.enriched.json` (or similar), following the same base schema as ai-nouns.v1 but extended with the added subfields. In other words, it contains the same list of noun nodes and all original fields, plus an added field like `analysis.theology` for each noun. The schema is designed to tolerate these known extension fields. The enriched file becomes the input for graph construction. If for any reason the enrichment fails (e.g., the model is unavailable), the pipeline is configured to fail closed, since proceeding without enrichment would violate the expected contract (alternatively, a stub analysis could be allowed via an override in a dev scenario).

Make Target: `make ai.enrich` – reads `ai_nouns.json` and runs the enrichment process to produce `ai_nouns.enriched.json`. This corresponds to the `enrichment` node in the pipeline. It requires that the LLM service (LM Studio) is running and the theology model is loaded, otherwise the orchestrator will detect the missing model and stop (the **Qwen health gate** ensures the pipeline fails if `USE_QWEN_EMBEDDINGS=true` but the model server is unreachable [1]). The default models are specified in environment (e.g., `EMBEDDING_MODEL=text-embedding-bge-m3` for embeddings, `RERANKER_MODEL=qwen-reranker` for rerank – the theology model might be a fine-tuned Qwen or similar).

Batch & Confidence Validation (Quality Gates)

Purpose: Automatically validate intermediate outputs for completeness and confidence, to ensure only high-quality data proceeds through the pipeline. (This encompasses the **validate_batch** and **confidence_validator** steps in the pipeline.)

Responsibilities: After noun extraction, the **Batch Validator** checks that the output meets minimum expectations. The main rule here is the **minimum noun count**: if the nouns JSON has fewer than a threshold (50 by default), it's considered an incomplete extraction and the pipeline will halt with an error unless partial output was explicitly permitted. This agent may also verify no obvious duplication or empty fields in the noun list. After enrichment, the **Confidence Validator** examines the LLM-generated content (the theology analysis) for any signs of low-confidence or rule violations. For instance, if the LLM model provided a confidence score or metadata for its answers (some setups do, though not explicitly described here, we have an **ADR-008 Confidence-Aware Batch Validation** in the design [2]), this agent would ensure those confidences are above a certain threshold. It might filter out or flag any enriched data that appears hallucinated or unsupported (e.g., if cross_references are made to verses that don't exist, or the content looks too speculative). Essentially, it acts as a lightweight QA on LLM outputs before they are used for graph building. If any noun's enrichment fails the check (for example, the model wasn't sufficiently confident or included a forbidden content), the pipeline could either drop that analysis or fail entirely, depending on strictness settings. The existence of a dedicated confidence gate in the pipeline reflects the project's caution in trusting LLM output – aligning with the priority **LLM = metadata only** (the LLM's contributions are always subject to verification).

Output Contract: These validation steps do not produce their own JSON artifacts but rather ensure the integrity of existing ones. The batch validator will either pass (allowing `ai_nouns.json` forward) or raise an error (if failing, no new output is generated; the run stops). The confidence validator similarly doesn't alter the `ai_nouns.enriched.json` except possibly to remove or null out discredited portions (though typically it would halt on serious issues). In logs and HINTs, these agents will note what they did (e.g., "HINT: partial output disallowed, halting" or "HINT: removed 2 low-confidence analysis entries"). They essentially enforce rules like *"No partial batch without ALLOW_PARTIAL"* (internal Rule-025 phase gate) and *"Confidence must meet threshold"* (from ADR-008).

Make Target: There isn't a user-facing make target just for these (they run internally as part of the pipeline flow). However, their logic is embedded in the orchestrator pipeline: `validate_batch` is a node right after collect_nouns, and `confidence_validator` is right after enrichment. These run automatically when using `make orchestrator.full` or `make orchestrator.pipeline`. If they trigger a failure, the orchestrator will report it. (For testing, one could simulate a partial run by setting `ALLOW_PARTIAL=1` with a reason to see the pipeline continue past `validate_batch`.)

Graph Builder Agent (Relation Extraction)

Purpose: Construct the initial **knowledge graph** of concepts by establishing relationships (edges) between the extracted nouns (nodes). This agent determines how the nouns are connected based on textual, semantic, and theological links.

Responsibilities: The Graph Builder reads the enriched nouns data (`ai_nouns.enriched.json`) and creates a graph structure, outputting a JSON file (e.g. `graph_latest.json`) that follows the **graph.v1 schema**. Each noun becomes a node in the graph (using the same noun_id), and the agent decides which pairs of nouns should have an edge between them and why. It uses several heuristics and data sources for this:
- **Textual Co-occurrence:** If two nouns appear together in one or more verses or within a close context window, an edge of type `"cooccur"` is added, indicating they co-occur in scripture. The strength might be proportional to how frequently or closely they co-occur.
- **Semantic Similarity:** Using vector embeddings of the nouns or the verses where they appear, the agent checks if two concepts are semantically related. If their cosine similarity is high above a threshold, it adds an edge of type `"semantic"`. This captures connections like synonyms or thematic similarity not obvious from co-occurrence.
- **Theological/Contextual Links:** Based on the enrichment analysis, if two nouns share a theme or a cross-reference (for example, both are related to “covenant” or both appear in a known story), the agent may add a `"theology"` edge linking them. It might use cues from the LLM commentary (like if the enrichment notes that one name is the father of another, a relationship can be formed).
- **Other:** Any additional domain-specific relations can be included as type `"other"`. For example, identical gematria values might be a reason to link two otherwise unrelated nouns (if that rule is enabled; an ADR on gematria equivalences exists).

The Graph Builder assigns a **weight** to each edge, normalized between 0 and 1, representing the strength or confidence of that connection. Initially, these weights may be default or based on simple criteria. For instance, any detected relation might start with weight 1.0 (or co-occurrence count scaled into [0,1]). Each edge can also carry an explanation in an `analysis.edge_reason` field explaining why those two nodes were linked (e.g., “co-occur in Genesis 1:3” or “share theme of light”). By design, **node IDs in the graph remain exactly the noun UUIDs** from the previous stage, so that every edge can trace back to the original noun data easily. The graph builder essentially transforms a list of items into a connected network.

Output Contract: The output is `graph_latest.json` following `gemantria/graph.v1` **schema**. This schema defines two main sections: a **nodes** array and an **edges** array. Each node entry should at least have an **id** (matching a noun_id) and possibly additional info like the noun’s surface text or a pointer to which book it’s from (these could be included for convenience). Each edge entry contains **src** (source node id), **dst** (destination node id), a **type** (semantic, cooccur, theology, other), and a **weight** (0.0–1.0 float). The schema requires that every edge’s `src` and `dst` refer to valid node IDs present in the nodes list (no orphan edges). Example edge: `{"src": "UUID1234", "dst": "UUID5678", "type": "semantic", "weight": 0.85}` would mean noun 1234 is semantically related to noun 5678 with strength 0.85.

Initially, without advanced scoring, the agent might output unrefined weights (even all edges weight 1 or a few discrete levels). The key is the structure is in place. This file is the SSOT for all relationships – downstream stages will *only* read this file for graph structure (not regenerate edges). As with others, a schema validation runs to ensure it meets contract (all required fields, no extraneous keys, and basic sanity like 0≤weight≤1).

Make Target: `make graph.build` – generates the base graph JSON from the enriched nouns. It corresponds to the `network_aggregator` node in the pipeline. Internally it will load any required models for embeddings if needed (embedding model for similarity). If the environment variable `USE_QWEN_EMBEDDINGS` is true, it ensures the embedding model is available (via LM Studio) before proceeding. Otherwise, it could rely on a local function or skip semantic edges. This target should be run after enrichment; the orchestrator will automatically run it in sequence if using the full pipeline.

Graph Rerank & Scoring Agent

Purpose: Refine the initial graph by computing more precise similarity metrics for each edge and classifying edge strengths (strong/weak connections). This stage enhances the graph with quantitative analysis, turning a raw network into a scored network.

Responsibilities: The Rerank/Scoring agent takes the `graph_latest.json` output from the Graph Builder and augments it to produce (or update) `graph_latest.scored.json`. For each edge in the graph, it calculates two key measures: a **cosine similarity** between the textual embeddings of the two linked nouns, and a **reranker score** from a dedicated model that judges the relevance of that connection. The embedding similarity provides a general semantic relatedness, while the reranker (often a more complex bi-encoder or cross-encoder model) can consider the pair in context for finer judgment. These two scores are then **blended** into a single **edge_strength** value using a formula such as: `edge_strength = α * cosine + (1 - α) * rerank_score`. By default α may be 0.5 (equal weighting). The agent then labels each edge with a category based on this strength: for example, if edge_strength ≥ 0.90 it might label as **strong**, if ≥0.75 as **weak**, else maybe **very_weak** or other categories. These thresholds are configurable but the idea is to filter/prioritize the graph's relationships – strong edges likely indicate very meaningful connections (the nouns are highly related by both semantic and context metrics), whereas very weak edges might be incidental and could be ignored in visualization. The result is that each edge in the graph gets additional fields like `"cosine": 0.82, "rerank_score": 0.90, "edge_strength": 0.85, "class": "weak"` (for the example where it blended to 0.85 which was below the strong threshold). Under the hood, this agent uses the embedding model (e.g., the text-embedding-bge-m3) to vectorize noun descriptions or contexts, and a local reranker model (e.g., Qwen3 or another LLM acting as a scoring function) – both models run locally to avoid external dependencies. The entire process is automated (no human input) and effectively **enriches the graph's edge data with metrics**.

Output Contract: The output is an updated graph JSON. In some implementations, the same `graph_latest.json` file is updated in-place to include the new fields. More clearly, the project references a `graph_latest.scored.json` which contains the same nodes and edges as the input, but with edges now having the extra fields and possibly filtered sets. The schema is effectively the same `graph.v1` schema but extended (the schema likely allows additional properties on edges such as cosine, rerank_score, class for strength). The **Graph Stats schema** (described later) will rely on these classifications to count how many strong/weak edges, etc. If any edges fell below a certain threshold, the agent might mark or even remove them (though removal typically would be a separate pruning step; in our case it sounds like they keep all edges but just label them, for completeness). After this, the graph is considered **fully built and scored**. This is the data that will be analyzed for patterns and also can feed a visualization.

Make Target: `make graph.score` – this runs the rerank/scoring process on the latest graph. It corresponds to part of the `analysis_runner` or a separate node just after graph building (depending on pipeline definition). Running `make orchestrator.full` includes this automatically. If one is iterating on the scoring method or thresholds, they can regenerate the scored graph by running `make graph.score` again (it will use the existing `graph_latest.json` as input). This target requires the embedding and reranker models to be loaded; it performs no internet calls (the approach is entirely local to meet privacy/control requirements).

Analytics & Pattern Analysis Agent

Purpose: Perform deeper network analysis on the fully scored graph to derive summary metrics, detect clusters/patterns, and produce human-readable reports of the findings.

Responsibilities: The Analytics agent ingests the completed graph (`graph_latest.scored.json`) and computes a variety of graph-theoretic metrics and patterns, outputting both structured data files and a summary report. Key tasks include:
- **Graph Statistics:** Calculate overall metrics such as the total number of nodes and edges, graph **density** (how filled the graph is between 0 and 1), the number of connected components or clusters, and centrality measures (like average degree, and possibly more advanced ones like betweenness or eigenvector centrality). These give a high-level health check of the graph – e.g. is it sparse or dense, how many distinct clusters of concepts exist, etc.
- **Community Detection:** Use algorithms (like Louvain or other community detection) to find groups of nouns that are tightly interconnected. The agent could identify clusters and output them (each cluster might be a set of nouns that form a theme).
- **Recurring Patterns:** Identify significant patterns such as motifs or subgraphs that appear (e.g., triads of concepts that are often linked). If temporal data is considered, find **temporal patterns** – how connections evolve across the narrative of the text (for example, do certain themes intensify in later chapters?). An ADR reference suggests multi-temporal analytics to track patterns over time and possibly forecast them.
- **Forecasting (Speculative):** There is mention of a `pattern_forecast.json` – possibly using something like Prophet or time-series modeling to project trends (e.g., if concept frequencies were treated like a time series). This might be a more experimental feature to predict how patterns might continue.
- **Report Generation:** Compile a concise **Markdown report** summarizing the key insights from the above analysis. This report might say, for instance: "The graph contains X nouns with Y connections. There are Z distinct clusters; the largest cluster centers on [noun A] and [noun B]. Notably, [noun C] has the highest centrality, appearing across many themes. A recurring pattern is observed linking concepts of 'covenant', 'sacrifice', and 'promise' across multiple books…" etc. The agent likely uses an LLM in a controlled way to help draft this narrative from the raw statistics (instructed to not introduce any facts not present in the data). This ensures the report is accurate and based on the computed results.

The analytics stage is largely algorithmic (using libraries like NetworkX for graph metrics) rather than LLM-driven, which adds trustworthiness to the metrics. The LLM's role, if any, is in interpreting or nicely formatting the findings in the report.

Output Contract: This stage produces multiple outputs, each with its own JSON schema (defined in the `schemas/` directory). Important artifacts include:
- `graph_stats.json` – following the **GraphStats schema**, containing high-level metrics: node count, edge count, number of clusters, density of the graph, and possibly aggregate centrality values (e.g., avg degree, avg betweenness). It may also list counts of edges by strength category (e.g., how many strong, weak edges) and distribution info like the range of cosine scores observed.
- `graph_patterns.json` – structured data about recurring patterns or motifs found. The schema might capture common triples or subgraph patterns, or thematic groupings that aren't just clusters (for example, frequently co-occurring triples of nouns).
- `temporal_patterns.json` – data about patterns over time (since each noun's sources have verse references, and verses have order, one can derive how connections appear across chapters or books). This file could list, for instance, connections that first appear in later chapters or trends like "cluster around concept X grows in the latter half of the book."
- `pattern_forecast.json` – if implemented, outputs of any predictive model guessing how patterns might continue or what future data (hypothetical) might look like.
- `report.md` – a human-friendly summary of all the above, suitable for stakeholders to read as an overview.

All these outputs are governed by schemas to keep them consistent and machine-readable. For example, GraphStats schema ensures fields like *total_nodes*, *total_edges*, *num_clusters*, *density* are present and in a certain type/range. An excerpt: it records things like average degree or betweenness to summarize connectivity. The outputs are also inserted into the Gematria DB (clusters, centrality tables, etc.) for persistence and querying. The Markdown report does not need a schema but is kept concise (a few pages at most) highlighting key points.

Make Target: `make analytics.export` (or similar) – runs the analytics and generates all export files. In AGENTS.md, this corresponds to the `analysis_runner` node and is marked as NEW in the integrated pipeline. The orchestrator can run it as part of full pipeline or one can run analysis separately if the graph is already built (e.g., `make orchestrator.analysis OPERATION=all` as mentioned). After running, one should have all the JSONs in `exports/` as well as a `report.md` (often stored in `share/` or `exports/`). This stage typically runs quickly (graph algorithms on a few thousand nodes are fast), and it's often executed in CI for validation too.

Guard/Compliance Agent (Final QA Gate)

Purpose: Rigorously verify that all outputs adhere to schemas and rules, providing a final quality gate before declaring a pipeline run successful. This agent acts as an automated reviewer ensuring **nothing deviates from the SSOT contracts or project policies**.

Responsibilities: The Guard agent runs after all analysis is done (but before releasing artifacts). It performs checks such as:
- **Schema validation for every JSON output:** It validates each output file (ai_nouns, graph_latest, graph_stats, etc.) against its JSON schema definition. If any required field is missing, any extra field is present, or a value is out-of-type/range, this agent flags it. This ensures that upstream stages did not violate their data contracts (for example, if an LLM accidentally added an unexpected key somewhere, it will be caught here).
- **Data integrity invariants:** Beyond schemas, it checks domain-specific invariants. E.g., no "orphan" edges

(every edge's nodes exist), no empty text fields where there shouldn't be, numeric values make sense (like gematria values match the letters), etc. It likely implements specific rules like *Rule-039 Execution Contract* which might stipulate that if the pipeline is re-run, the outputs must exactly match unless an ADR says otherwise, etc. Another example: verify that if the pipeline allowed a partial (via ALLOW_PARTIAL), a proper manifest/log was produced (the noun stage said why it was partial).
- **Procedural compliance:** It checks if any infrastructure or schema changes occurred and if so, ensures the PR or run includes acknowledgement. For instance, *ADR enforcement*: if a schema changed (say a new field was added to ai_nouns), the guard looks for a reference to an ADR in the commit or PR (this corresponds to internal rules requiring ADRs for data schema changes). If such a change is not noted, it flags an error.
- **CI integration:** In continuous integration, this guard runs as part of the pipeline tests (and nightly in a broader sweep) to catch any drift or rule violations automatically. It's essentially a "fail-closed gating step" – if any issue is found, it will mark the pipeline run as failed and output the reason. Only if everything passes does it emit a success (often a "GUARD_OK" status).

This agent embodies many of the project's stringent policies at the code level. It is informed by the RULES_INDEX and specific high-priority rules. For example, rules about schema fidelity (Rule-026, Rule-039) are enforced here. If an LLM output had snuck in an extra field, the guard failing prevents that from going unnoticed. If a developer forgot to update documentation for a change, guard would catch it (via the docs sync rule, Rule-027). In effect, it's the automated "compliance officer" for the pipeline.

Output Contract: The Guard doesn't create a new JSON for data, but it may produce a log or record of its checks (possibly updating a PR comment or outputting a summary JSON of checks). In a CI context, its "output" is a pass/fail status. If it fails, the pipeline stops and the error must be addressed (ensuring SSOT principle holds). If it passes, it gives the green light for **Release**.

Make Target: `make guards.all` (as hinted by internal docs) – likely runs all guard checks on current outputs. This can be run manually by developers to verify their changes before pushing. In CI, an equivalent step is included (e.g., the **ssot-nightly** or **system-enforcement** workflow). The guard agent is also implicitly part of `make orchestrator.full` at the end (as a pipeline node after analysis). Only after `guards.all` passes does the orchestrator proceed to release packaging.

Release/Operator Agent

Purpose: Finalize the pipeline run by assembling all outputs, generating any release notes or metadata, and performing deployment or publication actions as needed. Essentially, this agent **packages the verified artifacts** and wraps up the run.

Responsibilities: The Release (a.k.a Operator) agent assumes all prior steps succeeded and outputs are verified. Its tasks include:
- **Artifact Packaging:** Collect all the final JSON files (nouns, graph, stats, patterns, etc.) and prepare a release bundle. This could mean copying them to a designated folder, zipping them, or attaching checksums for each. It ensures that an authoritative list of outputs is produced so nothing is missed.
- **Release Notes/Summary:** It generates a summary of the run, possibly including version info, which book was processed, timestamps, and key results. If using version control, it might tag a new version in Git or update a CHANGELOG with highlights of what was generated. It can incorporate the Markdown report from analytics or at least reference it. The agent is instructed to list an **acceptance checklist** – essentially documenting that all needed steps were done and all outputs are present. For example, it might output a

short markdown or JSON saying "Artifacts: X files generated (list with names and hashes), All tests pass, Data verified by Guard, etc.".
- **Deployment (if applicable):** In a production context, this step could automatically push data into a database or trigger an update to a front-end. For Gemantria's scope, it might, for instance, write the outputs to the application database (the Gematria DB) for the UI to consume, or upload the results to a cloud storage. The document suggests it "performs any deployment or publishing steps" that might be needed.
- **Governance Metadata:** As noted in the gap analysis, a possible enhancement is for the release agent to compile a final metadata block listing what versions of models were used, which ADRs are relevant, and any rule exceptions that were encountered. This would create a traceable record for governance (e.g., "Run completed using TheologyModel v2, ADRs 001, 009 applied, no rule exceptions"). This helps future audits see under what policy the data was produced [3].

Overall, the Release agent's job is to **formalize the outcome** of the pipeline run. After it, the pipeline is considered complete and its outputs ready for use.

Output Contract: The main "output" is a release package (could be conceptual, like a collection of files and a summary note). If a GitHub release or tag is created, that's an output. It may produce a file like `release_manifest.json` enumerating files and their checksums. The agent likely prints or logs a final summary. Importantly, it does *not* alter any data – it only aggregates and reports. It might also mark in the database that the run is done (could insert a record of completion).

Make Target: `make release.prepare` – presumably to run the release agent tasks. In integrated use, `orchestrator.full` will include this. The **PR template** for contributors hints that after a run, they should expect certain "emitted HINTs" and evidence of the process, which this stage might provide (like confirmation messages). After this, the data can be handed off to the UI/dashboard for visualization or to analysts.

Schema Definitions (Key JSON Schemas)

At each stage, Gemantria uses a strict JSON schema to define the output format. This enforces consistency and **single-source-of-truth** contracts between stages. The key schemas include:

- **AI Nouns Schema (`gemantria/ai-nouns.v1`):** Defines the structure for noun extraction outputs. As described, an AI nouns JSON has a top-level object with metadata (`schema`, `book`, `generated_at`) and an array of **nodes**. Each node must contain: a `noun_id` (UUID), the noun's `surface` text, array of `letters`, its `gematria` value, a `class` (e.g. person, place, thing), `sources` (each with scripture reference and character offset), and an `analysis` field for additional notes. No extra fields are allowed and none can be missing – producers (whether code or LLM) must output exactly this structure. This precise contract ensures any consumer (like the graph builder) knows exactly where to find each piece of data (e.g., it can trust that every noun has a gematria field computed, etc.).

- **Graph Schema (`gemantria/graph.v1`):** Specifies the format of the concept network JSON. The graph JSON contains a list of **nodes** and a list of **edges**. Node entries typically include at least an `id` (which should match a noun_id from the AI nouns stage) and possibly a `surface` name or other context. Edge entries include `src` node id, `dst` node id, a relation `type` (allowed values: `semantic`, `cooccur`, `theology`, or `other`), and a numeric `weight`. All weights are normalized between 0 and 1. The schema ensures referential integrity: every `src`/`dst` in edges must correspond to an id in the nodes list. It also fixes the allowed relation types so consumers (analytics, UI) can treat them accordingly. An example edge per schema: `{"src": "<noun_id1>", "dst": "<noun_id2>", "type": "semantic", "weight": 0.85}`. The **Graph Scored** variant (after reranking) follows the same schema but with additional fields on edges. Those fields (cosine, rerank_score, strength class) are either incorporated by an updated schema version or accepted as extensions (the project documentation treats them as augmentations to graph.v1). Regardless, the presence of those fields should be documented, and a rule (Rule-045) exists to enforce the correctness of the blend formula on every edge.

- **GraphStats Schema:** Defines the content of `graph_stats.json` produced in analysis. This schema includes top-level fields summarizing the graph: e.g., `total_nodes`, `total_edges`, `num_clusters` (communities detected), and measures of centrality or connectivity. For instance, it may have `average_degree` or `avg_betweenness` to describe how connected nodes are on average. It can also include distribution info, such as counts of edges by category (how many strong/weak edges) and a basic histogram of similarity scores perhaps. The goal is to present in JSON all key metrics that one would want to track or compare between runs. According to the Single Source documentation, GraphStats records things like graph density (ratio of actual edges to maximum possible edges) and cluster sizes, etc., in a structured way. An example snippet might be: `{"total_nodes": 3702, "total_edges": 1855, "num_clusters": 26, "density": 0.00027, "avg_degree": 1.00, "edge_strength_counts": {"strong": 741, "weak": 422, "very_weak": 692}}` (values taken from an initial snapshot described in the technical overview). This schema allows automated comparisons of graph properties over time or between books.

- **Pattern Schemas:** There are also schemas for `graph_patterns.json`, `temporal_patterns.json`, `pattern_forecast.json`, etc., each ensuring a consistent format for those outputs. For example, `graph_patterns.json` might list patterns each with an id, description, and occurrences count. `temporal_patterns.json` could have entries for each pattern or theme with a timeline of frequency. The exact fields are determined in the schema files (which were referenced in docs but not fully listed in the excerpt). The important point is each of these JSON outputs is **self-describing** with a `"schema": "gemantria/<name>.v1"` field and a defined structure so that front-end visualizations or further analysis can consume them without ambiguity.

- **Unified Envelope Schema (Phase 11):** There is mention in internal docs of an integrated envelope (`unified_envelope.json`) that might combine various outputs for UI use. This would be governed by a schema (`unified-v1`) and include an amalgamation of nodes, edges, patterns, etc., possibly used for a single JSON export to drive the dashboard. That schema would ensure all parts (graph, patterns, trends) align in one structure.

All schema files are versioned and stored in the repository's `schemas/` directory. The project enforces that **if any output format changes, the corresponding schema file must be updated and the change must be justified (often via an ADR)**. Schema validation is run as part of CI (e.g., `make schema.validate` and guard checks) to catch any mismatch immediately. This strict schema governance is central to maintaining the SSOT principle across the pipeline.

CI/CD and Operational Practices

Gemantria's CI/CD pipeline is designed to uphold the project's strict quality standards and keep the development process hermetic (controlled and reproducible). The **Continuous Integration (CI)** configuration includes both required checks on pull requests and nightly maintenance jobs.

CI Checks and Workflow: Every pull request must pass the **formatting/linting** and **build/test** workflows before it can be merged. The formatting check uses **Ruff** (which is the single source of truth for code style) – CI is configured to run `ruff format` (to verify formatting) and `ruff check` (lint) and will fail if any style issues are present. The CI build runs unit tests, integration tests, and pipeline smoke tests. The project enforces a **coverage threshold of 98%** test coverage for the code – there's a plugin that will cause CI to fail if coverage drops below this, ensuring new changes remain well-tested. There are shields/badges that track test coverage and quality trends over time, updated by the CI jobs. Branch protection is set such that at least the **"ruff"** check and the **"build"** (CI) must succeed before merging. This guarantees consistent code quality.

Database and Environment: CI runs in a fresh environment, so the pipeline is built to handle "first-run" scenarios gracefully. An **empty DB tolerance** policy ensures that if the application database starts empty, the setup scripts will create the necessary schema and not error out. Specifically, the CI process uses a script (`ensure_db_then_migrate.sh`) to bootstrap an empty database and apply all migrations before tests run. The pipeline agents themselves are coded to allow zero rows in DB tables when appropriate (e.g., Stats validation allows that if no data yet, it doesn't break). Similarly, file-dependent steps in CI anticipate that some outputs might not exist on first run; the code uses **file tolerance** to handle missing graphs or stats by substituting empty defaults rather than crashing. All file operations in scripts are preceded by existence checks – a practice enforced by **Rule-046** – to avoid blindly reading/writing files that aren't there. If a critical file is missing when it shouldn't be, the CI will log a loud failure (per Rule-039 fail-closed principle) rather than quietly make a new empty file.

Nightly & Periodic Jobs: The project has a set of scheduled **nightly workflows** (and possibly weekly). These include: **lint-nightly**, **typing-nightly**, **coverage-nightly**, **quality-badges**, **graph-nightly**, **ssot-nightly**, **soft-checks**, etc., as listed in the repository's CI config. The purpose of these jobs is to run more extensive checks that aren't required on every PR but ensure no bit rot: for example, *coverage-nightly* might run the full test suite with coverage and update badge artifacts; *lint-nightly* might enforce any formatting changes (though formatting is usually done on PRs); *ssot-nightly* likely revalidates that all documentation (ADRs, rules, agent docs) are in sync (fulfilling Rule-027 docs sync) and that all JSON in the repo matches their schemas (this is hinted as a non-blocking nightly sweep for schema validation). There's also a **graph-nightly** which perhaps runs the pipeline on a known dataset to produce a fresh graph and ensure nothing crashes over time. A **scorecards.yml** job runs the OpenSSF Scorecard security analysis nightly, uploading results to GitHub's security tab. Additionally, **Dependabot** is enabled (weekly for GitHub Actions) to keep dependencies up to date, and third-party actions are pinned to specific commit SHAs to prevent supply-chain drift. All these ensure the repo remains healthy and secure over time.

Makefile Conventions: There are various make targets to help with CI and ops. For instance, `make ops.verify` is a local target that runs deterministic checks to ensure the ops environment is correct (it checks that certain Phase-8 eval artifacts exist, etc.). This is used to catch issues in development before CI. Developers are encouraged to run `make lint type test.unit test.integration coverage.report` locally before pushing to pre-empt CI failures.

Pre-commit Hooks: A pre-commit configuration is present, which runs formatting and possibly other checks on commit. Notably, **share directory sync** is done before other checks: the repo uses a `make share.sync` target to synchronize the `share/` directory (which holds derived artifacts like badges or evaluation outputs) to ensure no generated files are out-of-date or missing. This runs before the `repo.audit` (which might check for things like file diffs or rule gaps) to satisfy Rule-030 (share sync) etc. The ordering guarantees that if developers forgot to commit an updated artifact, the pre-commit will update it.

Continuous Deployment / Release: While not explicitly a public deployment, the project treats a successful main branch build as something that could be released. If this were a product, one could set up CD to publish a new version of data or UI. In the current context, "release" is mainly about packaging data for internal use, but the pipeline's final stage is analogous to a deployment (to DB or to a release note). The **Phase-9 and Phase-10** documentation (referenced in README) suggest that later phases include a dashboard UI integration. The CI likely includes jobs to build or test the UI if it existed (e.g., a **pr-comment.yml** might post summaries on PRs, and **policy-gate.yml** could enforce commit message policies or ADR link presence).

Empty Data Handling and Idempotence: CI ensures that a brand new environment (no data, caches cleared) can run the full pipeline deterministically. The pipeline and tests are idempotent – repeated runs produce the same outputs, which is verified by comparing content hashes or through Rule-053 (Idempotent Baseline) caching. Rule-053 specifically caches a baseline evidence for up to 60 minutes to avoid redundant rechecks in a PR session. The bot will not re-run expensive baseline commands if nothing changed, and will note a `HINT[baseline.sha]` when using cached evidence. This helps CI efficiency while still ensuring evidence is fresh when needed.

In summary, the CI/CD strategy for Gemantria is to **catch any deviation or gap early** – whether in code style, tests, data schema, or documentation. It automates enforcement of the SSOT and other rules as much as possible, and where automation isn't feasible, it provides clear guidelines for human reviewers to follow.

PR & Contribution Standards

Contributing to Gemantria is governed by strict guidelines to ensure consistency, transparency, and compliance with project rules. All code changes flow through a **pull request (PR) process** with templates and checklists that must be followed.

Workflow Expectations: Developers work on feature branches (named `feature/<short-name>`) and are expected to write tests **first** for any new functionality. Code changes should be accompanied by unit tests and possibly integration tests, maintaining the high coverage bar (≥98%). Every commit message should follow the **Conventional Commits** style, e.g. `feat(area): short description [tags]` – with tags like `no-mocks`, `deterministic`, or `ci:green` to denote special contexts. This format and tagging helps in automated changelog generation and in indicating that certain criteria (no mocks used, determinism maintained, CI passing) are met in the commit. Large changes are broken into "small green PRs" whenever possible (small, focused, all tests passing).

Pull Request Template: Gemantria provides a PR template (which is automatically loaded when creating a PR) that contributors must fill out. The structure is as follows:
- **Title:** A concise, imperative title, including a PR code (if any) and scope keyword (e.g., `feat: ... (PR 024c)`).
- **Summary:** 2–4 lines describing what is being changed and why.
- **References (REQUIRED):** This section is crucial – the author must list relevant project artifacts:
- *Rules cited:* The specific rule numbers (and short names) that relate to this change. For example, if the change touches schema validation, they might cite *039-execution-contract* or *026-system-enforcement-bridge*. This ensures the PR is contextualized in the governance framework.
- *Agents referenced:* Any agent or section of AGENTS.md that is relevant (e.g., if modifying how the orchestrator works, reference `AGENTS.md#orchestrator`).
- *Docs touched:* List of documentation files updated (if the PR includes doc changes).
- *SSOT links:* Links to SSOT master documents or ADRs sections that are pertinent. For instance, if an ADR is supposed to be updated or a rule in RULES_INDEX is affected, link to that.
- **Scope (files):** An explicit list of files or directories that the PR touches. This should correspond to the diff; no drive-by changes outside this scope. It forces authors to be deliberate about what's included.
- **Acceptance (governance v6.2.3):** A checklist of required verifications the author must ensure before marking the PR ready. This includes:
- [ ] Ruff format/check green (code is formatted and linted according to SSOT standard).
- [] `make ops.verify` passes (which includes checks like no duplicate Make targets, etc., per Rule-054).
- [ ] Relevant smoke tests are green (ensuring basic pipeline operations aren't broken).
- [ ] No new "knobs" or duplicate scaffolding introduced (meaning no new global flags or redundant code paths without approval). This aligns with avoiding unnecessary complexity.
- **Emitted Hints (REQUIRED):** The author must list the key HINT lines their change will trigger at runtime. For example, "HINT: verify: database bootstrap OK" or any custom HINT they added. This makes both the CI and reviewers aware of new log outputs and ensures they were intentionally introduced (and not accidental). It improves observability of changes.
- **Evidence tails:** A place to paste the tail end of logs or test output if something failed or to show final success states. For example, if a test initially failed, paste the failing snippet; or after fixes, the passing evidence. This helps reviewers see the proof of behavior without needing to reproduce everything.

This template enforces that every PR is linked to the project's governing documents and rules – effectively tying changes back to the design decisions and policies. It also ensures that contributors perform a self-check (formatting, ops.verify, etc.) before seeking review.

Review Process: Once a PR is opened with the template filled, the CI will run. Only when **required checks are green** should a PR be merged. According to **Rule-051**, if all required checks (e.g., ruff and build) pass, any **non-required** automated reviews (like a bot analysis) are to be treated as **advisory only**. The Cursor bot will comment explicitly that "Required checks are green; automated review is advisory per AGENTS.md and Rule 051.". This prevents automated tools from blocking merges once the essentials are satisfied, though their feedback can still be considered by humans. At least **one human approval** is required for every PR (this is part of merge policy Rule-041). Reviewers are expected to verify that the PR's content matches the described scope, that all referenced rules/ADRs are addressed, and that evidence of tests passing is provided. If an ADR is required (e.g., an infrastructure change with no ADR, or a schema change without an ADR update), reviewers should demand it before approval (the guard agent might catch this too).

Evidence Bundle: The combination of the template's references, emitted hints, and evidence tails constitutes an "evidence bundle" for the change. It means that alongside the code, there is a trail of *why the change was made (rules, docs references)* and *what the change did (test/output evidence)*. This practice is enforced to the point that the PR will not be approved without it. It dramatically eases the reviewer's job and ensures traceability of each change to requirements.

Model Usage Documentation: If the PR involves UI/frontend generation or LLM usage, the team requires documenting model usage. In AGENTS.md, under UI codegen guidelines, it states that every PR **must include a "Model Usage" block** describing which AI models were used and how many iterations, for any AI-assisted generation. This is likely to appear in the PR body (outside the default template, maybe an addition). It ensures transparency when code is produced or influenced by AI.

Commit Hygiene: Fast-forward merges or squash merges are likely used (the template hints at squash via CLI). The **"policy-gate"** workflow (and branch protection) might require commits to be GPG-signed or verifiable – Rule-041 mentions signed-commit verification and conventional commit format as a gate. Thus, contributors should sign their commits and use the proper message format, or the **policy-gate.yml** will fail the PR.

In short, contributing to Gemantria means adhering to a disciplined process: small, well-tested changes, detailed documentation in the PR, explicit linkage to governing rules, and providing evidence. This not only maintains quality but also trains contributors to think in terms of the project's policies and long-term maintainability. By the time a PR is merged, it should be very clear *what* was changed, *why* (with respect to project rules/ADRs), and *how* it was validated.

Development Guidelines and Best Practices

The project's development guidelines ensure that all contributors and systems maintain the integrity, performance, and observability of Gemantria. Key areas of guidance include infrastructure, coding standards, model usage, and logging/metrics.

Infrastructure & Safety: The architecture uses a **two-database setup** (ADR-001) to enforce data safety. Developers must respect that **Bible_DB is read-only** – any attempt to write to it is prevented (a read-only adapter is used to enforce this at connection time). All writes go to the Gematria DB, which stores derived data (nouns, relations, metrics). Even within Gematria DB, caution is exercised: all SQL queries must be parameterized (no string concatenation for queries) to prevent injection. When adding migrations or changing schemas, one must also update corresponding JSON schemas and likely write an ADR explaining the change. The environment provides variables for DB connections (`BIBLE_DB_DSN`, `GEMATRIA_DSN`), and developers should use these rather than hard-coding any credentials or paths. The pipeline should always be run with the Bible DB in read-only mode to ensure queries (like noun extraction queries) cannot accidentally modify data; this is a core safety rule (Rule-001 DB safety).

Determinism & Seeds: Deterministic behavior is paramount. The agents use fixed random seeds for any stochastic process (e.g., if an embedding model has some randomness, a seed ensures same results each run). The pipeline state carries a `position_index` and other markers so that if resumed, it knows where to continue without divergence. Developers should not introduce any non-deterministic elements without a mechanism to control them. If a random process is needed, it should either be seeded or its outcomes should be validated not to affect final results significantly (and even then, better to avoid). This allows consistent debugging and trust in the reproducibility of results.

Coding Standards: Gemantria adopts strict coding conventions as Single Source of Truth. The codebase is formatted with **Ruff (ruff + ruff-format)** and no other formatter; all contributors run `ruff format` and CI will enforce this. The style guide includes: **max line length 120**, all imports at top of file (no inline imports), prefer list comprehensions or generator expressions over map/filter in many cases (though not explicitly in snippet, typical of projects), and specific idioms like using `["cmd", *args]` instead of concatenating lists for subprocess commands. Type checking is done with **MyPy** in a forgiving mode for external libraries (ignore_missing_imports), but all internal code should have proper type annotations. New development should maintain or improve code coverage – if adding a module, add tests for it; if fixing a bug, include a regression test.

Logging & Metrics: Observability is built-in. Each agent writes a machine-readable log of key actions to the `share/evidence/` folder. Developers extending agent logic should continue this practice – e.g., if a new decision point is added to an agent, log a brief summary to evidence (this can be as simple as printing a JSON line or writing to a structured log file). The evidence logs include counts (like "extracted 120 nouns") and notable decisions ("pruned 2 weak edges due to threshold X"). These logs are later used to debug or to provide context in PRs.

Metrics collection is also addressed: an ADR (ADR-005) covers **metrics & structured logging**, meaning the system likely integrates with **Prometheus** or a similar monitoring system. Indeed, the design mentions one can instrument metrics in code and scrape them via Prometheus. For example, one could count how often the math agent had to correct an LLM's gematria, or how many nouns each book yields, etc., and expose those metrics. Developers adding new long-running processes or significant steps should consider adding metrics for them. There's also an **observability dashboard** ADR (ADR-006) which suggests that metrics are visualized, so adding a metric implies adding it to the dashboard documentation. The logging system and metrics work together: logs give detailed per-run evidence, metrics give aggregated trends over time.

Local LLM Usage (LM Studio): All AI inference is done with **local models** through **LM Studio**, not external APIs. Developers working on LLM integration should ensure models are obtained and configured to run locally. The environment variable `USE_QWEN_EMBEDDINGS` and model name constants indicate which models to load. To set up LM Studio for development, one runs `lms server start --port 9994 --gpu=1.0` which the AGENTS.md explicitly notes. This spins up a local server that mimics the OpenAI API, hosting the chosen models. The default models at time of writing are: **text-embedding-bge-m3** (embedding model) and **qwen-reranker** (for rerank scoring), and presumably a Qwen 12B or similar for theology expert. If a developer wants to swap models or test new ones, they can do so in LM Studio, but they must keep in mind memory and determinism. The **"Live Gate"** is a mechanism that fails the pipeline if `USE_QWEN_EMBEDDINGS=true` but the models are not actually available at runtime. This ensures no silent fallback to a different mode – if the intent was to use the model and it's not running, the pipeline stops. The guideline here: always double-check your LM Studio is running the needed models before pipeline runs; and never rely on internet for model inference, it should be local.

Resource and Performance Considerations: The pipeline can be heavy (embedding many verses, running a 12B model etc.). Developers should be mindful of performance – e.g., use batch processing where possible, avoid O(n^2) loops on large lists (prefer vectorized operations or database queries). There is mention of using Postgres for checkpointing and also for storing results for quick querying, which means sometimes leveraging SQL for heavy filtering rather than Python loops. ADR-003 about batch semantics indicates the pipeline processes in batches of 50 verses/nouns to manage load, so developers should maintain that pattern (process in chunks, validate each chunk). If adding a new analysis that could blow up computation (like a huge all-pairs comparison), consider approximations or offload to efficient libraries.

UI/Frontend Guidelines: (Though the main focus is backend, the docs included some UI generation rules.) If contributing to the front-end, start with **Gemini 2.5 Pro** model for React/Next codegen and only escalate to Claude 4 for complex refactors. This implies the project sometimes uses AI to help generate UI code under controlled conditions. All such usage must be logged in the PR "Model Usage" section. The front-end should use the agreed stack: React 18+, Zustand or Context for state, Recharts/D3 for graphs, Tailwind for styling. These choices are fixed to ensure consistency, so a developer should not introduce a new state library or chart library without discussion. The file structure for front-end is defined (components/, hooks/, services/, views/) – adhere to it when adding new UI components.

Housekeeping and Automation: There are automated tasks for upkeep: e.g., **auto-docs sync** (Rule-055) ensures that documentation (like this reference doc, ADRs, rules index) stay updated when code changes. If you change something that affects the docs, do update the docs in the same PR, or the auto-docs sync will flag it (and possibly nightly job will catch it). Similarly, **auto-housekeeping** (Rule-058) might remove unused imports, sort things, etc., on a schedule – so don't fight those; it's easier to comply in your PR than to have the nightly bot do a cleanup PR later. **Context persistence** (Rule-059) likely refers to ensuring any state in the AI agent (conversation context) is saved if needed; as a dev, be mindful if you introduce multi-turn interactions.

In essence, the development guidelines can be summarized: **follow the rules and document everything.** Infrastructure changes require ADRs and cannot compromise DB safety. Code must pass strict format/lint/test gates. Use local resources for AI, ensure reproducibility, and log what you do. By adhering to these guidelines, developers contribute in a way that keeps the project robust, transparent, and aligned with its long-term vision.

Rules Inventory (Project Policies Index)

Gemantria maintains a comprehensive index of internal rules that govern all aspects of the project. Each rule is numbered and titled for easy reference (and many are enforced via code or review). The following is an inventory of the rules as listed in **RULES_INDEX.md** and the Forest Overview, with their short descriptions:

- **Rule 000 – SSOT Index:** *(Single Source of Truth index references)*
- **Rule 001 – DB Safety:** *(Ensure Bible DB is RO, two-DB separation, etc.)*
- **Rule 002 – Gematria Validation:** *(Correctness of gematria calculations and normalization)*
- **Rule 003 – Graph and Batch:** *(Pipeline batch size semantics, validation gates)*
- **Rule 004 – PR Workflow:** *(Enforce PR process: small green PRs, template use)*
- **Rule 005 – GitHub Operations:** *(Guidelines for using GitHub API, MCP bot operations)*

- **Rule 006 – AGENTS.md Governance:** *(Makes AGENTS.md itself a governed artifact – agent behavior rules)*
- **Rule 007 – Infrastructure:** *(Infrastructure code rules, e.g., use of certain libraries or patterns)*
- **Rule 008 – Cursor Rule Authoring:** *(How new rules for the assistant (Cursor) are written/tested)*
- **Rule 009 – Documentation Sync:** *(Docs (like ADRs, this reference) must be updated alongside code)*
- **Rule 010 – Task Brief:** *(Every task given to the AI agents must have a clear brief, probably related to prompts)*
- **Rule 011 – Production Safety:** *(Guidelines to ensure features are production-safe, maybe toggles or gating risky features)*
- **Rule 012 – Connectivity Troubleshooting:** *(Procedures or allowances for handling network issues, etc., given local setup)*
- **Rule 013 – Report Generation Verification:** *(Ensure generated reports are verified for accuracy)*
- **Rule 014 – Governance Index:** *(Probably points to maintaining an index of governance docs/rules; might tie into forest regen)*
- **Rule 015 – Semantic Export Compliance:** *(Ensure semantic data (embeddings, etc.) meet export format and compliance requirements)*
- **Rule 016 – Visualization Contract Sync:** *(Sync the contract between data and visualization specs)*
- **Rule 017 – Agent Docs Presence:** *(Ensure every agent has documentation present in repo)*
- **Rule 018 – SSOT Linkage:** *(All Single Source of Truth elements (schemas, code, docs) must be linked and consistent)*
- **Rule 019 – Metrics Contract Sync:** *(Keep metrics collection in sync with defined contract or ADR)*
- **Rule 020 – Ontology Forward Compat:** *(Design ontology (concept definitions) to be forward-compatible with future changes)*
- **Rule 021 – Stats Proof:** *(Proof or verification steps for stats outputs, ensure reproducibility of statistical results)*
- **Rule 022 – Visualization Contract Sync:** *(Likely a duplicate title in index – possibly meant for a variation or extension of Rule 016)*
- **Rule 023 – Visualization API Spec:** *(Define and enforce the spec for any visualization API endpoints or data)*
- **Rule 024 – Dashboard UI Spec:** *(Requirements for the Phase-10 dashboard UI integration spec)*
- **Rule 025 – Phase Gate:** *(Phase gate system – certain conditions must be met before new phases or PRs, e.g., forest regeneration before PRs)*
- **Rule 026 – System Enforcement Bridge:** *(Bridge between system (runtime) checks and governance – e.g., hints envelope enforcement)*
- **Rule 027 – Docs Sync Gate:** *(CI gate that ensures documentation, ADRs, rules are updated when code changes – no undocumented changes)*
- **Rule 028 – Phase Freshness:** *(Likely ensures that pipeline phases are up-to-date or rerun if stale, perhaps requiring forest regen if code changed)*
- **Rule 029 – ADR Coverage:** *(Any significant change must be covered by an ADR entry – prevents untracked design changes)*
- **Rule 030 – Share Sync:** *(The `/share` directory (artifacts like badges, evidence) must be kept in sync; CI or pre-commit ensures no drift)*
- **Rule 031 – Correlation Visualization Validation:** *(Validate correlation analysis outputs for visualization)*
- **Rule 032 – Pattern Integrity Validation:** *(Ensure integrity of pattern detection outputs – no contradictions or invalid data)*

- **Rule 033 – Visualization API Validation:** *(Test and validate that the visualization API (if any) returns correct data)*
- **Rule 034 – Temporal Suite:** *(Guidelines or tests for the temporal analytics suite, ensure chronological data is handled correctly)*
- **Rule 035 – Forecasting Spec:** *(Spec for the forecasting component, ensure it's adhered to by implementation)*
- **Rule 036 – Temporal Visualization Spec:** *(Specifically ensure temporal visualization aspects (charts over time) meet spec)*
- **Rule 037 – Data Persistence Completeness:** *(All data that should be persisted (DB or files) is indeed persisted and complete – no partial saves)*
- **Rule 038 – Exports Smoke Gate:** *(A smoke test (quick run) of exports must pass – likely ensures* `make ci.exports.smoke` *passes as gate)*
- **Rule 039 – Execution Contract:** *(The contract that all pipeline executions must meet – e.g., idempotence, no missing files, etc.)*
- **Rule 040 – CI Triage Playbook:** *(Guidelines for triaging CI failures – what is considered infra fault vs code fault, etc.)*
- **Rule 041 – PR Merge Policy:** *(The conditions for merging PRs – required status checks, approvals, commit conventions)*
- **Rule 042 – Formatter Single Source of Truth:** *(Use only one formatter (Ruff) and no deviance – any formatting issue is a failure)*
- **Rule 043 – CI DB Bootstrap:** *(Ensure CI can bootstrap DB from scratch – missing DB should be created without failing)*
- **Rule 044 – Share Manifest Contract:** *(Likely ensure that any share directory artifacts (like evidence bundle or manifests) follow a contract or are up to date)*
- **Rule 045 – Rerank Blend SSOT:** *(The blend formula for edge strength (cosine vs rerank) is single-sourced and must match in all places)*
- **Rule 046 – CI Hermetic Fallbacks:** *(CI must not rely on network or external state – if something not available, use hermetic fallback values; e.g., missing files -> use empty defaults)*
- **Rule 047 – *Reserved*:** *(Placeholder for future rule, kept to preserve numbering continuity)*
- **Rule 048 – *Reserved*:** *(Placeholder)*
- **Rule 049 – GPT-5 Contract v5.2:** *(Likely guidelines for using GPT-5 (when it becomes available) or how to interface with it, version 5.2 specifics)*
- **Rule 050 – OPS Contract:** *(Defines how the OPS (operations) agent interacts, specifically the 4-block output format and responsibilities)*
- **Rule 051 – Cursor Insight:** *(Rules for the Cursor assistant's insight and review behavior – e.g., making automated reviews advisory)*
- **Rule 052 – Tool Priority:** *(Specifies the priority of tools for the assistant – e.g., local code/tools first, then external models, etc., also context guidance)*
- **Rule 053 – Idempotence:** *(Ensure that repeating the same actions yields same results, implement caching of baseline evidence to avoid redoing identical work within a short window)*
- **Rule 054 – Reuse-First:** *(Encourages reusing existing solutions before writing new – e.g., use existing code or partial results if available; also might refer to the CLI reuse-first mode)*
- **Rule 055 – Auto-Docs Sync:** *(Automation to ensure docs are updated – possibly a CI job that fails if code and docs diverge)*
- **Rule 056 – UI Generation:** *(Guidelines for UI code generation using AI, as described in AGENTS.md UI section – essentially the model usage rules)*

- **Rule 057 – Embedding Consistency:** *(Ensure embeddings (vector dimensions, model versions) remain consistent across runs or if changed, then backfilled; consistency between embed and rerank usage)*
- **Rule 058 – Auto-Housekeeping:** *(Automated tasks for repo maintenance – could be auto formatting, dependency bumps, etc., that run on schedule)*
- **Rule 059 – Context Persistence:** *(For the Cursor/assistant, ensure it retains necessary context or can recall context between interactions in a PR; or in pipeline, that context (state) persists where needed – e.g., checkpointer)*
- **Rule 060 – Response Style:** *(Likely guidelines for how the assistant responds in OPS mode – e.g., always use the 4-block format, no extraneous chatter)*

Each rule is documented (the .mdc files contain details) and many are backed by ADRs for rationale. The **Forest Overview** document shows all active rules and marks Rules 047 and 048 as reserved (placeholders to avoid gaps). The project's governance "forest" ensures that these rules are kept synchronized: e.g., Rule-027 (Docs sync) requires that when rules or AGENTS.md are updated, the forest overview is regenerated (hence the generated timestamp in forest_overview.md). This inventory is the map of the project's policy landscape – any change proposed should be checked against these to see if it violates or needs to update one of them.

Known Issues and Gaps (as of Gap Analysis)

Despite the robust design, a few gaps and improvement areas have been identified in the latest pipeline review (**LangGraph Pipeline Gap Analysis**):

- **Automated Schema Enforcement:** While schemas are defined for all stages, the enforcement was not fully automated at every step. The analysis noted that schema validation should run after **each stage** of the pipeline, not just manually or at the end. At the time of review, it was possible that a stage could produce an output that doesn't exactly match the schema (extra/missing fields) and the pipeline might not catch it until later or unless manually checked. The recommendation is to integrate strict JSON schema validation into the pipeline's guard or each agent, so that any deviation from schema **immediately fails the run**. This includes implementing Rule-026 and Rule-039 checks in code: e.g., an automated test that ai_nouns JSON has all fields and no extras, and similarly for graph JSON. Closing this gap will prevent "schema drift" where code and schema definitions diverge unnoticed.

- **Schema Drift & ADR Traceability:** Relatedly, there was a concern about **schema drift** – new fields might be added in code without updating schema files or documenting via ADR. The gap analysis suggests instituting a policy (and possibly a guard check) that any change to an output structure must be accompanied by a schema file update and an Architectural Decision Record entry. They propose the guard agent could check commit messages or PR descriptions for an ADR reference if it detects schema changes. Ensuring this would maintain the SSOT integrity between what's coded and what's documented.

- **Fail-Closed Robustness:** The design calls for *fail-closed* behavior (never proceed on partial or invalid data), but the implementation needs thorough auditing to ensure every agent actually does this. For example, the Noun agent should halt if too few nouns, the Graph builder should halt if any edge refers to a missing noun, etc.. The gap analysis emphasizes verifying that **each agent node validates its outputs before passing to the next**, raising an error if invariants aren't met. This includes things like uniqueness of noun IDs, no zero-weight edges (unless allowed), etc. A systematic audit of each stage's code was suggested to plug any holes where a stage might silently produce incomplete data. The upshot: every stage must either succeed with a valid output or fail – no silent partial successes.

- **LangGraph Utilization:** It was noted that the pipeline could better leverage some of LangGraph's native features. For instance, LangGraph offers built-in support for looping or conditional branching that could simplify logic, and debugging tools (like stepping through nodes, introspecting state) that require structuring the code to be Studio-compatible. One gap is that the pipeline might not yet be fully integrated with **LangGraph Studio** for visual debugging. To use Studio, certain conventions in how the State is managed and how the graph is constructed must be followed. The analysis suggests ensuring the pipeline is **structured in a Studio-friendly way**, so that developers can visualize it and debug with breakpoints in the DAG. Adopting any missing patterns (like using LangGraph's built-in router nodes instead of custom logic) could improve maintainability and clarity.

- **Additional QA Agent:** The findings mention the idea of an interactive "Human-in-the-loop" node or a QA agent that would allow a developer to query the pipeline or double-check outputs during a run [3]. Currently, the pipeline is fully automated, but introducing an optional pause for human review in the middle (especially for critical decisions or to approve proceeding with partial data) was floated as a potential improvement. This is not implemented yet but could be considered for future (especially if using more powerful GPT-4/5 style agents that might benefit from human confirmation on ambiguous cases).

- **Release Metadata:** The gap analysis also suggests enhancing the Release stage to assemble a final metadata block that lists all relevant versions, ADRs, and any rule exceptions during the run. While the Release agent does packaging, it wasn't clear if it logs the governance info. Adding this would help audits – e.g., a release JSON could include "gemantria_version: X, model_versions: Y, ADRs: [1,9,19], rules_exceptions: none". This was identified as a gap that could be closed to improve transparency of each pipeline execution.

- **Make Target Completeness:** Another minor gap was around final output delivery and reproducibility. The analysis mentioned adding additional **Make targets or scripts** to make it easy to reproduce specific parts of the pipeline. For example, a target to regenerate a particular stage's output or to verify a completed run's outputs against expected invariants. While many targets exist (as shown in the orchestrator and analysis usage), the team highlighted some "nuts and bolts" around final output and reproducibility that could be tightened. Possibly this refers to things like a target to compare two runs' outputs (for idempotence check beyond 60 minutes caching), or to package all outputs into a single artifact for release. These are enhancements rather than errors, but they were noted.

- **Testing Gaps:** The project has extensive unit tests, but the gap analysis might have pointed out areas with less coverage, e.g. the need for more integration tests of the full pipeline. One known gap is the lack of tests simulating a multi-run scenario to ensure idempotence and resume functionality – implementing tests for the Checkpointer (starting a run, aborting, resuming) would be valuable. Ensuring that metrics and logging produce expected outputs could also be tested. If not already, these were probably recommended.

- **Model and Data Freshness:** There is an implicit reliance on local models (which could become outdated). A forward-looking gap could be how to update models or data (like Bible DB version) and ensure pipeline still works. While not explicitly in the text we have, a likely consideration is keeping the system updated with new model versions (GPT-4 -> GPT-5, etc.) when they become available and adjusting rules (like Rule-049 anticipating GPT-5 usage). The team might plan for forward compatibility (as suggested by Rule-020 ontology forward-compat and others).

In conclusion, the system is already quite rigorous, but the above gaps highlight the push for **even stricter enforcement and better tooling**: integrate schema checks everywhere, no silent errors, exploit all LangGraph features, and make runs as transparent and reproducible as possible. The development team is addressing these in ongoing work (some recommendations from the gap analysis are likely in progress, such as adding the schema_validator node and guard enhancements, which we saw mention of as "NEW" in AGENTS.md pipeline list). Regular "forest regeneration" (updating the rules/docs) and ADR updates will continue to close these gaps over time, moving the project closer to its goal of an error-proof, self-documenting pipeline for gematria analysis.

Sources: The insights above were drawn from the project's design documents and analysis reports, including *AGENTS.md* (governance and pipeline details), *forest_overview.md* (rules index), *Technical Architecture & Workflow Overview.pdf*, *Single Source of Truth Documentation.pdf*, the *LangGraph Pipeline Gap Analysis.pdf*, as well as schema definitions and the PR template. These comprehensive sources ensure that this reference reflects the authoritative state of the Gemantria project as of late 2025.

---

[1] AGENTS.md
file://file-JTWFGMpMG7bcCfJzsHgs47

[2] forest_overview.md
file://file-2UWsdU4nEZcy7rzC49sp3D

[3] Gemantria LangGraph Pipeline Gap Analysis.pdf
file://file-NqguzVZ7H9D1jEQcXcqnzb