

# Helical Pipeline Architecture for Autonomous Multi-Agent SaaS Development

Thomas Perry Jr.

Pastoral Tech · Independent Researcher

Dallas, Texas

February 2026

---

## Abstract

We present the *Helical Pipeline Architecture* (HPA), a compounding parallel execution model for autonomous AI agent teams building compound SaaS platforms. Unlike flat ring pipelines or sequential phase gates, HPA organizes work as a helix: four concurrent stage-groups (Forge, Define, Build, Verify) form a single revolution, and each completed revolution elevates products to a higher-order layer where the same four stages execute against progressively richer integration targets. At steady state the system achieves sustained parallelism across both stages and layers, with products at Layer  $\ell$  undergoing integration testing while products at Layer  $\ell-1$  are still being built and products at Layer  $\ell+1$  are being scaffolded for AI-powered refinement. The architecture incorporates gradient auto-compaction for agent context management, DAG-governed task dependencies, theological guardrail enforcement, and a formal state machine that guarantees the helical invariant: no product may enter Layer  $\ell+1$  without graduating all four stages of Layer  $\ell$ . We describe the architecture in the context of Pastoral Tech, a compound startup building 20+ interconnected ministry SaaS products atop a shared Unified Congregant Record, though the model generalizes to any multi-product platform with shared data infrastructure.

**Keywords:** multi-agent systems, pipeline parallelism, compound SaaS, autonomous development, helical architecture, AI orchestration, Claude Code, constitutional AI governance

## Introduction

---

Building a compound SaaS platform—one where 20+ products share authentication, billing, theming, and a unified data record—presents a scheduling problem that neither fully sequential nor fully parallel execution models solve well. Sequential phase gates (build all products in Phase 1, then all in Phase 2) waste pipeline capacity: scaffolding agents sit idle while build agents work. Fully parallel execution (launch all products simultaneously) creates resource contention and violates dependency constraints when products must integrate with each other.

We propose a third model: the *helical pipeline*, where products flow through a four-stage cycle that repeats at increasing levels of integration maturity. The key insight is that the “merry-go-round” metaphor fails because a ring returns to its starting position. A helix never does—each revolution deposits the product at a strictly higher altitude, and the work performed at each stage evolves with the layer.

## Contributions

This paper makes the following contributions:

- A formal definition of the Helical Pipeline Architecture including its state space, transition function, and invariants (Section 4).
- A concrete instantiation for autonomous AI agent orchestration using Claude Code subagent teams with gradient auto-compaction (Section 6).
- A layer taxonomy defining the work performed at each altitude of the helix (Section 5).
- A TypeScript orchestration engine specification that implements the helical state machine programmatically (Section 9).
- An application to Pastoral Tech’s 20+ product compound startup with theological governance constraints (Section 10).

## Related Work

---

### CI/CD Pipeline Architectures

Traditional continuous integration and deployment systems—Jenkins [1], GitLab CI [2], and similar platforms—model software delivery as a directed acyclic graph of stages (build, test, deploy). These architectures are optimized for single-product workflows and treat the pipeline as a one-shot traversal: code enters, artifacts exit. They do not model the iterative deepening required when products must integrate with each other across multiple passes.

### Multi-Agent Orchestration Frameworks

Recent work in LLM-based multi-agent systems has produced frameworks such as AutoGen [3], CrewAI [4], and CAMEL [5]. These systems excel at coordinating small teams of agents on bounded tasks but lack a topology for sustained, multi-product development. They typically assume a single conversation thread or a flat task queue, neither of which addresses the compounding integration requirements of a compound SaaS platform.

### LLM Context Management

The finite context window of large language models creates a fundamental constraint on agent longevity. Prior work has addressed this through retrieval-augmented generation (RAG) [6], recursive summarization [7], and memory systems [8]. Our gradient auto-compaction protocol extends these ideas by treating context management as a continuous resource allocation problem rather than a binary “in-context vs. retrieved” decision.

### Positioning of HPA

HPA differs from prior work in three ways: (1) it models execution as a helix rather than a DAG or ring, enabling compounding integration; (2) it assigns permanent teams to stages rather than agents to products, maximizing pipeline utilization; (3) it treats context compaction as a first-class architectural concern with graduated thresholds.

## Architecture Overview

### The Helix as Execution Topology

Consider a standard four-stage pipeline with stages  $\mathcal{S} = \{\text{FORGE}, \text{DEFINE}, \text{BUILD}, \text{VERIFY}\}$ . In a flat ring, a product  $p$  traverses  $\text{FORGE} \rightarrow \text{DEFINE} \rightarrow \text{BUILD} \rightarrow \text{VERIFY}$  exactly once. In the helix, the product traverses this cycle at Layer  $\ell = 0, 1, 2, \dots, L$  where each layer represents a qualitatively different scope of work.

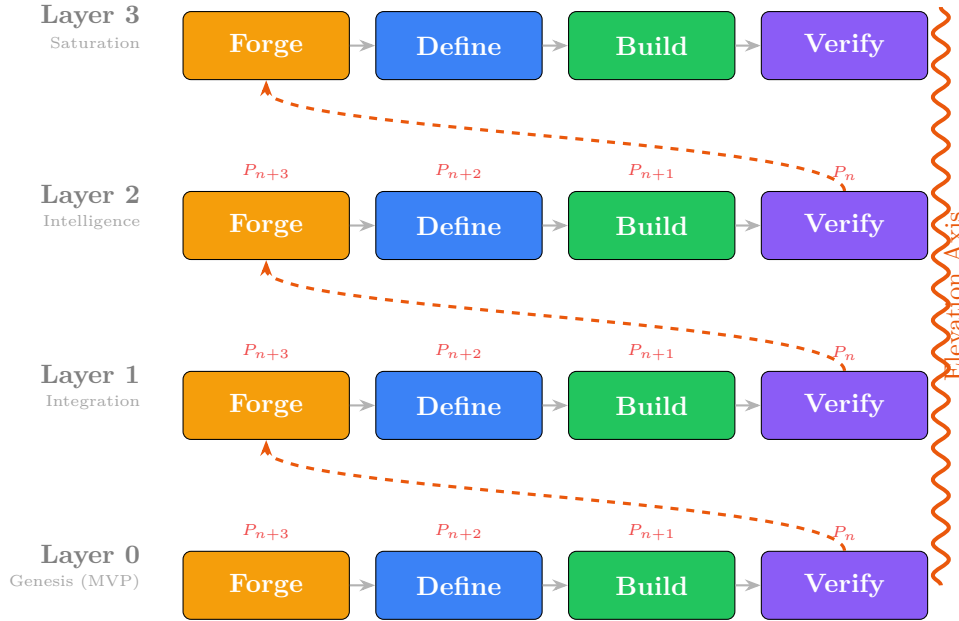


Figure 1: The helical pipeline. Each horizontal row is a four-stage pipeline at a given layer. Vertical connections show products elevating from VERIFY at Layer  $\ell$  to FORGE at Layer  $\ell+1$ . At steady state, four products occupy four stages across multiple layers simultaneously.

### Contrast with Prior Models

Property	Sequential Phases	Flat Ring	Helix (Ours)
Pipeline utilization	Low (idle stages)	High (all stages active)	Maximal (all stages, all layers)
Integration depth	Deferred to late phases	Single pass	Compounding per revolution
Recovery from failure	Restart phase	Re-enter ring	Re-enter at failed layer
Agent context budget	Large (long phases)	Medium	Small (bounded per stage)
Cross-product data flow	Absent until integration phase	Not modeled	Native at Layer $\geq 1$

Table 1: Comparison of execution topologies for multi-product SaaS development.

## Formal Model

---

### State Space

**Definition 1** (Product State). *A product  $p$  at any point in time occupies a state  $(p, \ell, s)$  where:*

- $p \in \mathcal{P}$  is the product identifier,
- $\ell \in \{0, 1, 2, \dots, L\}$  is the layer (altitude on the helix),
- $s \in \mathcal{S} = \{\text{FORGE}, \text{DEFINE}, \text{BUILD}, \text{VERIFY}\}$  is the stage.

*The global system state is the set  $\Sigma = \{(p_i, \ell_i, s_i)\}_{i=1}^{|\mathcal{P}|}$ .*

**Definition 2** (Stage Ordering). *Stages are totally ordered within a layer:*

$$\text{FORGE} \prec \text{DEFINE} \prec \text{BUILD} \prec \text{VERIFY}$$

*We write  $\text{next}(s)$  for the successor stage, with  $\text{next}(\text{VERIFY}) = \text{FORGE}$  at layer  $\ell + 1$ .*

### Transition Function

**Definition 3** (Stage Transition). *The transition function  $\delta : \Sigma \times \mathcal{E} \rightarrow \Sigma$  maps system states and events to new system states. For a product  $p$  at state  $(\ell, s)$ :*

$$\delta(p, \ell, s, \text{stage\_complete}) = \begin{cases} (p, \ell, \text{next}(s)) & \text{if } s \neq \text{VERIFY} \\ (p, \ell + 1, \text{FORGE}) & \text{if } s = \text{VERIFY} \text{ and } \ell < L \\ (p, \ell, \text{DONE}) & \text{if } s = \text{VERIFY} \text{ and } \ell = L \end{cases}$$

### Invariants

**Invariant 1** (Helical Monotonicity). *For any product  $p$ , the layer  $\ell$  is monotonically non-decreasing over time:*

$$\forall t_1 < t_2 : \ell_p(t_1) \leq \ell_p(t_2)$$

*A product never descends on the helix.*

**Invariant 2** (Layer Gate). *A product  $p$  may enter Layer  $\ell + 1$  if and only if it has completed all four stages at Layer  $\ell$ :*

$$(p, \ell + 1, \text{FORGE}) \text{ is reachable} \iff \forall s \in \mathcal{S} : (p, \ell, s) \text{ was completed}$$

**Invariant 3** (Stage Concurrency Bound). *At most  $k$  products may simultaneously occupy the same stage  $s$  at any layer:*

$$|\{p \in \mathcal{P} : s_p = s\}| \leq k \quad \text{where } k = \text{MAX\_CONCURRENT\_AGENTS}/|\mathcal{S}|$$

**Invariant 4** (Foundation Precedence). *No product may enter Layer 0 until the shared infrastructure (core package) has passed its own verification:*

$$\forall p \in \mathcal{P} : (p, 0, \text{FORGE}) \text{ requires } \text{core\_verified} = \text{true}$$

## Parallelism Analysis

**Theorem 1** (Steady-State Pipeline Width). *At steady state with  $n$  products and  $L + 1$  layers, the maximum number of concurrent active work units is:*

$$W_{\max} = \min(n, |\mathcal{S}| \times (L + 1)) = \min(n, 4(L + 1))$$

*Proof.* Each of the  $|\mathcal{S}| = 4$  stages at each of  $L + 1$  layers can host at least one product. Assuming  $n \geq 4(L + 1)$ , every slot is occupied. Otherwise,  $n$  bounds the parallelism. Subject to Invariant 3, the actual width is  $\min(n, 4(L + 1), k \cdot |\mathcal{S}|)$ .  $\square$

## Layer Taxonomy

The helix is not merely “the same work repeated.” Each layer represents a qualitatively distinct scope. The four stages are structurally identical but their *semantics* evolve with altitude.

### Layer 0: Genesis

The founding revolution. Products enter as empty directory stubs and exit as standalone MVPs with no cross-product dependencies.

Stage	Work at Layer 0
FORGE	Stamp product skeleton from template. Wire <code>@pastoral-tech/core</code> imports. Configure <code>tailwind.config.ts</code> (dark mode default). Create directory structure.
DEFINE	Product lead defines feature set, Prisma schema extensions, UCR read/write contracts, task DAG with dependencies.
BUILD	Backend agent implements API routes and business logic. Frontend agent builds pages and components (shadcn/ui). Both write tests.
VERIFY	Sentinel runs 12-point drift audit. QA agent validates UCR contract compliance. Type-check, lint, test coverage gates.

### Layer 1: Integration

Products that graduated Layer 0 re-enter the helix. The focus shifts from internal functionality to cross-product data flows through the UCR.

Stage	Work at Layer 1
FORGE	Scaffold integration adapters: event bus listeners, UCR enrichment hooks, webhook endpoints for sibling products.
DEFINE	Map cross-product data flows. E.g., Steward IQ detects financial distress $\rightarrow$ triggers care alert in Shepherd HQ. Define event schemas and contracts.
BUILD	Implement event producers/consumers. Build UCR enrichment pipelines (product A writes scores that product B reads). Integration test harnesses.
VERIFY	End-to-end integration tests across product pairs. Verify UCR data flows bidirectionally. Theological guardrail consistency across linked products.

## Layer 2: Intelligence

Products with verified integrations re-enter for AI-powered feature elevation. The “refuse rather than hallucinate” constraint becomes load-bearing at this layer.

Stage	Work at Layer 2
FORGE	Scaffold ML/AI pipelines: prediction model stubs, prompt templates, Claude API integration points, denominational constraint injection.
DEFINE	Specify AI features: churn prediction (Steward IQ), burnout detection (Equip AI), group matching algorithms (Gather AI). Define training data schemas from UCR.
BUILD	Implement prediction models, prompt chains with constitutional constraints, A/B testing infrastructure. Build denomination-aware output filters.
VERIFY	Validate model outputs against theological guardrails per denomination mode. Test “refuse rather than hallucinate” behavior. Accuracy benchmarks.

## Layer 3: Saturation

The final standard layer. Products receive denomination-specific modules, marketplace readiness, and operational hardening.

Stage	Work at Layer 3
FORGE	Scaffold denomination modules (Reformed, Wesleyan, Catholic, Pentecostal, Baptist, Lutheran, Anglican). Marketplace listing templates.
DEFINE	Define denomination-specific behavior deltas. Map liturgical calendar integrations. Specify compliance requirements per tradition.
BUILD	Implement denomination-specific guardrail configurations, UI variants, content libraries. Build marketplace APIs and billing tiers.
VERIFY	Cross-denomination regression testing. Verify no theological mode leakage between tenants. Load testing. Security audit. Production readiness.

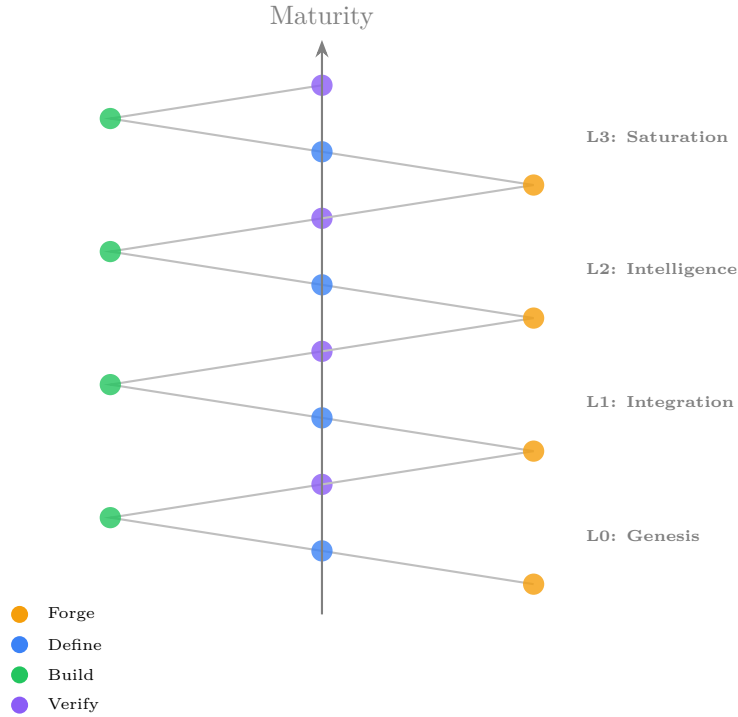


Figure 2: Three-dimensional projection of the helix. Each point represents a stage execution; connected path shows a single product’s trajectory from Layer 0 Genesis through Layer 3 Saturation. The vertical axis represents increasing integration maturity.

## Agent Team Architecture

### Stage-Owning Teams

Rather than assigning agents per-product (which creates idle time), the helical model assigns *permanent teams to stages*. Each team is expert in its stage’s work and processes whatever product the helix delivers to it.

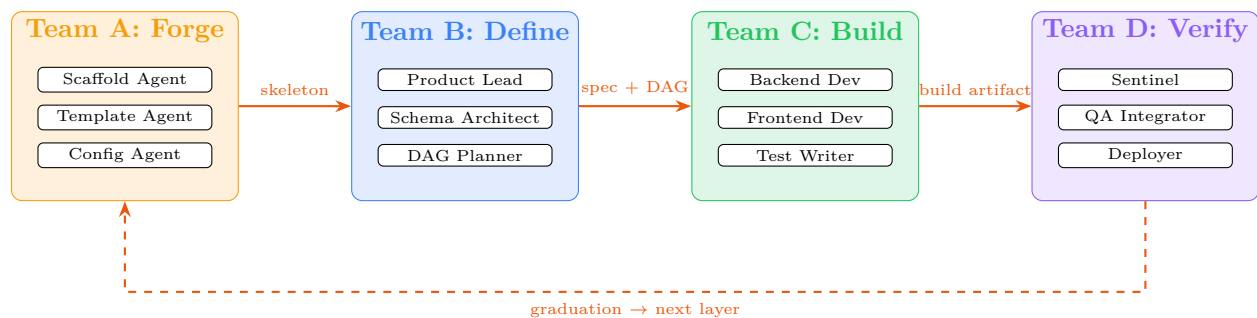


Figure 3: Stage-owning agent teams. Each team processes products as they arrive at its stage. The feedback loop from Verify to Forge represents layer elevation: graduated products re-enter at the next layer.

## Agent Roles by Stage

**Team A — Forge:** A lightweight team focused on speed. The Scaffold Agent stamps the product directory from the shared template. The Template Agent customizes boilerplate (product name, UCR hooks, theme config). The Config Agent wires dependencies and verifies compilation.

**Team B — Define:** The strategic team. The Product Lead defines the feature set and priorities. The Schema Architect designs Prisma schema extensions that integrate with the UCR. The DAG Planner decomposes the feature set into a dependency-ordered task graph.

**Team C — Build:** The largest team. The Backend Dev implements API routes, business logic, and service layer methods. The Frontend Dev builds pages and components using shadcn/ui with dark mode. The Test Writer creates unit and integration tests for both. All three work concurrently on non-overlapping file sets.

**Team D — Verify:** The quality gate. The Sentinel runs the 12-point drift detection protocol (type errors, lint, UCR compliance, theological guardrails, dark mode, no emoji). The QA Integrator runs integration tests and validates cross-product data flows. The Deployer packages build artifacts and writes deployment manifests.

## Concurrency Model

### Concurrency Budget

```
MAX_CONCURRENT_AGENTS = 6

Allocation:
  Slot 1: Sentinel (always running)
  Slots 2-3: Active stage team (2 agents)
  Slots 4-6: Product build team (3 agents)

When a product transitions between stages:
  - Outgoing team completes and releases slots
  - Incoming team spawns into freed slots
  - Sentinel never releases its slot
```

## Gradient Auto-Compaction

### The Context Window Problem

Each AI agent operates within a finite context window. Claude Opus 4.6 provides approximately 200,000 tokens of context. A long-running agent (Sentinel, Foundation) will exhaust this budget, requiring a strategy for graceful degradation that does not lose state.

### Five-Threshold Gradient Model

We define five context utilization thresholds that trigger progressively aggressive state management:



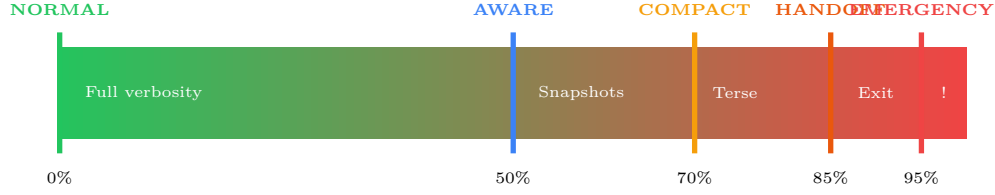


Figure 4: Gradient auto-compaction thresholds. Agent behavior degrades gracefully as context utilization increases, preserving state at each threshold.

**NORMAL (0–50%):** Full reasoning verbosity. Rich explanatory output. No special behavior.

**AWARE (50–70%):** Agent begins writing incremental state snapshots to disk after every completed task. Snapshots include decisions made, files modified, patterns established, and remaining work.

**COMPACT (70–85%):** Agent enters efficiency mode. Responses become terse—no explanatory prose. All reasoning moves to disk-based notes. Agent writes a comprehensive handoff document containing: complete work summary, exact in-progress state, all architectural decisions, file manifest, remaining tasks, and known blockers.

**HANDOFF (85–95%):** Agent signals the Orchestrator via the team messaging system, completes its current atomic task, and gracefully exits. The Orchestrator reads the handoff document and spawns a fresh continuation agent with the handoff as its initial context.

**EMERGENCY (95%+):** Agent force-writes a minimal state dump and exits immediately. This is the fallback when the agent could not complete a clean handoff.

## Continuation Protocol

When the Orchestrator receives a compaction handoff:

---

### Algorithm 1 Agent Continuation Protocol

---

```

1: procedure HANDLECOMPACTION(agent, handoffDoc)
2:   doc ← READFILE(handoffDoc)
3:   role ← agent.role
4:   tasks ← doc.remainingTasks
5:   SPAWNAGENT(role, doc, tasks)
6:   LOG("Continuation spawned for " + role)
7: end procedure

```

---

The continuation agent receives the full handoff document as its initial prompt, verifies all referenced files exist on disk, and resumes work exactly where its predecessor stopped. The chain continues indefinitely—each agent eventually compacts and hands off to the next.

## Orchestrator Self-Compaction

The Orchestrator itself is subject to context limits. When it reaches 70%, it writes a master state document containing all active agents, all task states, current phase, checkpoint summaries, and drift resolutions. It then prints terminal instructions for the human operator to start a new session

with the continuation prompt. This is the *only* point requiring human interaction—all background agents continue uninterrupted in their tmux sessions.

## Cross-Layer State Inheritance

---

### The Compounding Effect

The defining property of the helix is that each layer inherits *all* verified state from every layer below it. A product at Layer 2 has:

- Layer 0 state: standalone MVP (routes, components, tests, UCR contracts)
- Layer 1 state: cross-product integrations (event bus, enrichment pipelines, integration tests)
- Layer 2 state: AI-powered features (predictions, prompt chains, guardrail validation)

This compounding is not merely additive—it is multiplicative. Layer 2 AI features can leverage Layer 1 integrations (e.g., Steward IQ’s churn prediction uses Equip AI’s volunteer data via a Layer 1 enrichment pipeline). Each revolution expands the product’s capability surface nonlinearly.

### State Inheritance Graph

For a product  $p$  at layer  $\ell$ , the inherited state set is:

$$\mathcal{I}(p, \ell) = \bigcup_{k=0}^{\ell-1} \mathcal{V}(p, k)$$

where  $\mathcal{V}(p, k)$  is the verified output of product  $p$  at layer  $k$ .

The FORGE stage at layer  $\ell > 0$  differs fundamentally from FORGE at layer 0: instead of stamping a blank skeleton, it scaffolds *extensions* onto an already-verified product, using  $\mathcal{I}(p, \ell)$  as its foundation.

### Integration Surface Expansion

At Layer 0, a product’s integration surface consists of its UCR read/write contracts—a bounded set of data types. At Layer 1, the surface expands to include event schemas, webhook endpoints, and enrichment hooks from connected products. At Layer 2, it further expands to include AI model inputs/outputs and prompt injection points. The integration surface grows as  $O(\ell \cdot d)$  where  $d$  is the average product connectivity degree.

## Orchestration Engine

---

### Architecture

The orchestration engine is implemented as a TypeScript application that manages the helical state machine programmatically. It consists of five core modules:

**Helix State Machine:** Tracks every product’s position  $(p, \ell, s)$  on the helix. Validates transition legality. Emits events on state changes. Persists state to JSON checkpoints on disk.

**Pipeline Scheduler:** Manages the four concurrent stage slots. Assigns products to available slots based on priority (Tier 1 > Tier 2 > Tier 3). Prevents resource exhaustion by enforcing the concurrency bound.

**Agent Lifecycle Manager:** Spawns agents with role-specific prompts via the Claude Code Task API. Monitors context utilization thresholds. Handles compaction handoffs by reading handoff documents and spawning continuations.

**Revolution Tracker:** Tracks cross-layer state. Detects when all products complete a layer (triggering next-layer unlock). Computes helix metrics: throughput, bottleneck identification, velocity per stage.

**Claude Code Bridge:** Translates helical state machine operations into Claude Code API calls. Generates `Task()` invocations with helix-aware prompts. Manages the `Teammate()` messaging system for inter-agent coordination.

## Main Orchestration Loop

---

### Algorithm 2 Helical Pipeline Orchestration

---

```

1: procedure ORCHESTRATORLOOP
2:   helix ← INITHELIXSTATE( $\mathcal{P}, L$ )
3:   scheduler ← INITSCHEUDLER( $k$ )
4:   SPAWNSENTINEL
5:   BUILDFOUNDATION ▷ Layer -1: Core package
6:   while  $\neg$ ALLPRODUCTSDONE(helix) do
7:     events ← POLLINBOX
8:     for  $e \in events$  do
9:       if  $e.type = stage\_complete$  then
10:        HELIXTRANSITION(helix,  $e.product$ )
11:       else if  $e.type = compaction\_handoff$  then
12:        SPAWNCONTINUATION( $e.agent, e.handoffDoc$ )
13:       else if  $e.type = drift\_critical$  then
14:        HALTPRODUCT( $e.product$ )
15:       end if
16:     end for
17:     ready ← GETSCHEDULABLEPRODUCTS(helix)
18:     for  $p \in ready$  do
19:       if HASCAPACITY(scheduler,  $p.stage$ ) then
20:        SPAWNSTAGETEAM( $p, p.layer, p.stage$ )
21:       end if
22:     end for
23:     WRITECHECKPOINT(helix)
24:     CHECKSELFCOMPACTION
25:   end while
26: end procedure

```

---

## Case Study: Pastoral Tech

---

### Platform Overview

Pastoral Tech is a compound SaaS startup building the operating system for American church operations. The platform comprises 20+ interconnected products targeting church leadership (senior pastors, executive pastors, ministry directors) across every operational domain: sermon preparation, pastoral care, giving analytics, volunteer management, small group coordination, legal compliance, and more.

### The Unified Congregant Record

At the center of the platform is the Unified Congregant Record (UCR)—an 871-line Prisma schema with 20+ models and 30+ enums that captures every dimension of a person’s relationship with their church: identity, spiritual journey milestones, care interactions, prayer history, giving patterns, volunteer engagement, group participation, and attendance. The UCR is the gravitational data moat: every product reads from and writes to this shared schema through the `@pastoral-tech/core` service layer. Direct database access from product code is classified as a CRITICAL drift violation.

### Theological Guardrails

A domain-specific constraint distinguishes Pastoral Tech from generic SaaS platforms: theological integrity. Every AI-generated output must respect the tenant’s denominational mode (Reformed, Wesleyan, Catholic, Pentecostal, Baptist, Lutheran, Anglican, or Nondenominational). The platform’s first principle is “refuse rather than hallucinate”—if the system cannot ground its output in verified doctrine for the target denomination, it produces nothing.

These guardrails operate at the platform level via the `GuardrailConfig` model, which specifies per-tenant: denomination mode, custom constraints, blocked topics, required phrasing, and preferred Scripture version. The Sentinel agent verifies guardrail compliance as part of its 12-point drift detection protocol.

### Product Interconnection Map

The following cross-product data flows are implemented at Layer 1 of the helix:

- **Steward IQ** → **Shepherd HQ**: Financial distress detection triggers pastoral care alert
- **Pulpit HQ** → **Mantle AI**: Sermon content feeds discipleship pathways
- **Equip AI** → **Shield AI**: Volunteer assignment requires background check clearance
- **Gather AI** → **Mantle AI**: Small group participation feeds discipleship tracking
- **Equip AI** → **Steward IQ**: Volunteer-giving correlation analysis
- **Pulpit HQ** → **Prayer SP**: Sermon themes generate responsive prayer prompts

## HPA Application Results

Applying the Helical Pipeline Architecture to Pastoral Tech yields:

Metric	Value
Total products	20+
Total tasks (all layers)	200+
Phases	4 (Bootstrap, Foundation, Tier 1, Tier 2+)
Helix layers	4 (Genesis, Integration, Intelligence, Saturation)
Max concurrent agents	6
Steady-state pipeline width	$\min(20, 4 \times 4) = 16$ work units
Drift checks per checkpoint	12
Denomination modes supported	8
UCR models	20+

Table 2: Pastoral Tech HPA deployment metrics.

## Discussion

### Advantages

**Sustained Parallelism:** Unlike sequential phase gates, HPA keeps all four stage teams active at all times once the pipeline fills. The theoretical pipeline width of  $4(L+1)$  concurrent work units far exceeds sequential models.

**Graceful Degradation:** If a product stalls at any stage, only that product is affected. Other products continue flowing through the pipeline. The Sentinel continuously monitors for blocked products and can trigger re-entry at the failed stage.

**Bounded Agent Context:** By decomposing work into stages rather than assigning agents to entire product lifecycles, each agent’s task scope—and thus context usage—is bounded. The gradient compaction protocol provides a safety net when even bounded tasks exceed expectations.

**Natural Integration Ordering:** The layer structure naturally orders integration work: products are individually functional (Layer 0) before they integrate with each other (Layer 1), and integrations are verified before AI features rely on them (Layer 2). This avoids the integration nightmare of building everything simultaneously.

### Limitations

**Pipeline Fill Time:** The helix does not reach steady-state parallelism instantly. The first product must traverse Forge → Define → Build → Verify before the pipeline is “warm.” With four stages and conservative stage durations, the fill time is approximately  $4\times$  the longest single-stage duration.

**Coordination Overhead:** The Orchestrator’s polling loop, compaction handling, and agent life-cycle management consume resources. For very small platforms (fewer than 4 products), the overhead may exceed the parallelism benefit.

**Layer Semantics Require Domain Expertise:** The work performed at each layer must be carefully specified for each domain. The layer taxonomy described here is specific to compound SaaS; other domains (e.g., embedded systems, scientific computing) would require different layer definitions.

## Future Work

Several extensions are natural:

- **Distributed Execution:** Running stage teams across multiple machines, with the Orchestrator coordinating via a message broker rather than local tmux sessions.
- **Multi-Model Agent Teams:** Assigning different LLM models to different roles (e.g., a fast model for Forge scaffolding, a reasoning model for Define specification, a code-optimized model for Build).
- **Adaptive Layer Count:** Dynamically determining the number of helix layers based on product complexity metrics rather than a fixed count.
- **Cross-Helix Synchronization:** Running multiple independent helices for different product families that synchronize at integration layers.

## Conclusion

---

The Helical Pipeline Architecture provides a principled execution model for autonomous AI agent teams building compound SaaS platforms. By organizing work as a helix—where four concurrent stages repeat at compounding layers of integration maturity—HPA achieves sustained parallelism, bounded agent context, and natural integration ordering that flat pipelines and sequential phase gates cannot match.

The formal model (state space, transition function, invariants) provides guarantees: helical monotonicity ensures products never regress, the layer gate prevents premature integration, and the concurrency bound prevents resource exhaustion. The gradient auto-compaction protocol addresses the practical constraint of finite context windows without requiring human intervention.

Applied to Pastoral Tech’s 20+ product ministry platform, HPA coordinates 200+ tasks across 4 layers with theological guardrail enforcement, demonstrating that the architecture scales to real-world compound startup development. The model generalizes beyond church operations to any multi-product platform where products share infrastructure and must integrate incrementally.

The helix never returns to its starting position. Each revolution elevates. That is the architecture.

*Soli Deo Gloria.*

## References

---

- [1] J. Smart, *Jenkins: The Definitive Guide*, O’Reilly Media, 2011.

- [2] GitLab Inc., “GitLab CI/CD Documentation,” 2024. [Online]. Available: <https://docs.gitlab.com/ee/ci/>
- [3] Q. Wu, G. Banber, et al., “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation,” *arXiv:2308.08155*, 2023.
- [4] J. Moura, “CrewAI: Framework for Orchestrating Role-Playing Autonomous AI Agents,” 2024. [Online]. Available: <https://github.com/joaomdmoura/crewAI>
- [5] G. Li, H. Hammoud, et al., “CAMEL: Communicative Agents for ‘Mind’ Exploration of Large Language Model Society,” *arXiv:2303.17760*, 2023.
- [6] P. Lewis, E. Perez, et al., “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” *NeurIPS*, 2020.
- [7] J. Wu, X. Ouyang, et al., “Recursively Summarizing Books with Human Feedback,” *arXiv:2109.10862*, 2021.
- [8] Z. Packer, “MemGPT: Towards LLMs as Operating Systems,” *arXiv:2310.08560*, 2023.
- [9] A. Agarwal, “The Compound Startup,” 2024. [Online]. Available: <https://www.rippling.com/blog/the-compound-startup>
- [10] Y.-K. Kwok and I. Ahmad, “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors,” *ACM Computing Surveys*, vol. 31, no. 4, pp. 406–471, 1999.