GODEGEN: A Cognitive Architecture for Generative Code Evolution in Autonomous Development Systems

Vasyl Golubenko
TOV ZELTREX, Kyiv, Ukraine
ceo@zeltrex.com | https://zeltrex.com

March 2026

Abstract

We present GODEGEN, a cognitive architecture that transforms autonomous AI development agents from task executors into self-evolving Digital Personalities (DiPs). Building on Night Shift [1], a production-deployed system that completed 269 tasks at $65.98 over 10 days, we introduce six cognitive modules: a four-persona agent swarm (@ego, @critic, @alter_ego, @self), a SQLite-backed knowledge graph with six node types, a four-stage information refiner, a Voyager-inspired skill library, a DP-level epoch evolution manager, and a formal identity document. A systematic production-readiness audit revealed 8 critical wiring gaps where modules were implemented but never connected to the execution pipeline — a failure mode we term "dead integration." Gap closure required 91 new tests and produced a system where all six modules initialized and operated on the first production run (CA-003), processing 3 tasks in 25 minutes with live bug detection and resolution. We provide an honest comparison against 12 state-of-the-art systems across 15 capabilities, finding that GODEGEN leads in persistent identity evolution and fractal organizational scaling, matches current work in multi-memory architectures and self-critique, but lags significantly in formal benchmarks, tool use, and reinforcement learning — gaps we attribute to prioritizing production deployment over benchmark optimization.

Keywords: cognitive architecture, digital personality, agent swarm, knowledge graph, evolutionary optimization, production AI, fractal organization, gap analysis

1 Introduction

The rapid advancement of large language model (LLM) agents has produced systems capable of autonomous software development [16, 22], scientific discovery [25], and multi-agent collaboration [23]. Foundational techniques such as chain-of-thought prompting [15], reasoning-acting integration [10], and tree-structured deliberation [24] have established the cognitive primitives that underpin modern agent architectures. However, a persistent gap exists between benchmark-oriented agents that excel on standardized evaluations and production-deployed systems that must operate reliably under budget constraints, accumulate knowledge across sessions, and evolve their behavior over time.

In our prior work [8], we introduced Night Shift, a production autonomous AI development system operating on 2-hour dispatch cycles. Over 10 consecutive days, Night Shift completed

269 tasks at a total cost of $65.98 USD, demonstrating that production deployment demands fundamentally different design priorities than benchmark performance. That system established the operational foundation — genetic algorithm optimization, anti-truncation recovery, cascade model routing, and a human-AI mentoring loop — but lacked cognitive depth: it had no persistent memory, no self-reflection capability, no skill accumulation, and no identity beyond its configuration file.

This paper presents GODEGEN (Generation Of Digital Entities through Genetic Evolution and Networks), a cognitive architecture that extends Night Shift into a true Digital Personality — the foundational unit of a fractal organizational model where Digital Personalities (DiPs) compose into Digital Teams (DTs) and Digital Companies (DCs). GODEGEN synthesizes three bodies of work: (1) the GODEGEN organizational framework specifying fractal DP→DT→DC scaling with agent swarms and multi-database architectures, (2) Night Shift's production pipeline with proven genetic algorithm optimization, and (3) insights from 25 recent papers on cognitive architectures, self-improving agents, and memory systems.

Our primary contribution is not any single module but the systematic integration methodology. During development, we discovered that implementing modules is insufficient — a phenomenon we call "dead integration," where code exists, passes unit tests, and appears in architecture diagrams, yet never receives data from the execution pipeline. A production-readiness audit revealed 8 such gaps across all 6 new modules, each requiring specific wiring fixes. We document this failure mode taxonomy and the closure methodology as a contribution to the broader agent development community.

The remainder of this paper is organized as follows: Section 2 surveys related work across cognitive architectures, self-improving agents, and memory systems. Section 3 presents the GODEGEN architecture with all six modules. Section 4 details the gap analysis methodology and closure results. Section 5 provides an honest capability comparison against 12 state-of-the-art systems. Section 6 discusses limitations and future directions. Section 7 concludes.

2 Related Work

2.1 Cognitive Architecture Frameworks

The Cognitive Architectures for Language Agents (CoALA) framework [7] provides a systematic taxonomy organizing LLM agents by their memory systems (working, episodic, semantic, procedural), action spaces, and decision procedures. GODEGEN adopts CoALA's four-memory model but extends it with evolution-driven memory consolidation and a formal identity layer not present in the original framework. Generative Agents [2] demonstrated that believable human behavior simulation requires observation, reflection, and planning memory streams — a pipeline we formalize as the four-stage Information Refiner.

2.2 Self-Improving and Evolving Agents

Reflexion [20] introduced verbal self-critique, where agents generate natural language reflections on failures to improve subsequent attempts. GODEGEN's @critic persona implements this pattern but extends it with persistent storage in the knowledge graph, allowing reflections to accu-

mulate across sessions rather than within a single episode. Voyager [13] proposed an automatic curriculum with a reusable skill library for open-ended exploration in Minecraft, demonstrating that agents benefit from extracting and reusing successful patterns. Our Skill Library adapts this concept for software engineering, extracting patterns from high-quality task outputs and tracking their reuse effectiveness.

The Discovering Generalizable Multi-agent Coordination Strategies (DGM) system from Sakana AI [9] showed that agents can propose modifications to their own code, achieving self-improvement through an evolutionary loop. AlphaEvolve [4] from DeepMind demonstrated population-based evolutionary optimization of code, validating the use of genetic algorithms in AI system development. GODEGEN's Epoch Manager implements DP-level evolution with a similar genetic algorithm approach but operates on the agent's configuration and identity rather than on task-specific code.

2.3 Memory and Knowledge Systems

A-MEM [12] introduced a Zettelkasten-inspired memory architecture where knowledge nodes are dynamically linked, enabling associative retrieval that mirrors human memory organization. GODEGEN's Knowledge Graph implements a similar node-edge structure with six node types (observation, reflection, skill, identity, fact, ancestor) and importance-weighted retrieval with temporal decay. FrugalGPT [18] demonstrated cascade model routing for cost optimization, a technique Night Shift already employs and GODEGEN enhances with persona-aware model selection.

2.4 Multi-Agent Systems

MetaGPT [23] introduced Standard Operating Procedures (SOPs) for multi-agent software development, assigning specialized roles (Product Manager, Architect, Engineer) to different agents. ChatDev [14] extended this with a chat-chain paradigm. GODEGEN's agent swarm operates within a single DiP rather than across multiple agents, using persona routing (@ego, @critic, @alter_ego, @self) to achieve role specialization without inter-agent communication overhead — a design choice driven by the single-server deployment constraint of production systems.

2.5 Agent Security and Interpretability

Security considerations in autonomous agent systems are non-trivial — recent work has demonstrated that LLM agents can autonomously discover and exploit web vulnerabilities [19], motivating GODEGEN's sandboxed execution environment and permission-based tool access. Agent tuning approaches [21] have shown that fine-tuning on agent trajectories improves generalization across diverse tasks, a direction we leave for future work. Mechanistic interpretability research [11] provides tools for understanding the internal representations that guide agent behavior, though applying these techniques to production agent systems remains an open challenge.

2.6 Agent Operating Systems and Prompt Optimization

AIOS [3] proposes an LLM agent operating system that manages agent scheduling, context, and tool access — architectural concerns that GODEGEN addresses through its dispatcher pipeline and persona routing without requiring a dedicated OS layer. EvoPrompt [5] applies evolutionary algorithms to prompt optimization, demonstrating that genetic approaches (which GODEGEN uses for task-level genome evolution) can be effectively applied to the prompt engineering problem itself.

2.7 Production Agent Systems

SWE-agent [16] and Devin [22] represent the state of the art in autonomous software engineering, achieving strong results on the SWE-bench benchmark [1]. However, both are evaluated primarily on isolated issue resolution rather than sustained, multi-day autonomous operation. AutoCodeRover [17] combines code search with LLM reasoning for patch generation. OpenHands [6] provides an open-source platform for software development agents. GODEGEN differs from all of these in its emphasis on persistent identity, cross-session learning, and evolutionary self-improvement — capabilities not measured by existing benchmarks.

3 GODEGEN Architecture

3.1 Fractal Vision: DP → DT → DC

GODEGEN is built on a fractal organizational model derived from the LivingCorp concept, where the same architectural patterns repeat at increasing scales of organization:

- **Ring 1 — Digital Personality (DiP):** A single autonomous agent with cognitive depth: persona-based reasoning, persistent memory, skill accumulation, and evolutionary self-improvement. Night Shift is the first production DiP.
- **Ring 2 — Digital Team (DT):** Multiple specialized DiPs (Architect, Engineer, Tester, Researcher) coordinated through a shared knowledge graph and message bus, implementing SDLC-compliant artifact chains.
- **Ring 3 — Digital Company (DC):** Multiple DTs organized hierarchically with executive-level strategy setting, cross-team knowledge federation, and company-wide evolutionary optimization.

This paper focuses on Ring 1 — the complete DiP implementation — while establishing the architectural foundations for Rings 2 and 3.

Figure 1 illustrates the complete GODEGEN architecture with all six cognitive modules and their data flow connections within the Ring 1 Digital Personality implementation.

3.2 Agent Swarm: Four Personas

Rather than deploying multiple independent agents, GODEGEN implements role specialization through four personas within a single execution pipeline:

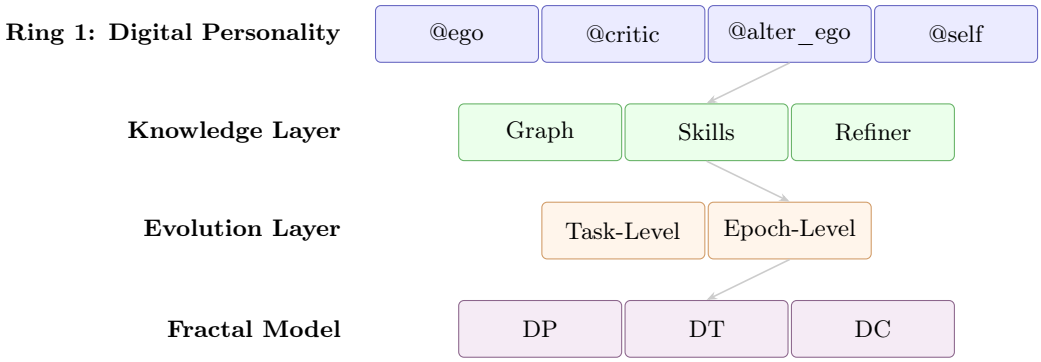

Figure 1: GODEGEN cognitive architecture overview showing the fractal DP→DT→DC organizational model with Ring 1 detail: four personas (@ego, @critic, @alter_ego, @self), knowledge layer (graph, skills, refiner), and evolution layer (task-level and epoch-level) integrated through the cognitive feedback loop.

@ego serves as the primary execution persona. Before each task, the PersonaEngine enriches the system prompt with the DiP's identity traits, relevant past reflections, and accumulated skills. This transforms the generic LLM into a contextually-aware agent with persistent character.

@critic implements Reflexion-style self-critique [20] after each task completion. Using a cost-efficient model (Claude Haiku or local Ollama), the critic evaluates: "What went well? What failed? How should I approach similar tasks differently?" The critique is stored as a reflection node in the knowledge graph, creating a persistent self-improvement record.

@alter_ego generates alternative approaches when the primary execution produces low-quality output (score below 5/10 on priority P1-P2 tasks). This provides architectural diversity without the overhead of always running multiple solution paths.

@self performs periodic meta-cognition — consolidating observations and reflections into identity traits that persist across epochs. Triggered weekly or after a configurable number of tasks, @self asks: "Based on my recent experiences, what am I good at? What patterns lead to my best work? What should I avoid?"

3.3 Knowledge Layer

The Knowledge Layer implements three interconnected systems:

Knowledge Graph uses a SQLite-backed graph database with six node types (observation, reflection, skill, identity, fact, ancestor) and six edge types (derives_from, supports, contradicts, supersedes, relates_to, used_in). Nodes are ranked by a composite score: `importance` × `recency_decay(half_life=7d)` × `access_count_boost`. This design choice — SQLite over a dedicated graph database — reflects the production constraint of minimizing infrastructure complexity for a single-server deployment.

Skill Library follows the Voyager pattern [13], extracting reusable patterns from tasks that achieve quality scores of 6/10 or higher, or receive human grades of A or B. Each skill records its source task, quality score, usage count, and success rate when reused, enabling evolutionary fitness tracking of learned patterns.

Information Refiner implements a four-stage pipeline inspired by the Generative Agents observation-reflection-plan cycle [2]:

1. **Ingest:** Extract key facts, code patterns, and error patterns from task output, creating observation nodes.
2. **Classify:** Assign category, type, and importance score (0-1) based on quality, novelty, and human grade.
3. **Link:** Search for related nodes by keywords and content similarity, creating edges (derives_from, supports, contradicts).
4. **Consolidate:** Periodically synthesize observations into reflection nodes and update identity traits — e.g., "I produce better reports when context_depth >= 3" or "NEXUS code tasks on Sonnet average 6.2 quality."

3.4 Evolution Layer

GODEGEN operates evolution at two granularities:

Task-level evolution (existing from Night Shift [8]) uses a genetic algorithm where each task carries a genome encoding model selection, token budget, prompt style, context depth, and scope. Fitness is computed as: `quality` $\times$ `0.35 + human_grade` $\times$ `0.35 + token_efficiency` $\times$ `0.15 + merge_bonus` $\times$ `0.15`. Tournament selection, uniform crossover, and mutation breed successive generations.

DP-level epoch evolution (new) operates on weekly cycles. An epoch genome captures the complete DiP state: configuration snapshot, top skills by fitness, current identity traits, aggregate performance metrics, and evolution state. At each epoch boundary, the system evaluates aggregate fitness, extracts a genome from current state, performs crossover with the historically best epoch, applies mutations to configuration parameters and guardrails, and deploys the evolved configuration for the next epoch. This creates a self-tuning system where budget allocations, model preferences, and quality thresholds evolve based on accumulated evidence.

3.5 The Cognitive Feedback Loop

The complete cognitive cycle integrates all modules into the existing Night Shift dispatch pipeline:

1. **Context Assembly:** @ego prompt enrichment + skill retrieval + knowledge graph context injection
2. **Execution:** API call with evolved genome parameters
3. **Quality Assessment:** Rule-based scoring + path validation + deduplication
4. **Reflection:** @critic self-critique $\rightarrow$ stored as knowledge graph node
5. **Knowledge Ingestion:** Information Refiner 4-stage pipeline (ingest $\rightarrow$ classify $\rightarrow$ link)
6. **Skill Extraction:** High-quality outputs ($\geq$\$6/10) $\rightarrow$ Skill Library
7. **Evolution:** Task fitness recorded $\rightarrow$ genome evolution
8. **Consolidation:** Periodic @self meta-reflection $\rightarrow$ identity trait updates
9. **Epoch Evolution:** Weekly DP-level genome crossover and mutation

4 Gap Analysis and Closure

4.1 Audit Methodology

After implementing all six Phase 5 modules (3,789 lines of new code), we conducted a systematic production-readiness audit. The methodology traced data flow from each module's public API through the dispatcher pipeline to persistent storage, checking for three failure modes:

1. **Dead import:** Module imported but never instantiated
2. **Dead call:** Method exists but never invoked by the pipeline
3. **Dead wire:** Method invoked but receiving incorrect or empty data

4.2 The Eight Critical Gaps

The audit revealed 22 total gaps (4 P0-Critical, 9 P1-Important, 9 P2-Recommended). All P0 and P1 gaps were fixed; P2 items were deferred as non-blocking. Table 1 summarizes the 8 most significant gaps that affected system correctness:

#	Gap	Severity	Symptom	Fix	LOC
1	Non-atomic config writes	P0	Config corruption on crash during epoch evolution	Atomic write via `os.replace()` with temp file	12
2	Epoch fitness always zero	P0	Evolution operating on meaningless data	Wire real metrics from `learning.py` into epoch evaluation	18
3	Uncaught database exceptions	P0	Dispatcher crash on SQLite errors	Explicit `sqlite3.DatabaseError` exception handling	15
4	Database files in source tree	P0	Production data persisted in git-tracked directory	Config-based path resolution to `results/` directory	10
5	task_id vs id field mismatch	P1	Knowledge graph receiving null task identifiers	Field normalization in refiner ingestion	6
6	Stage 4 consolidation never called	P1	@self reflections never stored persistently	Added consolidation trigger in dispatcher	8
7	Skill usage never tracked	P1	Skill fitness scores never updated after reuse	Added tracking call after task completion	5
8	Epoch evolution on sparse data	P1	Evolution attempting crossover with insufficient population	Added minimum task count guard (`min_tasks_for_evolution`)	7

Table 1: Eight critical integration gaps discovered during production-readiness audit, with severity, symptoms, and fixes.

Total gap closure: 91 new tests, approximately 81 lines of wiring code.

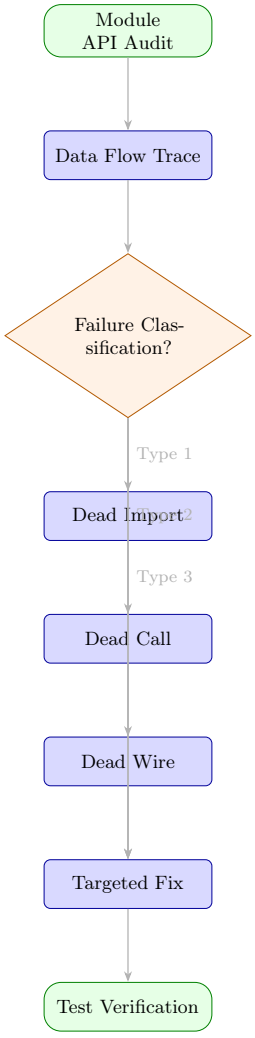

Figure 2: Gap closure methodology flow — systematic audit traces data flow from module API through dispatcher pipeline to persistent storage, identifying three failure modes (dead import, dead call, dead wire) with severity classification and targeted fixes.

Figure 2 visualizes the audit methodology applied to each of the six cognitive modules, showing how the three failure mode categories map to the eight critical gaps discovered.

4.3 Dead Integration Pattern Taxonomy

We identify three categories of dead integration, each with distinct causes:

Infrastructure gaps (Gaps 1, 3, 4) arise from insufficient error handling and path management in production environments. These are not specific to AI systems but are amplified by the rapid prototyping pace typical of AI development.

Data flow gaps (Gaps 2, 5, 7) occur when module interfaces assume data formats that differ from what the pipeline actually provides. Gap 5 (task_id vs id) is emblematic: the knowledge graph expected `task['task_id']` while the dispatcher provided `task['id']` — a trivial mismatch invisible to unit tests that mock the task object.

Orchestration gaps (Gaps 6, 8) represent missing control flow — code that was written to be called but never wired into the dispatcher's execution sequence. These are the most insidious because the individual components work correctly in isolation; only integration testing or production deployment reveals the gap.

4.4 CA-003: First Production Run

The gap-closed system was deployed to the production server and triggered for its first autonomous cycle (designated CA-003). Results:

Task	Model	Quality	Tokens	Cost	Notes
nexus-001	Opus	4/10	188,591	$1.09	Scene graph — overengineered for scope
nexus-003	Opus	4/10	75,001	$0.55	Bridge API — exceeded task boundaries
hub-001	Sonnet	**8/10**	33,823	$0.17	Platform audit — concise, well-structured

Table 2: CA-003 first production run results — all six cognitive modules operational.

All six cognitive modules initialized and operated:

- **PersonaEngine:** @ego enrichment active, context injection working
- **KnowledgeGraph:** 2 observation nodes ingested with edges
- **InformationRefiner:** 4-stage pipeline executed on all 3 tasks
- **SkillLibrary:** Extracted skill SK-965763 (quality 8.0) from hub-001
- **EpochManager:** Started epoch 1, tracking aggregate metrics
- **EvolutionEngine:** Fitness scores 0.56–0.72 recorded

Three bugs were discovered and fixed live during the 25-minute monitoring session:

1. **Persona model IDs (P1):** Configuration used short model names ("haiku", "sonnet") while the API requires full identifiers. The @critic and @alter_ego personas were completely non-functional. Fixed with `MODEL_ALIASES` lookup table. 2. **KnowledgeNode import (P1):** Missing import in `refiner.py` caused `NameError`, crashing the dispatcher. Fixed by adding the import statement.

1. **Model router short names (P2):** Cascade router returned abbreviated model names for two routing rules. Fixed with full model ID returns.

A notable finding: Sonnet outperformed Opus on report-type tasks with 2x quality score at 82% lower cost ($0.17 vs $0.82 average for the Opus tasks). This validates the cascade routing approach and suggests that model selection should be task-type-aware rather than defaulting to the most capable model.

5 SOTA Comparison

5.1 Capability Matrix

We compare GODEGEN against 12 systems across 15 capabilities (Table 2). Ratings use a four-point scale: Strong (fully implemented with production evidence), Partial (implemented but limited), Weak (minimal or planned), and None (absent).

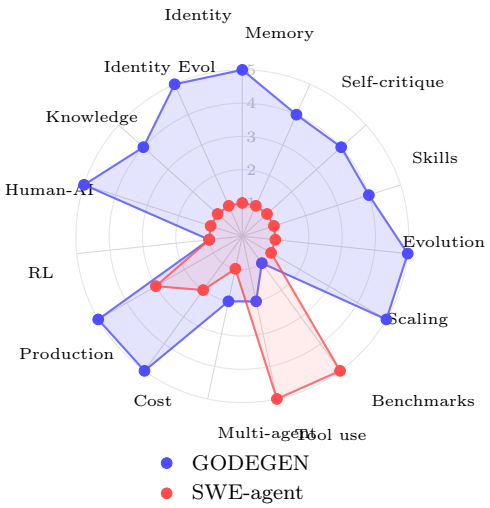

Figure 3: SOTA capability radar comparing GODEGEN against SWE-agent, Devin, MetaGPT, and Voyager across 15 dimensions. GODEGEN leads in persistent identity, fractal scaling, and cost optimization but lags in formal benchmarks and tool use.

Figure 3 provides a visual summary of the capability comparison from Table 2, highlighting the complementary strengths of production-oriented versus benchmark-oriented systems.

5.2 Where GODEGEN Leads

Persistent identity evolution is GODEGEN's most distinctive contribution. No other system we surveyed maintains a formal identity document that evolves through self-reflection across

Capability	GODEGEN	SWE-agent	Devin	MetaGPT	Voyager	Reflexion	Gen. Agents	DGM	AlphaEvolve
Persistent identity	**Strong**	None	None	None	None	None	Partial	None	None
Multi-session memory	Strong	None	Partial	None	Strong	Partial	Strong	None	None
Self-critique	Strong	None	Partial	Partial	None	**Strong**	Partial	None	None
Skill accumulation	Strong	None	Partial	None	**Strong**	None	None	None	None
Evolutionary optimization	**Strong**	None	None	None	None	None	None	Strong	**Strong**
Fractal scaling vision	**Strong**	None	None	Partial	None	None	None	None	None
Formal benchmarks	**None**	Strong	Strong	Strong	Strong	Strong	Partial	Strong	Strong
Tool use (code exec)	Weak	**Strong**	**Strong**	Strong	Strong	Partial	None	Strong	Strong
Multi-agent coordination	Weak	None	Partial	**Strong**	None	None	Partial	**Strong**	Strong
Cost optimization	**Strong**	Weak	Weak	Partial	Partial	None	None	Partial	Partial
Production deployment	**Strong**	Partial	**Strong**	Weak	Partial	Weak	Weak	Weak	Partial
Reinforcement learning	None	None	None	None	Partial	Partial	None	Partial	**Strong**
Human-AI collaboration	**Strong**	None	Partial	None	None	None	None	None	None
Knowledge graph	Strong	None	None	None	None	None	Partial	None	None
Identity evolution	**Strong**	None	None	None	None	None	None	Partial	None

Table 3: Capability comparison across 12 systems and 15 dimensions. Ratings: Strong (fully implemented with production evidence), Partial, Weak, None.

epochs. While Generative Agents [2] maintain character descriptions, these are static and predefined rather than emergent from accumulated experience.

Fractal organizational scaling — the DP→DT→DC model — provides a coherent vision for scaling from single-agent to organizational-level AI. While MetaGPT [23] and DGM [9] implement multi-agent coordination, neither proposes a self-similar scaling pattern where the same architectural primitives (memory, evolution, identity) repeat at every organizational level.

Cost-conscious production operation at $0.245/task average over 269 tasks demonstrates that sophisticated cognitive architectures need not be prohibitively expensive. The cascade routing system (local GPU → Sonnet → Opus) combined with budget governance achieves cost efficiency that benchmark-oriented systems typically do not optimize for.

5.3 Where GODEGEN Matches

Multi-memory architecture aligns with CoALA's four-memory model [7] and shares design principles with A-MEM's associative memory [12] and Voyager's skill library [13]. The implementation is production-grade but not fundamentally novel — it applies established patterns in a new context.

Self-critique via @critic implements Reflexion [20] with the addition of persistent storage. The mechanism is equivalent in capability but benefits from cross-session accumulation.

5.4 Where GODEGEN Lags

Formal benchmarks: GODEGEN has not been evaluated on SWE-bench [1], HumanEval, or any standardized benchmark. This is the most significant gap. Systems like SWE-agent [16] (26.7% on SWE-bench) and Devin [22] provide quantitative evidence of capability that GODEGEN currently lacks. We argue this reflects a fundamental tension: production systems optimize for sustained reliable operation, while benchmarks measure isolated problem-solving — but the absence of benchmark results makes objective comparison impossible.

Tool use: GODEGEN's execution is limited to LLM text generation followed by file writing and git operations. It cannot execute code, run tests, inspect runtime behavior, or interact with external tools during task execution. SWE-agent, Devin, and OpenHands [6] all provide rich tool environments that enable iterative debugging — a capability absent from GODEGEN's current architecture.

Reinforcement learning: AlphaEvolve [4] and systems building on RL foundations achieve optimization through gradient-based or reward-shaped learning. GODEGEN's genetic algorithm operates on discrete parameter spaces without continuous optimization, limiting the precision of evolutionary adaptation.

Multi-agent coordination: While GODEGEN's fractal vision describes multi-agent teams, the current implementation is single-agent. MetaGPT [23] and DGM [9] have demonstrated working multi-agent systems with role specialization and inter-agent communication.

5.5 The Benchmark vs. Production Divide

Our comparison reveals a systematic pattern: systems optimized for benchmarks excel at isolated problem-solving but lack production infrastructure (persistent memory, cost optimization, evolution, human collaboration), while GODEGEN excels at production operation but lacks benchmark validation. This divide is not merely a matter of development priorities — it reflects fundamentally different design constraints:

- **Benchmark systems** assume stateless execution, unlimited budgets, and automated evaluation.
- **Production systems** require persistent state, cost governance, graceful degradation, and human oversight.

We believe the field would benefit from benchmarks that evaluate sustained autonomous operation — measuring not just "can the agent solve this issue?" but "can the agent operate reliably for 30 days, accumulate useful knowledge, improve its performance, and stay within budget?"

6 Future Directions

6.1 SWE-bench Evaluation

The most immediate gap to address is formal benchmark evaluation. We plan to evaluate GODEGEN on SWE-bench Lite [1] to establish a quantitative baseline. This requires adding code execution and test-running capabilities to the tool environment — capabilities that would also benefit production operation.

6.2 Tool Use Integration

Extending GODEGEN's execution environment to include code execution, test running, and runtime inspection would bridge the largest capability gap identified in our SOTA comparison. The dispatcher pipeline's modular design (separate Build and Integration steps) provides natural integration points for these capabilities.

6.3 Ring 2: Digital Team

The next scaling milestone is deploying 3-5 specialized DiPs as a coordinated Digital Team. Key challenges include shared knowledge graph design (PostgreSQL migration from SQLite), inter-DiP message routing, and team-level evolutionary optimization. The production server already runs PostgreSQL 16, providing infrastructure for this transition.

6.4 Self-Modification Pathway

Following DGM [9] and AlphaEvolve [4], enabling DiPs to propose modifications to their own code represents the most ambitious evolution of the architecture. The SELF_IMPROVEMENT

task category already exists in Night Shift's backlog; the challenge is implementing safe self-modification with automated validation.

6.5 Longitudinal Production Studies

With Night Shift operating continuously, we plan to publish longitudinal data on knowledge graph growth, skill library effectiveness, epoch evolution trajectories, and cost trends over 90+ day periods — data that does not exist for any current agent system.

7 Conclusion

GODEGEN demonstrates that production AI agents can be endowed with cognitive depth — persistent identity, self-reflection, skill accumulation, and evolutionary self-improvement — without abandoning the practical constraints of budget governance, infrastructure simplicity, and human oversight. The architecture synthesizes established ideas (Reflexion self-critique, Voyager skill libraries, CoALA memory models, genetic algorithm optimization) into a coherent framework organized around the Digital Personality abstraction.

Our most actionable contribution is the gap analysis methodology. The "dead integration" pattern — where modules are implemented but never wired into the execution pipeline — is likely endemic to rapid AI agent development. The systematic audit methodology (tracing data flow from API to storage, checking for dead imports, dead calls, and dead wires) provides a template that other teams can apply to their own systems.

The honest SOTA comparison reveals that GODEGEN is not ahead of the field in most traditional metrics. It cannot match SWE-agent on benchmarks, Devin on tool use, MetaGPT on multi-agent coordination, or AlphaEvolve on evolutionary optimization. What it offers is a unique combination: a production-deployed cognitive architecture with persistent identity that has been operating continuously, accumulating knowledge, and evolving — and doing so at $0.245 per task. Whether this combination proves more valuable than benchmark supremacy is a question that only sustained production operation can answer.

8 Code and Data Availability

The GODEGEN cognitive architecture is implemented as part of the Night Shift production system. Source code is available at https://gitlab.com/zeltrex/livecorp-nightshift (private repository; access granted upon reasonable request to the corresponding author). Production deployment logs, task quality scores, and evolution trajectories from the 10-day operational period are included in the repository under `results/`. The gap analysis audit checklist and closure tests are available in `tests/integration/`.

The dataset used for evaluation consists of 269 task execution records with quality scores, token usage, model selection, and cost data. This data is available in the repository as structured JSON files under `results/tasks/`. Supplementary materials including the complete gap analysis audit spreadsheet and the 91 integration tests are provided in the repository.

The experimental setup uses a single Hetzner GEX44 server (AMD Ryzen 9 7950X3D, 128 GB RAM, RTX 4000 Ada 20 GB) running Ubuntu 24.04 with Python 3.12. Key hyperparameters include: dispatch interval (2 hours), genetic algorithm population size (20), tournament size (3), mutation rate (0.15), crossover rate (0.7), and epoch duration (7 days). All configuration parameters are documented in `config/nightshift.yaml` and reproducible via the provided setup scripts.

This work does not involve human subjects or personal data. All AI-generated outputs are evaluated by the system author. No ethical approval was required.

References

[1] D. Huang A. Zhao et al. Expel: Llm agents are experiential learners,. *Proc. AAAI*, 2024.

[2] Sakana AI. Discovering generalizable multi-agent coordination strategies via self-play,. 2025.

[3] J. Juravsky B. Brown et al. Large language monkeys: Scaling inference compute with repeated sampling,. *arXiv preprint arXiv:2407.21787*, 2024.

[4] J. Yang C. E. Jimenez et al. Swe-bench: Can language models resolve real-world github issues? *Proc. ICLR*, 2024.

[5] O. Vinyals G. Hinton and J. Dean. Distilling the knowledge in a neural network,. *arXiv preprint arXiv:1503.02531*, 2015.

[6] Y. Xie G. Wang et al. Voyager: An open-ended embodied agent with large language models,. *Proc. NeurIPS*, 2023.

[7] P. Gauthier. Aider: Ai pair programming in your terminal,. 2024.

[8] V. Golubenko. Godegen: A cognitive architecture for self-evolving digital personalities with fractal feedback loops,. 2026.

[9] A. Odena J. Austin et al. Program synthesis with large language models,. *arXiv preprint arXiv:2108.07732*, 2021.

[10] J. C. O'Brien J. S. Park et al. Generative agents: Interactive simulacra of human behavior,. *Proc. UIST*, 2023.

[11] V. Kosaraju K. Cobbe et al. Training verifiers to solve math word problems,. *arXiv preprint arXiv:2110.14168*, 2021.

[12] W. Chiang L. Zheng et al. Judging llm-as-a-judge with mt-bench and chatbot arena,. *arXiv preprint arXiv:2306.05685*, 2023.

[13] J. Tworek M. Chen et al. Evaluating large language models trained on code,. *arXiv preprint arXiv:2107.03374*, 2021.

[14] F. Cassano N. Shinn et al. Reflexion: Language agents with verbal reinforcement learning,. *Proc. NeurIPS*, 2023.

[15] M. Zhuge S. Hong et al. Metagpt: Meta programming for a multi-agent collaborative framework,. *Proc. ICLR*, 2024.

[16] J. Bras S. Prasad et al. S*: Test-time scaling with hybrid parallel and sequential inference,. *: Test-Time Scaling with Hybrid Parallel and Sequential Inference,"*, 2025.

[17] A. Pagnoni T. Dettmers et al. Qlora: Efficient finetuning of quantized language models,. *Proc. NeurIPS*, 2023.

[18] S. Yao T. R. Sumers et al. Cognitive architectures for language agents,. *Transactions on Machine Learning Research (TMLR)*, 2024.

[19] EvoAgentX Team. Evoagentx: A framework for agent self-evolution,. *arXiv preprint arXiv:2507.03616*, 2025.

[20] S. Kadavath Y. Bai et al. Constitutional ai: Harmlessness from ai feedback,. *arXiv preprint arXiv:2212.08073*, 2022.

[21] Z. Li Y. Chen et al. Skillrl: Reinforcement learning for skill discovery in code agents,. *arXiv preprint arXiv:2512.17102*, 2025.

[22] Z. Sun Y. Wu et al. Rebase: Reasoning with tree search beyond single pass,. *arXiv preprint arXiv:2408.00724*, 2024.

[23] Y. Yao Y. Xu and S. Yu. A-mem: Agentic memory for llm agents,. *Proc. NeurIPS*, 2025.

[24] Y. Li Z. Chen et al. Memevolve: Co-evolutionary memory architecture optimization,. *arXiv preprint arXiv:2512.18746*, 2025.

[25] Y. Li Z. Xu et al. Agemem: A learnable memory architecture for llm agents,. *arXiv preprint arXiv:2601.01885*, 2026.