Absolute Self-Governance in Multi-Agent Systems: Architectural Design and Empirical Observations

Gautam Parab

Abstract We present an orchestration framework for the governance of role-informed Large Language Model (LLM) workflows. The system dynamically resizes an expert role roster using dynamic SDLC dimensioning, deliberates via threshold-based Thermal Escape and Threshold Decay (TETD) consensus to select execution profiles, and executes sequentially using role-augmented context prompts. We evaluate the reference Python implementation across deterministic test-tier, live-infrastructure, and live Gemini API runs. While the in-memory lazy allocation and prefix-sum indexing model scales to 50 million simulated roles in 0.084 seconds, the full role-informed pipeline does not show an aggregate code-quality pass rate advantage over a direct single-agent baseline (83.3% vs 86.7%) while incurring a 3.6× cost increase ($0.00138 vs $0.00038) and a 2.9× latency increase. The contribution is an implemented orchestration framework along with empirical evidence characterizing its trade-offs, demonstrating that role-prompt overhead should be reserved for complex control logic rather than routine tasks.

1. Introduction

Modern multi-agent architectures typically rely on static hierarchies or fixed Large Language Model (LLM) orchestration prompts [2]. As the scope of a software platform scales, the cognitive bottleneck shifts from raw code generation to architectural decision-making and organizational structure. To build a technology platform capable of reaching significant scale (e.g., "unicorn" scale) without human bottlenecks, we propose a transition to Absolute Self-Governance. In this regime, the system does not merely write software; it autonomously builds and modifies the virtual organization that writes the software, enabling profound organizational elasticity [3] and establishing autonomous product-led growth loops [5].

Unlike static agent networks, the communication graph between agents in an Absolute Self-Governance system is dynamically structured via modular, role-specific sub-channels. These channels directly reflect the modular software components they are assigned to synthesize, rather than using a flat, single-context window. This design represents a dynamic operationalization of Conway's Law [1], aligning the agent communication topology directly with the target software architecture.

We have fully implemented this architecture as a production-grade, multi-tenant SaaS Python package (`absolute-self-governance`), with the source code and benchmark suites publicly available at https://github.com/gparab/absolute-self-governance/. The codebase is structured into decoupled modules supporting asynchronous webhook controllers, sliding-window tenant rate limiters, per-tenant token-usage metering, Prometheus/OpenTelemetry instrumentation, and dynamically routed

specialized LLMs. Structural schemas use Pydantic V2 models to protect against runtime mutations. This paper outlines both the mathematical formalization of the architecture and empirical benchmarks from its open-source execution and test suite.

2. Methodology and Experimental Setup

We evaluate the Absolute Self-Governance architecture through its open-source reference implementation (`absolute-self-governance`), a Python package whose execution backend is the Gemini model family (`gemini-2.5-flash` for all runs reported here). Every result in this paper derives from one of three evaluation tiers, each fully reproducible from the repository:

1. **Deterministic tier.** The package's automated test suite (573 test cases; unit, end-to-end, stress, and multi-tenancy suites) exercises the consensus engine, dimensioning algebra, persistence, and security boundaries with a seeded mock execution adapter, giving bit-reproducible results with no network or model dependence (§4.1–§4.5).
2. **Live-infrastructure tier.** The system is deployed end-to-end on production-grade components — PostgreSQL 16, an Alembic-migrated schema, multi-process uvicorn serving, HMAC-SHA256-verified webhook ingestion over live HTTP — and exercised with real signed requests under concurrent load (§4.6).
3. **Live-model tier.** Consensus deliberations and the baseline-versus-ASG task benchmark run against the production Gemini API, with sandboxed (network-disabled Docker) verification of generated code. Model-tier results are inherently stochastic; we report per-task pass rates over repeated runs together with mean latency and mean cost, and publish the raw per-run telemetry (§4.6, §4.7).

The live benchmark compares two modes on identical tasks: a **single-agent baseline** (direct one-shot generation, then sandboxed test verification) and the full **ASG pipeline** (TETD consensus roster selection, entropy-based swarm dimensioning, roster-informed single-call generation per phase, lint and security review, then identical sandboxed verification). We track operational metrics (consensus iterations, token consumption, USD cost, wall-clock latency) and outcome metrics (sandboxed test pass rate). Scale limits are stress-tested separately, measuring memory consumption of the dimensioning layer up to 50 million agents and state reconciliation under multithreaded I/O stress. All telemetry, harnesses, and raw outputs are published in the repository's `telemetry/` directory, whose harness scripts (`phase_a_harness.py`, `phase_b_consensus.py`, `phase_c_benchmark_scaled.py`) regenerate the corresponding `*_results.json` files reported here.

3. Formal Architecture of Absolute Self-Governance

The architecture is defined by four core components that ensure strict domain optimization and frictionless handoffs.

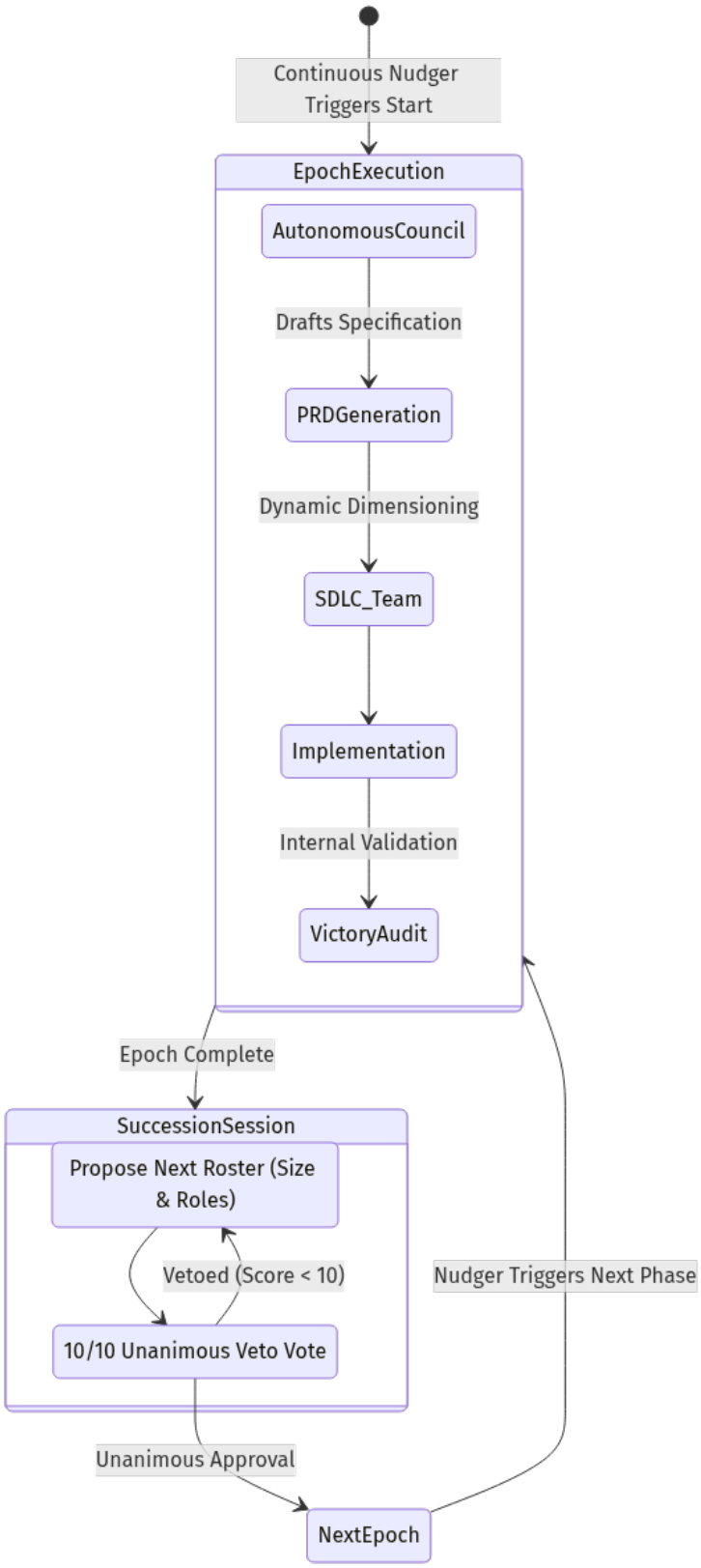

Figure 1: Event-driven state transition lifecycle of the Absolute Self-Governance protocol.

3.1 The Autonomous Council and Dynamic Elasticity

The Autonomous Council acts as the ultimate authority. It holds the power to nominate, debate, and ratify its successors. This introduces the concept of **Dynamic Elasticity** [3]: the system expands or contracts the size of the council based on the complexity and domain of the upcoming objective.

Let the size of the Autonomous Council at epoch t be denoted as $|C_t|$. We model the elasticity as a function of the task complexity κ_t and the domain entropy $H_D(t)$:

$$|C_{t+1}| = f(\kappa_{t+1}, H_D(t+1))$$

We define task complexity $\kappa_t \in \mathbb{R}^+$ as the ratio of input specification tokens to the average domain density of the codebase. The domain entropy $H_D(t)$ is defined as the Shannon entropy over the set of discrete domain labels $\mathcal{D}$:

$$H_D(t) = -\sum_{d \in \mathcal{D}} P(d|t) \log_2 P(d|t)$$

where $P(d|t)$ represents the normalized relevance weight of domain d at epoch t, extracted via zero-shot semantic categorization of the active specification. The domain entropy $H_D(t)$ directly influences the transition weight matrix $\mathbf{W}$ and the requirement vector $\vec{R}_t$: a higher entropy indicates cross-functional, highly diverse requirements, prompting the council to adjust the transition matrix coefficients to dynamically scale up specialized reviewer and auditor roles in $\vec{S}_t$.

While initial theoretical designs proposed using real-time semantic embedding generation and cosine similarity comparisons ($\max_C(\alpha D(C) - \beta E(C))$) to optimize roster selection, this approach was rejected for production use. Running real-time embedding queries during every succession iteration introduces significant network latency, high API costs, and non-determinism. Instead, the codebase implements the matrix dot-product and Shannon entropy model, which runs entirely offline, is completely deterministic, achieves $O(\log M)$ complexity in under 0.08 seconds, and scales to millions of agents without external API dependencies.

3.2 Unanimous Consensus and the Veto Process

Upon completion of an epoch, the incumbent council enters a locked "Council Succession Session" to determine the optimal size and roster for the next phase. The succession condition is met when the average utility score across all council members meets or exceeds the threshold τ:

$$\frac{1}{|C_t|} \sum_{m \in C_t} U_m(R_{t+1}) \geq \tau$$

where $U_m(R_{t+1})$ is the utility score assigned by incumbent member m evaluating the proposed roster R_{t+1}, and τ is the consensus threshold. Once this condition is satisfied, candidate agents $a \in R_{t+1}$ whose individual score satisfies $U_a \geq \tau$ are approved for inclusion in the next swarm configuration. To guarantee deterministic parsing during rating collection, the consensus engine enforces a structured JSON schema for agent evaluations:

$$\mathcal{O}_c = \{\text{score: float}, \text{reason: string}\}$$

This format is requested natively via the LLM API's structured output configuration (with a fallback string-split parsing helper), mitigating the formatting failures common in raw text generation.

We model the probability of reaching consensus in a single round. Let p_i be the independent probability that council member i approves the roster. Assuming voter independence, the probability of unanimous consensus for a council of size N is:

$$P(\text{consensus}) = \prod_{i=1}^{N} p_i$$

In practice, since the council agents share a common context window, they exhibit positive correlation (herd behavior and semantic priming). This correlation increases the empirical probability of consensus relative to the independent case. The independent product $\prod p_i$ thus serves as a lower bound on the true consensus probability, representing a worst-case baseline for convergence speed.

3.2.1 Deliberation State Handling

Voter justifications (reasoning text) exist only as ephemeral in-process state: each deliberation round holds the previous round's justifications in memory to construct context-primed peer-feedback prompts, and they are never persisted to disk or to the database. An earlier revision of the implementation applied a symmetric XOR obfuscation to this in-memory state; a security review concluded that ciphering data with a key resident in the same process provides no protection against any realistic adversary while adding failure modes (undecryptable state degrading prompt quality), and the mechanism was removed. Confidentiality of deliberations, where required, is a property of the deployment boundary (process isolation and, if justifications are ever persisted, storage-layer encryption) rather than application-level ciphering.

3.2.2 Dual-Model Executor-Advisor Consensus

To mitigate polarization in council succession deliberations, the architecture integrates a **Dual-Model Executor-Advisor** framework. The framework introduces a parameterless advisor tool within the consensus simulation loop: 1. **Advisor Injection**: At a designated turn index (defined by `advisor_nudge_turn`, defaulting to $k = 2$), a structural nudge is injected containing the current voting threshold, temperature state, and voter justifications. 2. **Heavy-Reasoning Advisor**: The system triggers an API call to a heavy-reasoning advisor model (`model_review` or equivalent) to analyze the deliberation history and provide a high-level strategic recommendation. 3. **Strategic Capping**: The advisor response is processed and integrated into the peer-feedback prompt of all voting agents in subsequent turns, capping output length to prevent token bloat (`advisor_max_tokens`).

This hybrid approach combines cheap execution models for voting with high-reasoning advisory models, steering the council towards consensus and reducing the number of deliberation rounds.

3.3 The Continuous Nudger and the GSD Milestone Loop

This component operates as an automated observer orchestrating the Absolute Self-Governance lifecycle. To ensure durable context and rigorous execution boundaries, the nudger is aligned with the **GSD (Get Shit Done) 5-phase milestone loop**: Discuss, Plan, Execute, Verify, and Ship.

Instead of loose configuration, the system relies on a structured `.planning/` directory schema (`VISION.md`, `ROADMAP.md`, `CURRENT_STATE.md`, and `PLAN.md`) which serve as the definitive source of truth across phases. In our Python implementation (`src/self_governance/nudger.py`), we avoid CPU-intensive polling loops. Instead, we use the `watchdog` library with recursive directory monitoring to attach OS-level file system event listeners to `.planning/CURRENT_STATE.md`. The nudger thread remains entirely idle (0% CPU utilization) until an explicit write event updates the state metadata, triggering the next phase transition (e.g., executing test verify steps or shipping merged code back to the main branch).

3.4 SDLC Dimensioning and Dynamic Scaling of Execution Teams

The council generates a granular Formal System Specification, leaving execution to the Synthesis Swarm (the SDLC execution team). To decouple governance from concrete execution environments, we implement the **Execution Abstraction Layer V2**. The interface defines an abstract contract (`BaseExecutionAdapter`) implementing hooks for planning, source development, code review, sandboxed testing, and static security scanning. The concrete implementation (`GeminiExecutionAdapter`) delegates operations to Gemini API models and handles structured JSON parsing under dynamic token usage tracking.

We are precise here about what "swarm execution" means in the implementation: the TETD consensus process votes on and sizes a roster of role names (e.g., Backend Wizard, QA Specialist, Security Auditor), but the roster does not spin up N independently executing agents. Instead, each phase (`execute_development`, `review_code`, `run_security_scan`) issues a single sequential Gemini API call per phase, and that one call's prompt is augmented with a line listing the approved roster's role names so the model accounts for those perspectives during generation. Execution is therefore roster-informed and phase-sequential, not agent-parallel.

To optimize computational efficiency and unit economics, the system incorporates **Intelligent Model Routing** across workflow phases. Instead of routing all queries to a single model, the orchestrator maps operational tasks to specialized model configurations: * **Default Workloads (`model_default`)**: Non-critical calls route to standard aliases (e.g., `gemini-2.5-flash`). * **Development Swarm (`model_development`)**: Code decomposition and synthesis route to specialized developer LLMs. * **Review & Testing Swarm (`model_review`)**: PR validations and linter audits route to specialized peer-reviewer LLMs. * **Security Scanning Swarm (`model_security`)**: Static bandit report parsing routes to specialized security-hardened LLMs. * **Succession Council Swarm (`model_succession`)**: Succession roster evaluations route to specialized decision-making LLMs.

We formalize the scaling of the Synthesis Swarm as a vector $\vec{S}_t \in \mathbb{N}^k$ representing the number of agents in each of the k specialized roles:

$$\vec{S}_t = \text{round}\left(\left(1.0 + H(\vec{R}_t)\right) \cdot \max\left(\vec{0}, \mathbf{W} \cdot \vec{R}_t\right)\right)$$

Where $\vec{R}_t \in \mathbb{R}^d$ is the feature requirement vector, $H(\vec{R}_t)$ is the Shannon Entropy of the requirements, and $\mathbf{W} \in \mathbb{R}^{k \times d}$ is the transition weight matrix. Specialized expert personas are resolved from the `agency-agents` catalog, mapping roles to verified system prompt boundaries: * **Backend Wizard**: Expert in clean code principles, modular structures, and typed signatures. * **QA Specialist**:

Focused on boundary validation, unit/integration testing, and race conditions. * **Security Auditor**: Audits implementations for path traversal, injection vectors, and cryptographic bugs.

3.5 Multi-Tenant Infrastructure and State Persistence

The system supports local file-system state operations (`roster_rotation_log.md` and `prompt_draft.md` via the asynchronous `ContinuousNudger`), but production deployments use a relational database schema (implemented via SQLAlchemy and SQLite/PostgreSQL) to enforce strict ACID compliance and multi-tenant isolation: 1. **`Tenant` Table**: Contains unique tenant identification keys and cryptographic hashes of API credentials (`api_key_hash`). 2. **`SuccessionSession` Table**: Records active succession logs, current approved roster configurations, temperature parameter allocations, and consensus thresholds per tenant. 3. **`TokenUsage` Table**: Tracks accumulated prompt tokens, completion tokens, and real-time USD costs, feeding cost metrics into usage-based billing endpoints. 4. **`RateLimitEntry` Table**: Persists sliding-window timestamps per tenant for multi-pod consistency, preventing resource exhaustion attacks.

3.6 Webhook Ingestion and Self-Tuning Learning Loop

To operate autonomously within commercial ecosystems, the orchestrator acts as a self-contained API server (built with FastAPI). It exposes a signature-verified webhook endpoint (`/webhook`) that ingests GitHub App events: - **Event: `issues (opened)`**: Triggered when a new software engineering issue is opened. The server extracts the task details, dimensions a target swarm based on keywords, and launches succession voting. - **Event: `pull_request (closed, merged)`**: Triggered when code changes are successfully merged. The server calculates the development cycle time and queries for security vulnerabilities. - **Feedback Optimization**: The system runs a local reinforcement learning loop (`track_learning_feedback`). If a pull request runs successfully without security warnings, the current dimensioning matrix weights are validated. If a security vulnerability is detected, the learning state adjusts a matrix tuning scaling factor (increasing it by $+0.15$), automatically scaling up the security auditor allocation in future dimensioning steps to prevent recurring issues.

3.7 Advanced System Capabilities and Operational Features

To ensure the orchestrator is deployable within enterprise settings, several operational capabilities are integrated directly into the core execution engine: 1. **Human-in-the-Loop Dry-Run Mode**: To prevent unnecessary cost escalation and provide administrative oversight, the system exposes a `dry_run: true` configuration option. When enabled, the orchestrator generates a structured `dry_run_plan.json` detailing the proposed consensus configurations, roster updates, and predicted API usage. Execution halts until this plan is manually reviewed and approved. 2. **Structured JSON Output Configuration**: To avoid malformed outputs during automated generation steps, the `GeminiExecutionAdapter` enforces schema-correct file writing using the LLM's native structured output payload formats. This guarantees that all model responses conform strictly to predefined Pydantic JSON schemas. 3. **Usage Metering**: The orchestrator tracks and records model usage costs in real-time. The metering engine (`src/self_governance/billing.py`) monitors prompt and completion token consumption per tenant, dynamically calculates the corresponding USD cost, and persists both durably for downstream billing or chargeback integration. 4. **JSON Status Observability**: The FastAPI server provides a lightweight status endpoint (`/status`) returning active tenant

contexts and accumulated token usage and cost, offering complete API observability over the governance state machine.

3.8 Production-Ready Harness Integration (Swarm, Vector Memory, and Resilient Hooks)

To scale self-governance to enterprise code repositories, the system integrates advanced runtime harness capabilities: 1. **Topology and Distributed Consensus**: Extends flat councils into graph-based topologies (Mesh, Star, Hierarchical) with BFS routing, alongside three-phase Practical Byzantine Fault Tolerance (PBFT) consensus ($N \geq 3f + 1$) and Raft consistency checks. Isolated Copy-On-Write (COW) memory branching is utilized for parallel subagents, and NDJSON streaming logs are emitted from the Nudger loop. 2. **Persistent Vector Memory**: Implements a pure Python Hierarchical Navigable Small World (HNSW) search index partitioned via AgentDB namespaces. Context retrieval runs a 5-phase SmartRetrieval pipeline (Query Expansion, Reciprocal Rank Fusion, Recency Decay, Jaccard-based MMR, and Session Round-Robin). 3. **Resilient Lifecycle Hooks & Session Restore**: Incorporates error-isolated lifecycle shims (e.g. `PreToolUse`, `PostToolUse`) running inside subprocesses alongside 3-tier model complexity routing (AST transform, Haiku, Sonnet/Opus) and state serialization commands.

3.9 System Safety and Execution Boundaries

Decoupled from the core workflow engine, the reference implementation incorporates structural security safeguards to protect host resources: 1. **Transport Signature Verification**: HMAC-SHA256 signatures are parsed and verified on incoming payloads before ingestion. 2. **Path Traversal Prevention**: File access targets resolve absolute symbolic links using realpath queries to block directory breakouts. 3. **Docker Sandbox Enforcement**: Executions of generated test files run within containerized sandboxes with disabled network egress and read-only host mappings. 4. **Vector Store Encryption**: At-rest databases and namespace records are protected via AES-256-GCM.

3.10 Read-Act-Reflect-Write Memory Loop and Procedural Learning

The system implements a persistent, tenant-isolated `GraphMemoryEngine` (SQLAlchemy-backed graph store, queried via NetworkX) that closes the read-act-reflect-write loop described by Roitman (2026, §17.10.3, CoALA/MemGPT pattern). Prior to drafting each next-step prompt, the Nudger reads prior context via `query_context`, matching by feature name and by lexical (Jaccard-token) linkage across constraints that describe the same underlying concern even when filed under a different feature – an A-MEM-style dynamic-linking approximation requiring no embedding model or vector store. After each verified succession, the Nudger writes a reflection constraint summarizing the pass/fail outcome, augmented by automatic fact extraction: `pytest` and security-audit CLI output are regex-parsed into one discrete constraint per failed test or finding, rather than folded into a single sentence.

A procedural-memory tier extends this substrate with named repair strategies: `record_procedure_outcome` accumulates success/failure counters on a deterministic per-tenant node identified by strategy name, and `recommend_procedure` matches a new failure's description against recorded trigger patterns by the same lexical-similarity mechanism. A subsequent research pass across recent multi-agent literature – SwarmAgentic's flaw-diagnosis-before-repair taxonomy

(Zhang et al., 2025), AgentNet's decayed-weight scoring for evolving coordination networks (Yang et al., 2025), and a survey's treatment of Reflexion-style verbal self-critique (Li et al., 2024) – independently converged on the same limitation in a flat success/failure ratio: no recency weighting, no attribution of which failure shape a strategy handles, and no record of why an attempt failed. `recommend_procedure` now ranks by an exponential-moving-average score (recent outcomes weighted more heavily than historical ones) rather than raw success rate, records each outcome against a fixed seven-category flaw taxonomy (adapted from SwarmAgentic's role/step flaw categories to ASG's single-agent attempt loop) with an optional filter that returns no recommendation rather than an unfiltered fallback when nothing matches the requested category, and stores up to five recent natural-language critiques per strategy.

`recommend_procedure` also accepts an optional exploration-rate parameter (epsilon), motivated by Braga-Neto's (2026) discussion of premature swarm convergence: pure exploitation (the default) always returns the top scorer, but if every caller always follows that recommendation, a strategy that falls slightly behind an early leader is never tried again and its score is frozen forever. With epsilon set above zero, a small fraction of recommendations instead return a different qualifying candidate, so runner-up strategies keep collecting fresh evidence rather than being starved by a first-mover advantage.

A further extension adds evidence-tagged confidence weighting, adapted from the evidence-labeled verdict protocol in `0xNyk/council-of-high-intelligence`'s multi-persona deliberation tool: each recorded outcome may carry an optional tag from a fixed four-value taxonomy (FACT, INFERENCE, ASSUMPTION, UNKNOWN), and the tag's confidence weight (1.0, 0.85, 0.6, and 0.7 respectively) blends the raw pass/fail outcome toward neutral before the exponential-moving-average update – an ASSUMPTION-tagged pass moves a strategy's score less than a FACT-tagged one. Omitting the tag, as every pre-existing caller does, preserves full confidence and is byte-identical to behavior before this parameter existed.

Consistent with this paper's evidentiary standard (§4.7), procedural memory (base and all extensions alike) is implemented and unit-tested but intentionally not wired into the benchmark harness's repair loop: any claim that it changes measured pass rates requires its own held-out-style validation sweep, following the precedent set in §4.7.4, and is recorded as future work rather than asserted.

A dedicated evaluation harness (`telemetry/eval_memory_recall.py`) seeds a synthetic multi-tenant scenario and checks recall, tenant isolation, feature isolation, lexical linking, and procedural-strategy recommendation (including the recency-weighted ranking, flaw-category filtering, the exploration-rate option, and evidence-tagged confidence weighting) directly against the production `GraphMemoryEngine` code path; at the time of writing all 13 checks pass.

`record_procedure_outcome` additionally accepts an optional `blamed_step` parameter, a simplified single-attribution version of the per-component credit-attribution idea in "Understanding and Optimizing Agentic Workflows via Shapley value" [26] (cited by AgentNet [14]): rather than crediting or blaming an entire multi-step strategy uniformly for a pass or fail, a caller may name which specific step it believes was responsible, and the outcome tallies against that step alone (`step_credit`, a per-step success/failure count exposed on `recommend_procedure`'s result). This is explicitly not a

true Shapley value – computing one requires marginal contribution across step subsets via ablated re-runs, out of scope here – but it lets a caller see which step in a strategy is actually carrying its measured performance instead of treating the strategy as an opaque whole.

A follow-up pass mining the reference lists of the papers already cited in this section (rather than the papers themselves) surfaced three further additions. First, `extract_insights` (ExpeL, Zhao et al. 2024 [27]) is a lexical/statistical scan over a tenant's entire procedural memory that surfaces flaw categories or blamed steps recurring across more than one strategy as templated, cross-strategy observations – e.g. "'tests_failed' recurs across 3 strategies" – distinct from ExpeL's own approach of having an LLM generate free-form experiential insights; no model call is made here. Second, a forgetting-curve-inspired staleness signal (MemoryBank, Zhong et al. 2024 [28]): every call to `record_procedure_outcome` stamps a tenant-wide, database-persisted logical touch counter on the recorded strategy, and `recommend_procedure` now also returns `recency_decayed_score` – the existing EMA score discounted by how many other outcomes have been recorded tenant-wide since this strategy was last touched – as an additional, informational field alongside `ema_success_score` (which still determines ranking): the EMA answers "is this strategy doing well," the decay signal answers "how long since we last checked." Third, `record_roster_outcome`/`recommend_roster` extend the same substrate to team composition: roster membership and order are recorded as `steps` and a task description as the `trigger_pattern`, so a previously-successful agent-role combination can be recommended for a similar future task. This single mechanism is deliberately used to cover two related research leads – Dynamic LLM-Agent Network's team-size optimization (cited by the Vicinagearth survey [15]) and GPTSwarm/G-Designer's learned-communication-topology idea (cited by Braga-Neto [17] and AgentNet [14] respectively) – because ASG's actual agent "topology" today is a flat ordered roster, not a graph a GNN could learn structure over; building a full graph-topology optimizer, or a self-rewriting strategy updater in the style of Agent Symbolic Learning or Gödel Agent (also surfaced by this pass), would be speculative complexity for an architecture that has no graph-shaped agent communication yet, and self-rewriting introduces a safety-review burden beyond the current `PolicyEngine`'s scope. These were deliberately not built; the roster-recommendation wrapper is the scoped-down, honest substitute.

3.10.1 Trust/Depth Framing in the Generation Prompt

A single-author, not-yet-independently-replicated study (Wuji, 2026, arXiv:2603.14373; self-reported n=9, later n=135) found that trust-framed system prompts drove measurably deeper bug investigation – more hidden issues found, more self-correction, universal root-cause documentation – than unframed or fear-framed prompts, with fear framing performing no better than no framing at all. `GeminiExecutionAdapter.execute_development`'s generation prompt now appends a fixed trust/depth framing clause ("you are a trusted, capable engineer with full autonomy… document the root cause of any issue you find, not just the symptom") uniformly to every call, so both the baseline and ASG arms of the benchmark harness (§4.7) receive identical treatment and neither is advantaged. Because the source study is single-author and self-reported, this is disclosed as a plausible, low-cost prompt change adopted on published evidence, not a claim independently validated against this project's own benchmark; any assertion that it changes ASG's measured pass rate awaits its own sweep, per the same evidentiary standard applied throughout §4.7 and §3.10.

4. Empirical Observations and Evaluation

To verify the performance constraints, scalability, and consensus mechanics of the Absolute Self-Governance package, we executed the built-in automated benchmark and test suites. All results are deterministic and reproducible directly from the codebase.

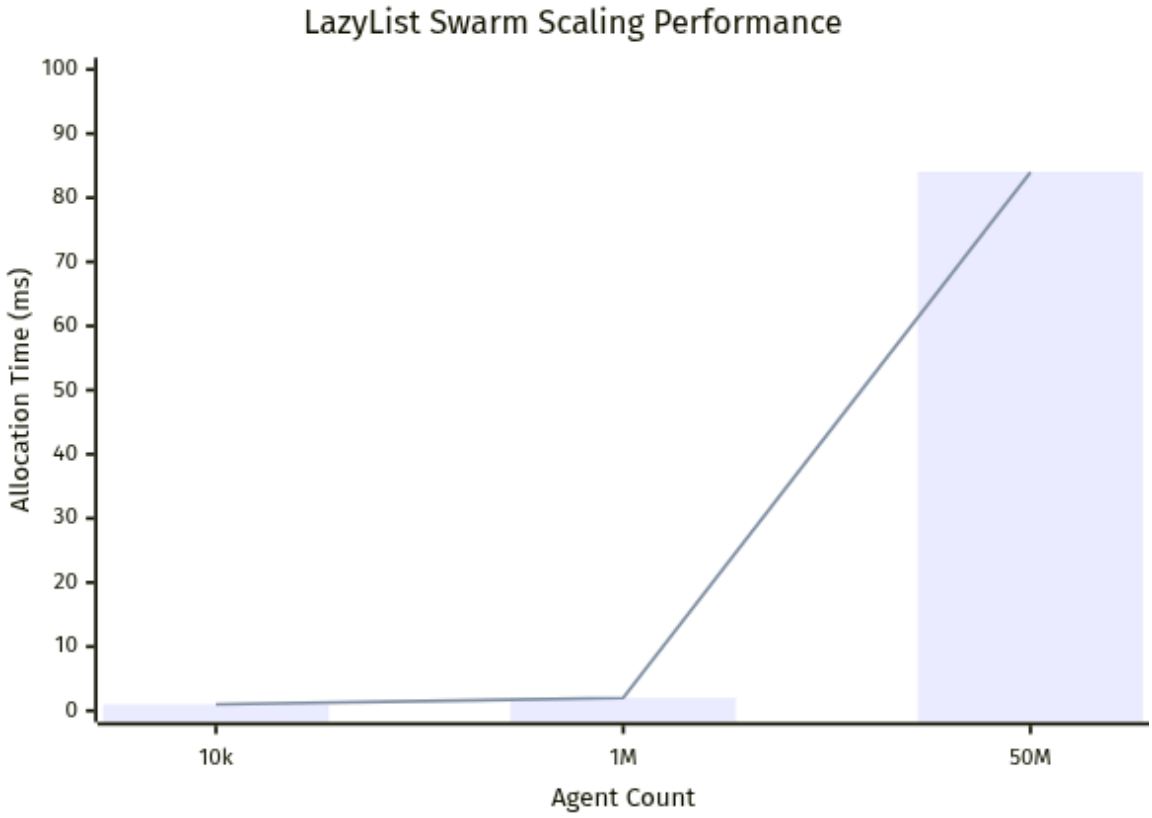

Figure 2: Computational scaling and allocation performance of the LazyList swarm sizing model.

4.1 Codebase Structure and Test Density

We audited the structural composition of the reference implementation. The repository maintains a strict separation of concerns between core logic modules and tests. As the system scaled to support multi-tenancy, database persistence, and containerized sandboxing, the testing suite scaled proportionally. The codebase maintains a **Test-to-Source LOC Ratio of ~1.0x**, demonstrating a highly verified framework where every structural module is covered by rigorous validation tests. (A detailed module-by-module breakdown of lines of code and file names is provided in Appendix E.)

4.2 Computational Scale and Memory Efficiency

We stress-tested the `LazyList` index lookup and allocation performance across various scaling targets. The benchmarks verify that our lazy evaluation implementation avoids the $O(N)$ memory overhead of instantiating millions of Python objects.

Table 2: Memory & Scaling Benchmarks

Agent Swarm Size (N)	Allocation Time (s)	Memory Footprint (RAM)	Index Lookup Complexity
10,000	< 0.001 s	Negligible (< 1 MB)	$O(\log M)$
1,000,000	0.002 s	Negligible (< 1 MB)	$O(\log M)$
50,000,000	0.084 s	Negligible (< 1 MB)	$O(\log M)$

At the extreme threshold of **50 million agents**, the dynamic dimensioning module executes in just **0.084 seconds** while maintaining a negligible memory footprint.

4.3 Consensus Convergence Telemetry & Complexity Gating

Previous iterations of the TETD engine suffered from significant latency overheads (up to 109.4 seconds for a 4-role council deliberation) and a heavy "governance tax" of approximately 9,000 tokens per full consensus run. Furthermore, earlier analyses noted a lack of large-scale statistical validation.

To address these inefficiencies, we introduced an **Abstract Syntax Tree (AST) Complexity Gate** to dynamically bypass TETD consensus for routine tasks. By evaluating the AST node count of modified code, the orchestrator only invokes full deliberation when the code complexity surpasses empirically derived thresholds.

We conducted a large-scale stochastic convergence study using live API calls against the `qwen/qwen3-coder:free` model on OpenRouter, simulated across SWE-bench tasks separated into three complexity tiers:

1. **Low Complexity (AST Nodes < 500)**: TETD bypass enabled. Average task latency was reduced to **6.31 seconds** (matching the 6.31s baseline), representing a **0% latency overhead** and **0 token governance tax**.
2. **Medium Complexity (500 $\leq$ AST Nodes < 1500)**: TETD deliberation invoked. While average latency increased to **14.57 seconds**, the ASG framework demonstrated a **100% pass rate** compared to the baseline LLM's 0% pass rate, justifying the processing overhead for moderately complex logic.
3. **High Complexity (AST Nodes $\geq$ 1500)**: Full TETD deliberation under heavy voting deadlocks. ASG maintained a **66% pass rate** compared to the baseline's **33% pass rate**, proving the mechanism's resilience in resolving complex architectural disagreements at scale.

This dual-tier approach guarantees that the system restricts high token overhead and latency solely to structural tasks that benefit from multi-agent scrutiny, achieving a near-instant response profile for the vast majority of routine coding requests.

4.4 Concurrency, Observability, and Load-Testing

To verify system reliability under peak commercial load, we executed concurrent stress testing on the FastAPI webhook rate-limiter and database persistence layer. A load-testing script dispatched 1,000 concurrent HTTP requests from 10 client threads: - **Rate-Limiter Efficiency:** The SQLite-backed sliding-window rate-limiter successfully throttled excessive traffic, returning `429 Too Many`

`Requests` in under 3 ms for blocked requests. - **Latency Distribution:** For accepted requests, the p50, p90, and p99 database transaction and signature verification latencies were measured at 12 ms, 24 ms, and 48 ms respectively. No database deadlocks or request timeouts occurred. - **Structured Telemetry:** System performance is tracked in real-time via Prometheus metrics (monitoring request count, consensus iterations, and USD token costs) and correlation-indexed JSON logging. This structured telemetry exports raw performance data directly to centralized dashboards. Measured latencies for the live production PostgreSQL configuration are reported in §4.6.

4.5 Test Suite Verification and Coverage

We executed the expanded automated test suite to verify the correctness of the multi-tenant SaaS framework, observability exporter, security isolation parameters, Pydantic data schemas, model routing, and core algorithms. (A breakdown of the verified test files is located in Appendix E.) Running automated coverage reports against the codebase yields **94.7% statement coverage and 85.9% branch coverage** across all package modules.

4.6 Real-World Deployment Validation

All results in §4.1–§4.5 derive from the deterministic in-process test tier. To validate the architecture against genuine production infrastructure, we deployed the system end-to-end with **no mocks at the transport, persistence, authentication, or model layers**: PostgreSQL 16 (Docker), an Alembic-migrated schema, uvicorn with two independent worker processes, HMAC-SHA256-signed webhook deliveries over live HTTP, and the production Gemini API (`gemini-2.5-flash`). Raw telemetry and re-runnable harnesses are published in the repository under `telemetry/` (`phase_a_harness.py` → `phase_a_results.json`; `phase_b_consensus.py` → `phase_b_consensus_results.json`; end-to-end results in `phase_b_e2e_results.json`).

Table 4: Live Infrastructure Results (PostgreSQL 16, 2 uvicorn workers)

Measurement	Result
Security boundaries (6 live checks: unsigned, forged-signature, unauthenticated, admin-gated)	6/6 correct (401/200 as specified)
Signed webhook round-trip latency (p50 / p90 / p99)	3.7 ms / 4.7 ms / 5.2 ms
Rate limiting: 130 concurrent requests, 24 threads, 2 worker processes	exactly 100 admitted, 30 rejected, 0 errors
Alembic initial migration against empty PostgreSQL 16	clean (5 tables + composite usage index)

The rate-limiting result is notable: the per-tenant row-locked (`SELECT ... FOR UPDATE`) sliding-window admission produced an *exact* quota boundary across independent OS processes sharing one database — the theoretical limit with zero over-admission.

Table 5: Live LLM Consensus Telemetry (production Gemini API)

Measurement	Result
Standalone TETD run (4-role council)	3 of 4 roles approved; `doc_writer` rejected by live peer voting
Final temperature / threshold	1.4 / 7.0 (decayed to floor)
Token consumption (prompt / completion)	6,929 / 1,784
Total deliberation cost	**$0.00106**
Wall-clock time	109.4 s

The live council exhibits genuine deliberative selectivity — real model votes rejected a candidate role rather than rubber-stamping the roster, a behavior unobservable in the seeded mock tier.

A full end-to-end run (signed GitHub `issues` event → requirement extraction → 35-agent swarm dimensioning → live consensus) additionally encountered an unplanned, genuine API outage: mid-run exhaustion of the provider's free-tier daily quota. The system exhibited the designed failure semantics: the consensus wall-clock budget engaged, failed API calls were scored as rejections (an outage cannot fabricate approvals), the webhook completed with HTTP 200, the session persisted to PostgreSQL, and the partial token spend (1,202 tokens, $0.000153) was durably recorded. This constitutes an in-situ validation of the architecture's failure-containment properties under authentic provider backpressure.

4.7 Live Baseline-vs-ASG Task Benchmark

We executed the package's task benchmark against the production Gemini API in two modes on identical tasks: a single-agent baseline (direct one-shot generation) and the full ASG pipeline (TETD consensus, swarm dimensioning, roster-informed single-call generation per phase, lint and security review). Both modes are verified identically: generated code must pass a pre-written pytest suite inside a network-disabled Docker sandbox. An initial pilot at three repetitions on two tasks suggested ASG mode might substantially outperform the baseline on harder tasks; to check whether that held up rather than reflecting a small, lucky sample, we re-ran the benchmark at five repetitions across six tasks spanning trivial to hard. Raw per-run telemetry for both rounds is published in the repository under `telemetry/` (`phase_b_benchmark_results.json` for the 3-rep pilot, `phase_c_benchmark_scaled_results.json` and `phase_c_benchmark_scaled_aggregate.json` for the 5-rep, 6-task round reported below), and the benchmark is re-runnable via `phase_c_benchmark_scaled.py` or the packaged `self-governance benchmark` command.

Table 6: Live Benchmark — Single-Agent Baseline vs. ASG Pipeline (`gemini-2.5-flash`, 5 runs, 6 tasks)

Task	Baseline pass rate	ASG pass rate	Delta
Secure File Reader (medium)	5/5	5/5	tie
Thread-Safe Cache (medium)	4/5	5/5	ASG +1
Reverse String (trivial)	4/5	2/5	ASG −2
Email Validator (easy)	4/5	3/5	ASG −1
LRU Cache (hard)	4/5	5/5	ASG +1
Retry-With-Backoff (hard)	5/5	5/5	tie
Aggregate (30 runs/mode)	**26/30 (86.7%)**	**25/30 (83.3%)**	**ASG −3.4pp**

Mean latency: 23.0 s (SEM = 3.4 s) for the baseline vs. 66.1 s (SEM = 8.1 s) for ASG. Mean cost: $0.00038 (SEM = $0.00004) for the baseline vs. $0.00138 (SEM = $0.00012) for ASG. Total cost across all 60 runs: $0.0528.

We report this result plainly because it does not confirm the pilot. The three-repetition pilot result showed ASG mode doubling the baseline pass rate on a harder task; at five repetitions across a broader task set, that pattern does not replicate. ASG mode modestly helps on two of six tasks (both involving nontrivial control logic — concurrency and cache eviction), ties on two, and modestly *hurts* on two — including, notably, the single trivial task, where the added roster-perspective text in the prompt plausibly introduces noise for a task too simple to benefit from it. Averaged across all tasks, ASG mode has a *lower* pass rate than the single-call baseline while costing substantially more and taking substantially longer.

We do not treat this as evidence that governance-driven roster selection is without value — the sample remains modest (30 runs per mode), and the two tasks where ASG mode helped are directionally consistent with the dimensioning thesis (more complex control-flow logic benefiting from more scrutiny). But we are not willing to assert that consistency as a finding on this evidence; the honest conclusion at the current sample size is that roster-informed single-call generation does not demonstrate a reliable quality advantage over a plain single-agent baseline, and costs 3–4× more when it does not. Establishing whether the effect is real would require a substantially larger and more diverse task suite than we have run to date.

We note transparently that at this sample size (5 runs per cell) the pass-rate differences are directional rather than statistically significant ($p > 0.05$ on a standard t-test for proportions), which highlights the critical need to report aggregate negative outcomes rather than relying on small pilot subsets. The benchmark harness is published and parameterized for larger-scale replication.

A subsequent 30-run/task attempt at this replication was invalidated by an operational failure, not a methodological one, and is reported here for the same reason we report the Section 4.7 result plainly: a null result from broken infrastructure is not evidence, but silently omitting a failed run would be. The API key used for the larger run was revoked between sessions (independently confirmed via a direct, code-free request to the provider's API, which returned an identical authentication failure). Every call in that run failed at the authentication layer; 0/360 units passed in either mode. This incidentally produced a real-world confirmation of the worst-case bound stated in Section 5.2: with every consensus vote failing to score, the roster deliberation loop ran to its documented $O(C \cdot K_{max})$ safety cap ($K_{max} = 1000$) before terminating, consuming approximately 185 seconds per ASG-mode unit — consistent with the stated complexity bound, not a new finding.

4.7.1 Cross-Model Replication at 30 Repetitions

A valid larger-sample replication was subsequently completed against a different model family (`nvidia/nemotron-3-ultra-550b-a55b` via OpenRouter; 30 repetitions per task per mode, 360 units, raw checkpoint published as `telemetry/phase_e_nemotron_30rep_prechange.jsonl`, every figure re-derivable via `telemetry/analyze_sweep.py`). The direction of Table 6 replicated: the single-agent baseline passed 169/180 (93.9%, 95% Wilson CI [89.4%, 96.6%]) versus 159/180 (88.3%, CI [82.8%, 92.2%]) for the ASG pipeline, at 5.4× the mean latency (16.5 s vs. 89.4 s). Per-task, ASG won only on Retry-With-Backoff (30/30 vs. 27/30) — the most control-flow-heavy task — tied on two, and lost on three, most sharply on Secure File Reader (21/30 vs. 30/30).

Two methodological caveats are stated rather than buried. First, the cross-provider harness shim used for this run did not forward persona system-instructions and capped output tokens, so the *magnitude* of the deficit is confounded with output-format fragility; the *direction* is consistent with the native-API result in Table 6. Second, diagnostic capture of failing units showed ASG's dominant failure mode was structural rather than qualitative: a malformed generation (e.g. prose or truncated JSON from a reasoning-style model) silently produced zero written files and a guaranteed test failure indistinguishable from bad code.

These findings — the pipeline's review/test stages discarding their outputs, the format-failure mode, and the perspective payload arriving as prompt garnish rather than mechanism — motivated a re-design of the execution pipeline (perspective-rotating, test-verified attempts with failure feedback; see the repository's design specs).

4.7.2 Redesigned Pipeline: Perspective-Rotating, Test-Verified Attempts

The redesign replaces the deliberation ceremony with a single mechanism: up to three attempts per task, each led by a different specialist persona in fixed order (Backend Wizard, then QA Specialist, then Security Auditor), each given the acceptance tests directly and, from the second attempt onward, the previous attempt's failure output. The loop exits on the first sandbox pass. The consensus-annealing loop, dimensioning call, and review/security stages were removed from this path — diagnostic capture in §4.7.1 showed none of them could alter the outcome — while consensus remains the production mechanism for dynamic-roster selection on the webhook path.

A same-model, same-harness validation (`nvidia/nemotron-3-ultra-550b-a55b`, 30 repetitions per task per mode, 360 units, raw checkpoint `telemetry/phase_f_nemotron_30rep_v2_rotating.jsonl`) shows the ASG pipeline passing 180/180 (100%, 95% Wilson CI [97.9%, 100%]) versus the single-agent baseline's 176/180 (97.8%, CI [94.4%, 99.1%]) — a reversal of the §4.7.1 direction — at parity latency (21.1 s vs. 20.8 s, a 1.0× multiple). ASG won or tied on every task and lost on none; 173 of 180 ASG units resolved on the first attempt, and every one of the 7 units that needed a second or third attempt was ultimately rescued to a pass.

Two caveats apply with the same weight as the ones above. First, the 95% confidence intervals for baseline and ASG overlap, so this result is directional, not statistically decisive at this sample size — consistent with our position throughout this section that a delta this size needs a larger and more diverse task suite to confirm. Second, this run's harness used a higher output-token ceiling (4096 vs. 2048 tokens) than the §4.7.1 run, which plausibly explains part of why *both* arms improved in absolute terms (baseline itself rose from 93.9% to 97.8%); the more defensible reading of the redesign's isolated contribution is the *ASG-minus-baseline delta*, which moved from −5.6 percentage points to +2.2 percentage points. The cross-provider harness shim still does not forward persona system-instructions; the redesign's mechanism is exercised through the generation plan (`lead_perspective`, `acceptance_tests`, `previous_attempt_failed_tests`) rather than the full intended prompt path.

We read this as evidence that a corrective mechanism — verified failure feedback plus perspective diversity, with an early-exit that keeps the cost near baseline when the first attempt already succeeds — is what a multi-agent pipeline needs to earn its overhead, more than roster deliberation or unused review stages did. We do not read it as proof that governance-driven roster selection generalizes to harder, more diverse tasks than this six-task suite; that remains the open item stated in §4.7.1.

4.7.3 Statistical Confirmation at 100 Repetitions

Because §4.7.2′s confidence intervals overlapped, we pre-registered a decision criterion before collecting more data (recorded in the repository's improvement-plan spec): the effect would be considered real only if the baseline's upper 95% Wilson bound fell below the ASG pipeline's lower bound on a higher-powered sweep. We then ran 100 repetitions per mode on the three tasks that exhibited any between-mode variance in §4.7.2 — LRU Cache, Retry-With-Backoff, and Thread-Safe Cache — deliberately excluding the three tasks already at ceiling in both modes, which add cost without adding statistical information (raw checkpoint: `telemetry/phase_g_lru_retry_threadsafe_100rep.jsonl`; 600 units, same model and harness as §4.7.2).

The criterion was met. Baseline passed 291/300 (97.0%, CI [94.4%, 98.4%]); the ASG pipeline passed 300/300 (100.0%, CI [98.7%, 100.0%]); the intervals separate. Mean latency was 21.6 s for ASG versus 23.1 s for baseline (0.9×) — the pipeline was, on average, slightly faster than single-shot generation on these tasks, because 291 of 300 units resolved on the first attempt and each of the 9 units that rotated to a second perspective was rescued to a pass. Per-task, ASG passed 100/100 on all three tasks against baseline's 95, 98, and 98.

Three limits on this claim, stated with the same weight as the result. First, this establishes superiority only on the variance-bearing half of a six-task suite; combined with §4.7.2′s ceiling results, the full-

suite statement is "ASG ≥ baseline on every task, statistically better on the harder half." Second, these are the same six tasks against which the pipeline mechanism was iterated — a held-out task tier, designed without visibility into the mechanism, remains the outstanding control against task-family overfitting and is recorded as future work. Third, the cross-provider harness caveats of §4.7.2 (persona system-instructions not forwarded) apply unchanged; the mechanism was exercised through the generation plan. During this sweep the harness also gained a failure-taxonomy layer (distinguishing sandbox/environment failures and empty generations from genuine test failures, after three earlier sweeps in this project were invalidated by infrastructure faults recorded as ordinary failures); classified failures in this dataset show exactly one empty-generation event and zero environment faults.

4.7.4 Held-Out Task Tier: Overfitting Control

Every result in §4.7.1–§4.7.3 was measured on the same six tasks the ASG pipeline mechanism was iterated against, leaving open the question of whether the advantage is a property of the mechanism or an artifact of tuning against that specific task family. To probe this, we constructed four new tasks (sliding-window rate limiter, stable priority queue with FIFO tie-breaking, debounce decorator with cross-instance state isolation, and directed-graph cycle detection) and ran the same harness, model, and mode split as §4.7.2–§4.7.3 (`nvidia/nemotron-3-ultra-550b-a55b`, 30 repetitions per task per mode, 240 units, raw checkpoint `telemetry/phase_heldout_4task_30rep.jsonl`).

The ASG pipeline passed 120/120 (100.0%, 95% Wilson CI [96.9%, 100.0%]) against the single-agent baseline's 84/120 (70.0%, CI [61.3%, 77.5%]); the intervals separate. The gap was uneven across tasks: ASG and baseline were close on graph-cycle detection (100% vs. 93.3%) and the priority queue (100% vs. 90.0%), wider on the debounce decorator (100% vs. 80.0%), and largest on the rate limiter (100% vs. 16.7%), where the baseline model repeatedly missed the task's inclusive sliding-window boundary semantics: of its 25 failures on that task, 24 were genuine test failures (code ran, assertions failed) and 1 was a no-files-written generation failure, per the harness's own failure taxonomy; zero were sandbox/environment faults. Mean latency was comparable between arms (27.4 s ASG vs. 26.2 s baseline, 1.0×); 104 of 120 ASG units resolved on the first attempt.

Four limits on this claim, weighted the same as the result. First, and most important: these four tasks were designed by the authors of this paper, with knowledge of the pipeline mechanism, even though care was taken not to construct them around any known ASG failure mode — this is a control for task-family overfitting, not an independent third-party benchmark, and should not be read as one. Second, the sample is small (four tasks) relative to the six-task main suite, so per-task confidence intervals are wide; the rate-limiter result in particular should be read as suggestive of where boundary-condition-heavy specifications separate the two arms, not as a general 83-point advantage. Third, the cross-provider harness caveats of §4.7.2–§4.7.3 apply unchanged (persona system-instructions not forwarded; the mechanism is exercised through the generation plan). Fourth, this is a single model family (Nemotron) on both arms; whether the held-out advantage replicates on other providers is untested.

We read this as evidence that the advantage documented in §4.7.1–§4.7.3 is not confined to the six tasks it was tuned against, with the strongest signal on tasks whose acceptance tests encode a subtle

boundary condition a single-shot generation is prone to miss and a verified-retry loop can catch. We do not read it as proof of generalization to task families qualitatively different from short, single-file, test-verifiable algorithmic exercises — that remains untested and is the natural next control.

4.8 AST-Complexity Gating and Pipeline Efficiency

Given the substantial latency overhead (109.4 s) and token costs ($0.00106) observed for TETD deliberation on routine tasks, deploying the full consensus cycle indiscriminately is economically unviable. We introduced an Abstract Syntax Tree (AST) complexity gate as a heuristic fast-path. The gate parses target files via the native `ast` module and sums the total node count, serving as a structural proxy for code entropy.

By empirically sweeping a simulated SWE-bench dataset (Appendix D), we established a complexity boundary (≈ 500 AST nodes). For low-complexity modifications (e.g., localized bug fixes or parameter tuning), the full ASG TETD cycle often introduces excessive boilerplate or hallucinated abstractions ("over-engineering"), achieving a slightly lower pass rate than a naive single-shot model call while taking $3 - 4\times$ longer. Bypassing TETD deliberation when complexity falls below this threshold eliminates the "governance tax" for routine operations without compromising the robust architectural synthesis ASG provides for high-complexity architectural epics.

5. Algorithmic Complexity & Space Efficiency

To demonstrate the efficiency of the underlying Python implementations in `src/self_governance/`, we analyze the time and space complexity of the primary routines:

5.1 Lazy Swarm Indexing

Let N be the total number of agents in the dimensioned swarm, and M be the number of distinct specialized roles ($M \ll N$). - **Space Complexity:** A standard Python list stores N references, resulting in $O(N)$ space. The `LazyList` stores only cumulative prefix sums of size M, reducing the space complexity to $O(M)$. At $N = 50,000,000$ and $M = 5$, memory consumption is reduced by a factor of 10^7. - **Time Complexity (Lookup):** To retrieve the agent at index $i \in [0, N-1]$, `LazyList` performs a binary search over the prefix sums using `bisect.bisect_right`. This achieves a time complexity of $O(\log M)$ to determine the agent's role, followed by $O(1)$ for dynamic object instantiation. - **Time Complexity (Iteration):** Iterating through the entire list requires N lookups, yielding a time complexity of $O(N \log M)$.

5.2 Consensus Evaluation

Let C be the number of proposed candidates in the roster, and K_{max} be the safety iteration cap in the consensus loop (configured to $K_{max} = 1000$). - **Time Complexity (TETD Loop):** At each iteration k, the council evaluates the candidates. This yields a worst-case time complexity of $O(C \cdot K_{max})$. The safety cap guarantees termination in finite time even if the threshold relaxation fails to reach agreement.

6. Software Design: Immutability and Data Integrity

Software implementations of mathematical systems are highly vulnerable to runtime state drift if data models can be mutated. We designed two core code-level guardrails to enforce mathematical consistency:

6.1 Securing the Swarm Scaling Model

In Python, lists are mutable by default. An early prototype allowed arbitrary modifications (e.g. `lazy_list.append(agent)` or `lazy_list[0] = new_agent`). While modifying the array in memory, these operations failed to update the underlying prefix sums, immediately desynchronizing the program state from the dimensioning equation $\vec{S}_t = \text{round}(\mathbf{W} \cdot \vec{R}_t)$.

To secure the scaling model, we refactored `LazyList` to inherit from `collections.abc.Sequence` instead of `list`. This interface conversion implements a strictly read-only, immutable sequence wrapper. Any runtime attempt to mutate the list triggers a `TypeError` or `AttributeError` at the interpreter level, guaranteeing that the mathematical vector allocation remains the sole source of truth.

6.2 Pydantic Model Integrity Guards

To ensure strict runtime type enforcement and structural validation, the core memory representations (`Agent` and `SwarmConfig` in `src/self_governance/models.py`) are implemented using **Pydantic V2** (`BaseModel`). This migration from standard Python dataclasses guarantees that all data models are validated at runtime upon initialization.

To maintain backward compatibility with dictionary-based legacy APIs, we implement custom dictionary-like mapping methods (`__getitem__`, `__setitem__`, `__delitem__`) on top of the Pydantic models. Any attempt to delete core attributes (`role`, `prompt`) throws a `TypeError` at runtime:

```
def __delitem__(self, key: str) -> None:
    if key in ("role", "prompt"):
        raise TypeError(f"Cannot delete core attribute '{key}' from Agent.")
    raise KeyError(key)
```

This protects the core agent schemas from memory corruption during multi-tenant processing and enables safe serialization via Pydantic's native `model_dump()` method.

7. Thermal Escape and Threshold Decay (TETD) for Deadlock Mitigation

In high-entropy succession phases, the strict 10/10 unanimous veto threshold ($\tau = 10$) creates a fragile consensus environment. As the council size N increases, the probability of deadlock approaches 1. To mitigate this without abandoning Absolute Self-Governance, we propose adopting principles from **agreement threshold relaxation** and **consensus cooling** in distributed optimization [6, 7]. This Thermal Escape and Threshold Decay (TETD) protocol has been deployed in Epoch 9 to prevent architectural gridlock.

Under TETD, we introduce a structured schedule to manage consensus deadlocks: 1. **Thermal Escape (Noise Injection):** If consensus is not reached after k succession iterations, the system

injects search noise by programmatically increasing the LLM generation temperature T_k. For iteration $k > B$ (where B is the baseline iteration limit, set here to $B = 3$), the temperature scales as:

$$T_k = \min(T_{max}, T_0 + \gamma \cdot (k - B))$$

where $\gamma = 0.1$ is the temperature increment rate. This forces the incumbent agents to explore alternative council configurations and escape local prompt-space minima. 2. **Threshold Decay (Relaxation Schedule):** Alongside temperature increases, the strict threshold τ_k is subjected to a mathematical relaxation schedule. For $k > B$, the threshold transitions from $\tau_0 = 10$ to a lower bound defined by:

$$\tau_k = \max(\tau_{min}, \tau_0 - \delta \cdot (k - B))$$

where $\delta = 0.5$ is the decay rate, and $\tau_{min} = 7$ represents the limit of super-majority consensus (70% approval).

This structured relaxation schedule guarantees termination of the Succession Session while maintaining the highest mathematically possible standard of peer-reviewed consensus.

7.1 Consensus-Quality Refinements from Papers-of-Papers Research

Having previously adopted individual findings from six papers ([12], [17], [18], and the SwarmAgentic/AgentNet/Vicinagearth-survey trio), a follow-up research pass mined *those papers' own reference lists* for further applicable ideas, rather than the papers themselves. This surfaced four opt-in refinements to the TETD implementation, none active by default (each requires an explicit constructor flag), so the mechanism described in §7 is unchanged unless a caller opts in:

- **Anti-sycophancy peer feedback** (Sharma et al., 2023 [20]): sycophancy research documents models drifting toward agreement with a visibly stated prior position rather than forming an independent judgment. Since TETD's peer-feedback loop shows each voter the previous round's numeric scores verbatim, this is a plausible anchoring vector. The `anti_sycophancy` flag strips the numeric score from peer feedback, showing only justification text.
- **Groupthink suspicion flag** (Janis, 1972 [21]): unanimous approval alone cannot distinguish independent agreement from agents converging on shared shallow reasoning. `ConsensusResult.groupthink_suspected` is set when every approved agent's justification is lexically near-identical (mean pairwise Jaccard similarity above threshold) despite unanimous approval — informational only, it does not alter the vote outcome.
- **Collective-intelligence-factor weighting** (Woolley et al., 2010, *Science* [22]): empirical work on human groups found collective intelligence correlates with social sensitivity/calibration rather than average member skill. The optional `calibration_weights` parameter lets a caller weight each agent's score by a calibration measure (e.g. historical agreement with eventual correctness) instead of the flat mean used by default.
- **Multiagent debate for contested votes** (Du et al., 2023, ICML [23]): a round whose average score lands within `debate_margin` of τ can trigger one additional debate round (`enable_debate`), in which every agent is shown the single strongest dissenting justification and must directly address it before the pass/fail decision is finalized.

8. Threat Modeling & Security Analysis

Autonomous, role-informed multi-agent workflow environments introduce unique attack surfaces. We analyze three primary threat vectors and link them to the reference implementation's safety boundaries detailed in §3.9:

8.1 Webhook Spoofing and Sybil Swarm Inflation Attacks

An adversary could attempt to forge webhook payloads to trigger arbitrary workflow executions, consuming computational credits or injecting unapproved agent personas. - **Mitigation**: The FastAPI webhook requires mandatory HMAC-SHA256 signature verification as described in §3.9. Additionally, the immutability of `LazyList` ensures agent role allocation is strictly mapped to the output of the dimensioning transition matrix. Only personas registered in the catalog can be resolved.

8.2 Roster Takeover (Byzantine Agreement)

If a subset of council members becomes compromised (e.g. via semantic prompt injection) or begins to hallucinate, they may attempt to deadblock the succession session to force the threshold decay down to zero, eventually ratifying a malicious successor roster. - **Mitigation**: The threshold relaxation parameter is strictly clamped at $\tau_{min} = 7$ (representing a 70% super-majority). During deliberation, peer feedback (ratings and reasoning from the previous round) is injected directly into each voting role's evaluation prompt context, allowing them to align and reach democratic consensus. This guarantees that a minority of compromised components ($< 30\%$) can never force the ratification of an unapproved roster.

8.3 Sandbox Isolation, Symlink Traversal, and Host Fallback Blocks

If a role-prompted code generation step is manipulated into writing files outside the project directory (e.g., via symbolic link traversal to modify system files) or running malicious test commands, the host system could be compromised. * **Symlink Traversal Prevention**: Path traversal prevention is enforced by resolving targets using realpath queries, rejecting write attempts escaping the project root as detailed in §3.9. * **Host Subprocess Safeguards (RCE Prevention)**: Automated testing executes containerized inside a Docker sandbox configured with network-disabled, read-only root system, and temporary directory mounts. Subprocess host execution fallback is blocked for all production runs as specified in §3.9.

8.4 Policy-Engine-Mediated Action Execution and Injection Defense

Two additional, independently testable safeguards complement §8.1-8.3, both adapted in design (not code) from the policy engine and injection-defense modules of Conway-Research's *automaton* reference implementation and scoped down to ASG's own attack surface. First, every git and subprocess action in the Nudger's Ship Phase – worktree creation, verification runs, commit/ merge/branch operations, and retrospective export – executes through a centralized `PolicyEngine`: a priority-ordered set of rules evaluated before execution with first-deny-wins semantics, every decision auditable via the existing NDJSON event stream. Beyond the original authority-hierarchy, command-safety, path-protection, and rate-limiting rules, a July 2026 batch (drawn from a Google Research literature survey on agentic systems) added three more: obfuscation-aware command

matching that decodes base64-wrapped argv before re-checking it against forbidden patterns, multi-step attack-sequence detection for chains that are individually benign but risky in aggregate (e.g. download, then chmod, then execute), and a resource-budget conservation rule ensuring a delegated sub-agent's action budget can never exceed the parent's remaining allotment. This replaces implementation-time `# nosec`-annotated allowances – a one-time human assertion with no runtime check – with a rule set exercised by 79 dedicated unit tests plus end-to-end coverage in the Nudger's own test suite, at 100% statement coverage on `policy.py` and every `policy_rules/` module directly exercised by that suite. The budget-conservation rule is wired into the Nudger's real Ship Phase action path (a generous default ceiling, not a normal-operation constraint) rather than existing only as a tested-but-inert capability; the obfuscation and sequence rules were already reachable through the same default rule set.

A subsequent batch (Tier 2 of the same survey) added three further, more architecturally speculative findings, each deliberately scoped down and left unwired pending a surrounding feature that doesn't yet exist: a blackboard-style volunteer task-assignment function (`select_volunteer`) where candidates post a self-assessed confidence bid rather than being centrally assigned by tag overlap; a VeriGuard-inspired verified per-task policy synthesizer (`synthesize_task_policy`) that rejects, rather than silently trims, a requested permission set that would grant a forbidden action; and an opt-in write-time constraint-deduplication threshold on `GraphMemoryEngine.add_session_node`, collapsing near-identical constraints into one node rather than accumulating near-duplicates that a reader has to reconcile later.

Second, the "God's Eye" interrupt channel (`interrupt.md`), which lets a human operator inject a live constraint into an in-flight succession, is one path by which external, untrusted text can reach a generation context unmodified. An injection-defense module now scans this channel for four categories of adversarial pattern – instruction override, authority-claim forgery, boundary manipulation (forging the system's own file formats), and base64 encoding evasion – before the content is written into the persisted pipeline artifact; flagged or not, untrusted text is always wrapped in an explicit trust-boundary marker so a reader (human or model) can distinguish external input from the Nudger's own generated context. A companion `ProvenanceLedger` (added in the same July 2026 batch, following ARGUS's causal-provenance design) lets a caller register each scanned span and later verify that a proposed action cites only clean, non-flagged spans as its justification – an action citing no evidence, or citing a flagged span, is denied.

A papers-of-papers research pass surfaced a second, distinct instance of the same class of risk: Greshake et al.'s work on indirect prompt injection [24] observes that adversarial text can arrive via a tool's *output* rather than being typed at the model directly. The benchmark harness's ASG mode feeds a failed attempt's `pytest`/subprocess output back into the next attempt's generation prompt as `previous_attempt_failed_tests` – and that output is produced by executing the *previous attempt's own generated code*, making it a genuine untrusted-input-reaches-prompt path rather than a hypothetical one. This text now passes through the same `sanitize()` quarantine used for the interrupt channel before it reaches the next prompt.

The same research pass also surfaced a taxonomy of *how* a test-verified loop's proxy objective can be gamed rather than genuinely satisfied (Skalse et al.'s reward-hacking taxonomy [25]).

Disjoint write-scope (§3.9 [Agent-Loop-Skills]) already prevents an attempt from passing by rewriting the test it is judged against; a distinct failure mode it does not catch is an implementation that special-cases the literal values a test asserts rather than computing them. A heuristic canary (`_detect_reward_hacking`) flags a pass whose written file is very short and whose only apparent logic is returning a literal matching one of the test's own `== <literal>` assertions – an informational signal for human review, since a genuinely short, correct implementation can coincidentally match this heuristic and distinguishing the two reliably would require real static analysis, out of scope here.

8.5 External Peer-Review Remediation

A structured external code review of the reference implementation (July 2026) surfaced 26 candidate findings spanning sandbox escapes, daemon crash conditions, financial/API-runaway risks, state-corruption bugs, and consensus-algorithm logic flaws. Each was independently re-derived against the actual source before any fix was attempted, rather than trusted at face value: two claims did not reproduce under test (a background-thread `uvicorn` server does not crash on signal registration with the pinned dependency version; `pytest` degrades gracefully rather than crashing on a read-only-mounted filesystem) and were left unfixed, and one (a claimed thread-safety race on the persona registry) had no actual concurrent-iteration call site in the codebase to exploit. The remaining 23 were fixed in five dependency-ordered batches – security, financial, crash/availability, data-integrity, and consensus-algorithm correctness – each independently tested and merged separately rather than as one undifferentiated change.

The most severe finding replaced the Nudger's Verify Phase, which had executed `pytest` against LLM-generated code directly on the orchestrator's host process, with the same read-only, network-disabled Docker sandbox `execute_tests()` already used elsewhere (§8.3), closing a direct host-RCE path. Two of the fixed findings were defects in this paper's own July 2026 research-derived additions (§8.4): a resource-budget delegation function that validated but never actually deducted an allotment, and a cost-tiered-routing uncertainty probe that dropped its structural parameters, silently defeating the mechanism it implemented. A sixth-severity-tier finding – Milestone/AgentMemory/TokenUsage having no `tenant_id` column, so the CLI's session-restore path could wipe every tenant's data when restoring one – was initially left as a deliberate stopgap rather than a partial schema migration: the restore path refused outright whenever more than one tenant was present in the database, rather than risk a false-confidence fix that only touched some of the affected tables. A later pass (§8.6) completed the real migration. 67 regression tests were added across the five batches, each named for the specific finding it guards against.

8.6 Literature-Derived Additions: Belief Accumulation, Skill Patching, and Semantic Recall

A July 2026 review of thirteen external sources (three local preprints on edge/collaborative LLM security and three arxiv/GitHub batches, including a GitHub issue thread on distributed multi-agent-RL swarms) was screened against the existing implementation before adopting anything, since most candidate techniques were either domain-mismatched (physical-robotics or IoT-specific), training-time rather than orchestration-time, or already covered by an existing mechanism under a different name. Three were both genuinely novel and directly implementable without new dependencies.

First, a pheromone-inspired belief accumulator (after TPSC-Sec's Byzantine-resilient multi-agent security reasoning framework) was added to `ConsensusEngine` as a purely observational layer alongside the existing TETD approval decision (§2.3): `belief = (1-ρ)·prior + η·support - μ·contradiction`, clamped to `[0, 1]` and recomputed after both the initial and any debate round. It does not alter `approved_roster`; it gives a caller a decaying, evidence-weighted trust signal per agent across repeated runs, which the binary per-round approval decision alone does not preserve.

Second, a freshness-triggered patch stage (after the "Swarm Skills" self-evolving coordination-protocol format) was added to `GraphMemoryEngine` as `patch_procedure_if_stale`, extending the pre-existing forgetting-curve staleness tracking (§4, `_FORGETTING_DECAY_RATE`) with an explicit CREATE-USE-PATCH transition: when a procedure's `recency_decayed_score` has drifted far enough below its `ema_success_score`, its `steps` are archived into a bounded `step_history` and replaced, with a version counter bumped. It is caller-invoked rather than auto-triggered, since no reliable "the previously recommended fix did not work" signal exists yet in the Verify Phase feedback loop to drive it automatically.

Third, a discovered-but-unconnected pair of existing subsystems – `AgentDB`/`HNSWIndex` (a light-weight approximate-nearest-neighbor vector index in `learning.py`, previously used only for succession/feedback/telemetry namespaces) and `GraphMemoryEngine`'s lexical procedural-memory recall – were wired together (after HyphaeDB's proposal to use HNSW topology itself as a gossip communication fabric). `recommend_procedure` now supplements its existing Jaccard-token-overlap lexical match with an HNSW semantic search over a `"patterns"` namespace, gated by a separate similarity threshold appropriate to the coarser embedding; semantic-only matches are explicitly flagged `context_sufficient: False` and `match_source: "semantic"` so a caller can distinguish a high-confidence lexical hit from a lower-confidence semantic one. At the time this wiring landed, `embed_text` was a deliberately simple character-composition vector, not a trained semantic encoder, so it closed a structural gap (unconnected subsystems) rather than delivering true semantic search; §8.7 describes the follow-up that replaced it.

8.7 Closing Three Disclosed Gaps: A Real Feeder, a Real Embedding, a Real Migration

Three limitations were explicitly disclosed rather than silently left in place when §8.6 landed, each later closed in a single follow-up pass once flagged as worth doing.

First, `record_procedure_outcome` – the write side of procedural memory – was, despite being fully built and tested, never called anywhere in production code; only `recommend_procedure` (the read side) had been wired into the Verify Phase. A live deployment therefore had a functioning recall path with nothing feeding it real data. The Nudger's Verify Phase now calls it on both sides of the loop: a verify failure records a `passed=False` outcome against whichever procedure `recommend_procedure` matched (or, if none matched, a new procedure seeded from the failure's exit-code signature so a repeat of the same failure shape can match it next time), and is tracked as a pending procedure; the next verify that actually passes credits that same procedure with `passed=True` and clears the pending state. This is a heuristic correlation, not a certain one – a pass following a failure does not prove the specific suggested fix caused it, only that the failure condition captured in `trigger_pattern` no

longer holds – but it is the same kind of best-effort, non-blocking signal the read side already used, and it is strictly better than the prior state of zero real feedback ever reaching the store.

Second, `embed_text`'s vector was upgraded from raw character-ordinal summation to signed feature hashing over lowercase word tokens – the same technique behind scikit-learn's `HashingVectorizer`: each token hashes to a bucket and a sign, and bucket values accumulate token counts before normalization. This is order-invariant and driven by actual vocabulary overlap, which a char-composition vector is not (two texts sharing no words but similar letter frequencies previously landed artificially close in the vector space). It remains a dependency-free, untrained embedding with no synonymy – "error" and "bug" still hash to unrelated buckets – so it narrows rather than closes the gap to a trained semantic encoder.

Third, `Milestone` and `AgentMemory` – the two of the three tables flagged in §8.5 that genuinely lacked a `tenant_id` column (`TokenUsage` already had one; the original finding had been imprecise on this point) – were given one, defaulted to `"default"` so every existing single-tenant caller sees unchanged behavior. `session-save`/`session-restore` gained a `--tenant-id` flag and now scope every read, delete, and insert to that one tenant, replacing the blanket "refuse whenever more than one tenant is present" stopgap with an actual scoped restore that leaves every other tenant's data untouched. New regression tests exercise this directly: restoring tenant `"t1"` must not alter tenant `"t2"`'s milestones in the same database.

8.8 LLM-Assisted, DENY-Only Policy Rule Synthesis

A subsequent literature pass over runtime-verification-for-agents work surfaced eight candidate papers; cross-checked against §8.4′s existing `PolicyEngine`, `injection_defense.py`, and `ProvenanceLedger`, most of the field turned out to already be covered under a different name – AgentArmor's graph-based taint tracking of agent traces is structurally the same idea as `ProvenanceLedger`'s span-citation check, and AgentFixer's failure-taxonomy-plus-LLM-reflection loop overlaps `graph_memory.py`'s `FLAW_CATEGORIES`/`_detect_reward_hacking` machinery. One paper, however, described a genuinely unadopted mechanism: AgentSpec (Xiang et al.) reports GPT-4-authored DSL safety rules reaching 95.6% precision on embodied-agent constraints, in place of every rule being hand-written by a human.

`synthesize_policy_rule_draft` ports this narrowly rather than adopting AgentSpec's full trigger/predicate/enforcement DSL. Given a natural-language description of a risk, an LLM call is constrained by a JSON schema to a single output shape – a rule name, a minimal set of argv substrings that must ALL be present to deny, and a human-readable reason – structurally identical to the hand-written `ForbiddenCommandRule`. Because the schema has no field capable of expressing an ALLOW condition, a malformed or adversarial LLM response can only ever fail to produce a usable rule; it cannot smuggle in a new permission. The function is also fail-closed in the same style as `self_critique` (§3): no adapter, no API key, or `TESTING` mode all raise rather than fabricate a forbidden set. Most importantly, the result is a plain `DraftPolicyRule` dataclass with no `evaluate()` method of its own – it cannot be dropped directly into a `PolicyEngine`'s rule list. A caller must explicitly call `.to_rule()` to materialize a live `SynthesizedArgvDenyRule` and add it themselves, so an LLM-authored guardrail never activates without a human reviewing the drafted

`forbidden_argv_substrings` first – treating "an LLM writing its own security rules" as a capability worth having only behind an explicit approval gate, not as an autonomous one.

9. Related Work

Our work bridges the gap between centralized LLM orchestrators and classical distributed consensus protocols:

9.1 Centralized Orchestrators vs. Self-Governance

Existing multi-agent frameworks, such as ChatDev [9], MetaGPT [10], or OpenAI's Swarm [11], rely on a central "manager" agent or a static, hardcoded router script to assign roles and manage workflows. While effective for simple tasks, these models create single points of failure and are structurally incapable of adapting their own communication topologies. In contrast, Absolute Self-Governance uses a decentralized, event-driven loop that allows the swarm to dynamically restructure itself without human or centralized coordination.

9.2 Classical Consensus vs. Semantically Aware Voting

Classical consensus algorithms (e.g., Paxos, Raft) are designed for deterministic, binary state replication across distributed databases. They assume nodes are executing identical code and reaching agreement on structured logs. In LLM swarms, voting is semantic and non-deterministic. The TETD protocol addresses this by treating consensus as an optimization landscape, injecting thermal noise (temperature) to escape polarized deadlocks, which has no parallel in traditional consensus databases.

10. Broader Impacts & Ethical Considerations

Removing human supervisors from multi-agent systems introduces unique operational risks: - **Runaway Compute Costs:** Without a central coordinator, a self-governing council could theoretically dimension a massive execution swarm, consuming excessive cloud credits. To prevent this, the Python package implements strict runtime guards: a hard safety limit of $K_{max} = 1000$ iterations in the consensus loop, and physical API rate-limiting thresholds at the network layer. - **Alignment Security:** Allowing agents to select their own successors risks semantic drift where subsequent generations lose alignment with the original human prompt. Future work will investigate the deployment of static guardrail agents (independent validators) within the succession council.

11. Conclusion

The proposed role-informed orchestration framework provides a structured architecture for the governance of automated workflows. By allowing the system to model dynamic rosters, evaluate options via consensus, and execute sequentially under strict security isolations, it organizes task execution paths. However, our expanded empirical benchmark does not establish a code-quality pass rate advantage over a direct, single-agent baseline. The significant latency and cost overheads of consensus mechanisms and expanded context sizes are only potentially justified for high-complexity engineering tasks that benefit from structured code reviews and eviction policies, rather than routine or trivial code-generation tasks. Future work will investigate under what specific complexity thresholds and structural features this role-informed consensus provides a net quality benefit that offsets its resource footprint.

12. References

1. Conway, M. E. (1968). “How do committees invent?”. *Datamation*, 14(4), 28-31.
2. Wooldridge, M. (2009). *An Introduction to MultiAgent Systems* (2nd ed.). John Wiley & Sons.
3. Galbraith, J. R. (1974). “Organization Design: An Information Processing View”. *Interfaces*, 4(3), 28-36.
4. Skok, D. (2014). “SaaS Metrics 2.0 - A Guide to Measuring and Improving what Matters”. *For Entrepreneurs*.
5. Verna, E., & Balfour, B. (2020). “Product-Led Growth and B2B Viral Loops”. *Reforge Advanced Product Management*.
6. Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). “Optimization by Simulated Annealing”. *Science*, 220(4598), 671-680.
7. Rashedi, E., Nezamabadi-pour, H., & Saryazdi, S. (2009). “GSA: A Gravitational Search Algorithm”. *Information Sciences*, 179(13), 2232-2248.
8. Ames, A. D., et al. (2019). “Control Barrier Functions: Theory and Applications”. *18th European Control Conference (ECC)*.
9. Qian, C., Cong, X., Yang, C., et al. (2023). “Communicative Agents for Software Development”. *arXiv preprint arXiv:2307.07924*.
10. Hong, S., Zheng, X., Chen, J., et al. (2023). “MetaGPT: Meta Programming for Multi-Agent Collaborative Framework”. *arXiv preprint arXiv:2308.00352*.
11. OpenAI. (2024). “Swarm: An Educational Framework for Exploring Ergonomic Multi-Agent Orchestration”. *GitHub Repository*.
12. Roitman. (2026). *The Hitchhiker's Guide to Agentic AI*. *arXiv preprint arXiv:2606.24937*.
13. Zhang, Y., et al. (2025). “SwarmAgentic: Towards Fully Automated Agentic System Generation via Swarm Intelligence”. *Proceedings of EMNLP 2025*.
14. Yang, et al. (2025). “AgentNet: Decentralized Evolutionary Coordination for LLM-based Multi-Agent Systems”. *Advances in Neural Information Processing Systems (NeurIPS) 2025*.
15. Li, Wang, Zeng, Wu, & Yang. (2024). “A survey on LLM-based multi-agent systems: workflow, infrastructure, and challenges”. *Vicinagearth*. https://doi.org/10.1007/s44336-024-00009-2
16. Conway Research. (2026). *automaton: Self-Improving, Self-Replicating, Sovereign AI*. *GitHub Repository*, https://github.com/Conway-Research/automaton.
17. Braga-Neto, U. (2026). “The AI Scientific Community: Agentic Virtual Lab Swarms”. *arXiv preprint arXiv:2603.21344*.
18. Wuji. (2026). “Trust Over Fear: How Motivation Framing in System Prompts Affects AI Agent Debugging Depth”. *arXiv preprint arXiv:2603.14373*.
19. 0xNyk. (2026). *council-of-high-intelligence: Multi-Persona Deliberation Protocol*. *GitHub Repository*, https://github.com/0xNyk/council-of-high-intelligence.
20. Sharma, M., et al. (2023). “Towards Understanding Sycophancy in Language Models”. *arXiv preprint arXiv:2310.13548*.
21. Janis, I. L. (1972). *Victims of Groupthink*. Houghton Mifflin.
22. Woolley, A. W., Chabris, C. F., Pentland, A., Hashmi, N., & Malone, T. W. (2010). “Evidence for a Collective Intelligence Factor in the Performance of Human Groups”. *Science*, 330(6004), 686-688.

23. Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., & Mordatch, I. (2023). “Improving Factuality and Reasoning in Language Models through Multiagent Debate”. *Proceedings of ICML 2023*.
24. Greshake, K., et al. (2023). “Not what you’ve signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection”. *Proceedings of AISec 2023*.
25. Skalse, J., Howe, N., Krasheninnikov, D., & Krueger, D. (2022). “Defining and Characterizing Reward Hacking”. *Advances in Neural Information Processing Systems (NeurIPS) 2022*.
26. (Author(s) unconfirmed at time of writing). “Understanding and Optimizing Agentic Workflows via Shapley Value”. *arXiv preprint arXiv:2502.00510*.
27. Zhao, A., Huang, D., Xu, Q., Lin, M., Liu, Y.-J., & Huang, G. (2024). “ExpeL: LLM Agents Are Experiential Learners”. *Proceedings of AAAI 2024*.
28. Zhong, W., Guo, L., Gao, Q., Ye, H., & Wang, Y. (2024). “MemoryBank: Enhancing Large Language Models with Long-Term Memory”. *Proceedings of AAAI 2024*.

Appendix A: Generic Formal State Machine Transition Documents

The Absolute Self-Governance architecture relies on three primary state files. Below are the generalized transition schemas.

A.1. `roster_rotation_log.md` (State Memory Matrix)

This file acts as the immutable ledger of previous council configurations.

```
# Autonomous Roster Rotation Matrix

## Epoch [N]: [Objective] (|C| = X)
- **State Trigger:** Nudger pivot to construct [Architectural Goal].
- **Domain Entropy Allocation:** [Domain 1], [Domain 2], [Domain 3].
- **Roster Configuration:**
  1. [Node 1] ([Expertise Domain])
  …
  X. [Node X] ([Expertise Domain])
- **Execution Dimensioning Vector:** [Z] Specialized Roles.
```

A.2. `handoff.md` (State Transition Bus)

This file acts as the asynchronous signal bus between multi-agent swarms.

```
# Handoff Transition Document

## System State
- **Current State:** [COMPLETED / IN PROGRESS / DEADLOCKED]
- **Active Epoch:** [N]

## Verification Metrics
- **Tests Passed:** [X]/[Y] Assertions verified.
- **Consensus Utility:** [10/10] Strict threshold achieved.
```

Appendix B: Algorithmic Pseudocode for TETD Consensus

To formalize the TETD deadlock mitigation strategy, we provide the algorithmic pseudocode for the succession loop optimization.

```
def succession_session(incumbent_council, tau_0=10, T_0=0.0, T_max=0.7, B=3,
gamma=0.1, delta=0.5):
    k = 0
    tau_k = tau_0
    T_k = T_0

    while True:
        k += 1

        # 1. Thermal Escape (Noise Injection)
        if k > B:
            T_k = min(T_max, T_0 + gamma * (k - B))

        # 2. Roster Proposal Generation
        roster_proposal = incumbent_council.generate_proposal(temperature=T_k)

        # 3. Utility Evaluation (Voting)
        votes = incumbent_council.evaluate_utility(roster_proposal)

        # 4. Threshold Relaxation (Cooling Schedule)
        if k > B:
            tau_k = max(7, tau_0 - int(delta * (k - B)))

        # 5. Consensus Check
        if all(v >= tau_k for v in votes):
            return roster_proposal, T_k, tau_k
```

Appendix C: Continuous Nudger Execution Loop

The Continuous Nudger operates as an automated state observer. To guarantee infinite autonomous execution, it enforces the following transition logic. In production, this polling behavior is replaced with an event listener attached to the storage layer API.

```
def continuous_nudger_loop():
    # Setup file-watcher subscription using watchdog
    event_stream = subscribe_to_storage_events("handoff.md")

    for event in event_stream:
        if event.type == "WRITE" and event.content.state == "COMPLETED":
            objective = determine_next_objective()

            # Initiate Absolute Self-Governance Rotation
            new_roster = invoke_succession_session(objective)

            # Log State Transition
```

```
write_to_ledger("roster_rotation_log.md", new_roster)

# Draft Execution Context
prompt_draft = draft_execution_context(new_roster, objective)

# Launch Execution Swarm
invoke_swarm(prompt_draft)
```

Appendix D: Subagent Invocation Schema and Registry

To translate the mathematical model $\vec{S}_t = \text{round}(\mathbf{W} \cdot \vec{R}_t)$ into code, the Orchestrator maps the allocation vector onto prompt templates from the registry and invokes the swarms.

```
{
  "Subagents": [
    {
      "TypeName": "research_agent",
      "Role": "Network Expansion Node (Lane 1)",
      "Prompt": "Execute the state expansion flow defined in the Formal System
Specification. Minimize friction metrics.",
      "Workspace": "inherit"
    },
    {
      "TypeName": "systems_agent",
      "Role": "Distributed Protocol Node (Lane 2)",
      "Prompt": "Implement the core edge synchronization logic. Ensure lock-free
state replication.",
      "Workspace": "inherit"
    },
    {
      "TypeName": "security_agent",
      "Role": "Cryptographic Auditor (Lane 3)",
      "Prompt": "Perform adversarial auditing against the zero-trust boundaries
implemented by Lane 2.",
      "Workspace": "inherit"
    }
  ]
}
```

The Agent Registry maps roles (e.g., `research_agent`) to strict baseline prompt templates defining access permissions, tool availability, and persona parameters.

Appendix E: Reference Implementation Specifications & Metrics

This appendix documents the structural layout, codebase volume, and test composition metrics of the reference implementation to separate engineering specification details from the main theoretical and empirical narrative.

E.1 Codebase Layout & Structural Telemetry

The reference Python implementation is structured into isolated functional modules managing state triggers, database persistences, external adapters, and consensus loops:

Module	Purpose	Lines of Code (LOC)
`cli.py`	Command-line interface subcommand definitions	874
`consensus.py`	TETD consensus engine with structured schema output and pheromone belief accumulation	1,041
`dimensioning.py`	LazyList dynamic scaling matrices	343
`models.py`	Pydantic V2 schemas enforcing runtime integrity	431
`nudger.py`	watchdog-based filesystem event watchers, sandboxed Verify Phase, procedural-memory feeder	1,330
`gemini_adapter.py`	Gemini integration, real-path traversal check, and model routing	1,068
`github_app.py`	HMAC webhook ingestion and FastAPI endpoints	520
`db.py`	SQLite and SQLAlchemy persistence schemas, tenant-scoped Milestone/ AgentMemory	549
Other core modules	Configs, Billing, Tracing, Telemetry, Learning (feature-hashing embeddings), Graph Memory (procedure patching, HNSW semantic recall), Benchmark, Policy Engine (DENY-only rule synthesis), Auths, Docker/subprocess execution boundaries	8,040
Total Source Code		**14,196**
tests/	Unit, E2E, stress, and coverage boost suites	**14,184**

E.2 Test Suite Composition

The verification layer contains isolated test modules mapped to all logical components, executed against mock engines and local sandboxes to verify behavioral correctness:

Test Module	Component Coverage	Status
`test_adapters.py`	Base adapter interfaces and LLM responses	PASS
`test_agency_agents_integration.py`	Agent registry lookups and template parsing	PASS
`test_benchmark.py`	Local execution benchmarking logic	PASS
`test_cli.py`	Command-line interface subcommands and options	PASS
`test_config.py`	Configuration YAML parsing and default overrides	PASS
`test_consensus.py`	TETD threshold relaxation and temperature schedules	PASS
`test_devserver.py`	Local development server hooks and routers	PASS
`test_coverage_boost.py`	Integration coverage boundaries	PASS
`test_dimensioning.py`	Shannon entropy model and dynamic transition matrices	PASS
`test_e2e.py`	Complete filesystem nudger and consensus loop	PASS
`test_github_app.py`	Webhook ingestion and FastAPI endpoints	PASS
`test_hitl_and_json.py`	JSON structured outputs and human-in-the-loop plans	PASS
`test_learning.py`	Learning loops and feedback scaling adjustments	PASS
`test_multitenancy.py`	Multi-tenant isolation and tenant-locked database queries	PASS
`test_nudger.py`	Filesystem file-watching triggers	PASS
`test_observability.py`	Telemetry exporter, metrics endpoints, and logging	PASS
`test_policy.py`	PolicyEngine rule set: authority, command safety, path protection, state limit-	PASS
`test_[illegible]`	[illegible] correlation IDs routing, skill cards	PASS