

# Convergence by Composition

A Structured Adapter Architecture for Multi-Agent System Integration

Preprint – Feedback Welcome

Bogdan Banu  
bogdan@banu.be

March 30, 2026

## Abstract

Agent orchestration frameworks—Swarms, DeerFlow, AnimaWorks, Ralph, and A-Evolve—make fundamentally different architectural choices (graph-based, event-driven, hierarchical, evolutionary) yet share structural concerns: error amplification across handoffs, coordination overhead from tool proliferation, and the absence of convergence guarantees for adaptive loops. We present a structured adapter architecture centred on a single intermediate representation, **ExternalTopology**, that lets Operon’s structural analysis layer analyze the static structural skeleton of external agent configurations without importing the target framework. Five adapters parse framework-specific configurations into this common representation; four epistemic bounds serve as a structural linter; four TLA+ specifications verify safety invariants; and a co-design fixed-point proof (following Zardini) establishes that the adaptive assembly loop converges within  $\varepsilon$ . The architecture is implemented in 14 convergence modules with 216 convergence-specific tests as part of a broader suite of 1,404 tests and 98 worked examples.

## 1 Introduction

The past two years have seen a proliferation of agent orchestration frameworks, each embodying a different theory of how autonomous agents should coordinate. Swarms [5] provides graph-based workflows with sequential, concurrent, and hierarchical scheduling. DeerFlow, built on LangGraph, offers sub-agent harnesses with sandboxed execution and progressive skill loading. AnimaWorks [1] introduces developmental priming with heartbeat-driven memory consolidation. Ralph organizes agents as “hats” connected by event-driven transitions with backpressure constraints. A-Evolve implements an evolutionary Solve–Observe–Evolve–Gate–Reload loop that mutates workspace snapshots under a fitness gate.

These frameworks differ in execution model, memory architecture, and deployment infrastructure, yet they converge on a shared set of structural concerns. Every multi-agent system must manage error amplification across sequential handoffs, balance coordination overhead against parallel speedup, and decide when an adaptive loop has stabilized. As Evans, Bratton, and Agüera y Arcas argue, intelligence is “fundamentally plural, social, and relational” [3]; the engineering challenge is to build, verify, and optimize the structural infrastructure for such systems.

The Agentic Computation Graph (ACG) survey by Yue et al. [6] provides a comprehensive taxonomy of workflow optimization methods—from static template search to in-execution editing—but does not address formal verification or compositional convergence proofs. Kim et al. [4] identify scaling laws for agent systems whose functional form is consistent with the epistemic bounds derived in the present work.

Our contribution is a *structured adapter architecture* that decouples structural analysis from runtime execution:

1. **ExternalTopology** as a framework-agnostic intermediate representation: five adapters parse five frameworks into a single frozen dataclass; all downstream analysis is source-agnostic.
2. **Epistemic bounds as structural linter**: the adapter risk score combines error amplification, sequential penalty, and tool density bounds with a topology-mismatch flag; parallel acceleration is available from the epistemic layer but not included in the adapter’s composite score.
3. **TLA+ verified safety invariants**: four specifications cover template exchange, developmental gating, convergence detection, and evolution gating, verified by the TLC model checker with small-model parameters.
4. **Co-design convergence**: each adapter is a design problem in Zardini’s co-design theory [7]; series/parallel composition and feedback fixed-point iteration prove that scoring stabilizes.

The rest of this paper describes the architecture (Section 2), formal results (Section 3), implementation (Section 4), related work (Section 5), discussion (Section 6), and conclusion (Section 7).

## 2 Architecture

The convergence architecture is a five-layer stack (Figure 1). Each layer is independently useful: the cognitive layer can run without an orchestration layer; the structural layer can analyze topologies without executing them.

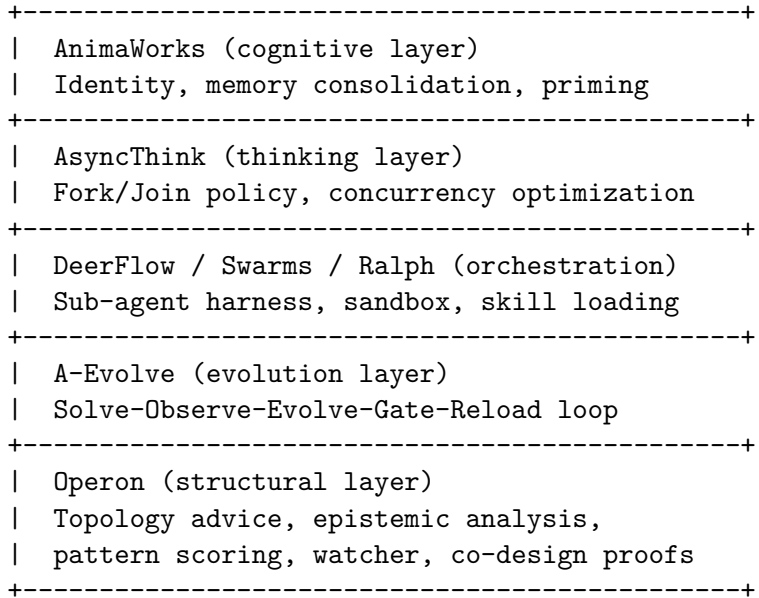


Figure 1: Five-layer convergence stack. Operon sits at the bottom, providing structural analysis to every layer above. Arrows flow upward (analysis results) and downward (topology descriptions).

## 2.1 ExternalTopology as Framework-Agnostic Intermediate Representation

The key abstraction is a single frozen dataclass that every adapter produces and every analysis function consumes:

Listing 1: ExternalTopology dataclass (convergence/types.py).

```
@dataclass(frozen=True)
class ExternalTopology:
    source: str # "swarms"|"deerflow"|...
    pattern_name: str
    agents: tuple[dict[str, Any], ...]
    edges: tuple[tuple[str, str], ...]
    metadata: dict[str, Any] = field(default_factory=dict)
```

The agent specs use `dict[str, Any]` to accommodate heterogeneous framework configurations. Python type hints annotate the top-level fields; the nested dicts carry no schema enforcement at runtime. The `@dataclass(frozen=True)` decorator prevents reassignment of top-level fields but does not deep-freeze nested mutable objects. The `source` field is a tag, not a type discriminant—all downstream code treats every `ExternalTopology` identically. Agents are plain dicts with at least `name` and `role` keys; edges are directed pairs. This representation is serializable as JSON, enabling analysis results to be stored, transmitted, and compared across framework boundaries.

## 2.2 Analysis Pipeline

The function `analyze_external_topology(topology)` takes any `ExternalTopology` and returns an `AdapterResult`:

Listing 2: AdapterResult dataclass (convergence/types.py).

```
@dataclass(frozen=True)
class AdapterResult:
    topology_advice: TopologyAdvice
    suggested_template: PatternTemplate | None
    warnings: tuple[str, ...]
    risk_score: float # [0.0, 1.0]
```

The pipeline constructs a wiring diagram from agents and edges, applies three epistemic bounds (error amplification, sequential penalty, tool density) plus a topology-mismatch flag, generates structural warnings, and optionally produces a `PatternTemplate` for the pattern library.

## 2.3 Design Principles

Three principles govern the adapter architecture:

1. **Serializable dicts, zero external imports.** Every adapter operates on plain dicts parsed from configuration files or API responses. No adapter imports `Swarms`, `DeerFlow`, `AnimaWorks`, `Ralph`, or `A-Evolve`.
2. **One parser, one analyzer.** Adding a new framework means writing one `parse_*` function that produces `ExternalTopology`; the analysis code is unchanged.
3. **Bidirectional exchange.** `DeerFlow`’s Markdown skill files convert to `PatternTemplate` and vice versa, enabling cross-framework template sharing.

### 3 Formal Results

#### 3.1 Epistemic Theorems as Structural Linter

Operon derives four structural bounds from epistemic topology—the study of how knowledge and uncertainty propagate through agent communication graphs. These bounds are applied to every `ExternalTopology` as a “structural linter,” flagging architectures that are likely to amplify errors or incur excessive coordination overhead.

**Definition 1** (Error Amplification Bound). *For a sequential chain of  $n$  agents, each with per-stage error rate  $\epsilon$ , the cumulative error probability is bounded by:*

$$P_{\text{error}}(n) \leq 1 - (1 - \epsilon)^n$$

*When  $n\epsilon \ll 1$ , the linear approximation  $P_{\text{error}} \approx n\epsilon$  applies. The linter warns when the chain depth produces  $P_{\text{error}} > \tau$  for a configurable tolerance  $\tau$ .*

**Definition 2** (Sequential Penalty). *For a topology with  $n_{\text{seq}}$  sequential handoffs and total agent count  $N$ , the sequential penalty ratio is:*

$$\rho_{\text{seq}} = \frac{n_{\text{seq}}}{N}$$

*Values of  $\rho_{\text{seq}} > 0.7$  indicate that the topology is predominantly sequential and may benefit from parallelization.*

**Definition 3** (Parallel Acceleration). *For a topology with  $k$  independent branches of depths  $d_1, \dots, d_k$ , the critical-path latency under parallel execution is:*

$$L_{\text{par}} = \max_{i \in [k]} d_i$$

*The speedup over sequential execution is  $S = \sum_i d_i / L_{\text{par}}$ . The linter reports the achievable speedup and flags topologies where  $S < 1.2$  (parallel structure exists but provides negligible benefit).*

The full epistemic analysis layer computes all four metrics. The `analyze_external_topology()` function exposes three of these (error amplification, sequential penalty, tool density) plus a topology-mismatch flag in the composite risk score. Parallel acceleration is computed and reported separately but not included in the risk score.

**Definition 4** (Tool Density). *For a topology with  $T$  distinct tools across  $N$  agents, the tool density is  $\delta = T/N$ . High density ( $\delta > 3$ ) suggests coordination overhead: each agent must manage many tool interfaces. Low density ( $\delta < 0.5$ ) suggests under-utilization.*

The composite risk score is a weighted sum of four components:

$$r = 0.4 \hat{P}_{\text{error}} + 0.3 \rho_{\text{seq}} + 0.2 \delta_{\text{density}} + 0.1 M_{\text{topo}}$$

where  $\hat{P}_{\text{error}}$  is the normalized centralized error bound,  $\rho_{\text{seq}}$  is the effective sequential overhead,  $\delta_{\text{density}}$  is the normalized planning-cost ratio (tool density), and  $M_{\text{topo}} \in \{0, 1\}$  is a binary topology-mismatch flag raised when the detected topology class diverges from the recommended class. The weights are  $w_1 = 0.4$ ,  $w_2 = 0.3$ ,  $w_3 = 0.2$ ,  $w_4 = 0.1$ .

These bounds are consistent with the scaling laws reported by Kim et al. [4]: their empirical observation that error rates grow with chain depth and that coordination overhead dominates at high agent counts corresponds to the Error Amplification and Tool Density bounds respectively.

### 3.2 TLA+ Verified Safety Invariants

Four TLA+ specifications model critical protocols in the convergence architecture. Each is verified by the TLC model checker with small-model parameters (2–3 agents, 2–3 templates/tools, bounded iteration counts). TLC model configurations (`.cfg` files) with small-model parameters and a step-by-step reproduction guide are provided in the `specs/` directory alongside the specifications. Table 1 summarizes the specifications and their key invariants.

Table 1: TLA+ specifications and verified safety properties.

Specification	Key Safety Invariants	Actions
TemplateExchangeProtocol	Adoption respects <code>min_stage</code> Trust changes only via <code>RecordOutcome</code> No self-adoption	Export, Import, RecordOutcome, StageAdvance
DevelopmentalGating	Critical periods never reopen Stages monotonically advance Tools respect capability gates	Tick, AcquireTool, OpenPeriod, ClosePeriod, Scaffold
ConvergenceDetection	Halt is terminal Intervention rate triggers halt	StageResult (with nondeterministic intervention)
EvolutionGating	Score monotonically non-decreasing Workspace advances only via Gate Version bounded by <code>MAX_VERSIONS</code>	Evolve, Gate, Rollback

**TemplateExchangeProtocol.** Models cross-agent template sharing with trust-weighted adoption. An agent imports a peer’s template only when three guards are satisfied: (i) the agent’s developmental stage meets the template’s minimum stage, (ii) trust in the peer exceeds `MIN_TRUST`, and (iii) the effective score  $\text{peer\_success\_rate} \times \text{trust} \geq \text{ADOPTION\_THRESHOLD}$ . Trust updates use exponential moving average (EMA) smoothing, applied exclusively in the `RecordOutcome` action—all other actions leave trust unchanged, a structural encoding of the invariant.

**DevelopmentalGating.** Models lifecycle progression with critical periods and capability gating. Agents progress through lifecycle stages, advancing through four stages (Embryonic  $\rightarrow$  Juvenile  $\rightarrow$  Adolescent  $\rightarrow$  Mature). The key safety property is *critical period irreversibility*: once a period transitions from “open” to “closed,” it can never reopen. This models time-limited configuration windows that close permanently.

**ConvergenceDetection.** Models the watcher’s convergence protocol. As agents execute stages, the watcher tracks the ratio of interventions (RETRY, ESCALATE, HALT) to completed stages. When this ratio exceeds `MAX_RATE`, the agent is halted in the same atomic step—a safety property expressing that non-convergent execution is always detected and terminated.

**EvolutionGating.** Models the A-Evolve mutation loop. The `Gate` action accepts a pending mutation only when its benchmark score meets or exceeds the current score plus `MIN_IMPROVEMENT`. This ensures monotonic score progression: the agent system never regresses. The specification intentionally provides no liveness guarantee, since evolution liveness depends on the score generator—an external assumption outside the structural model.

### 3.3 Co-Design Convergence

We apply a simplified version of Zardini’s co-design framework [7], using ordinary function composition rather than the full categorical apparatus. We model each adapter as a *design problem* (DP)—a monotone map from a resource poset to a functionality poset. Monotonicity means that more resources yield at least as many functionalities.

**Definition 5** (Design Problem). *A design problem is a pair  $(f, g)$  where  $f : \mathcal{R} \rightarrow \mathcal{F}$  maps resources to functionalities and  $g : \mathcal{R} \rightarrow \{0, 1\}$  is a feasibility predicate. The map  $f$  is monotone:  $r_1 \leq r_2 \implies f(r_1) \leq f(r_2)$ .*

**Definition 6** (Series Composition). *Given design problems  $DP_1$  and  $DP_2$ , their series composition  $DP_1 \rightarrow DP_2$  is defined by  $f_{1 \rightarrow 2}(r) = f_2(f_1(r))$ , with feasibility  $g_{1 \rightarrow 2}(r) = g_1(r) \wedge g_2(f_1(r))$ .*

**Definition 7** (Parallel Composition). *The parallel composition  $DP_1 \parallel DP_2$  is defined by  $f_{1 \parallel 2}(r) = f_1(r) \cup f_2(r)$ , with feasibility  $g_{1 \parallel 2}(r) = g_1(r) \wedge g_2(r)$ .*

The full adapter stack is a series composition: parsing  $\rightarrow$  analysis  $\rightarrow$  template generation  $\rightarrow$  scoring. The adaptive assembly loop (run  $\rightarrow$  record  $\rightarrow$  score  $\rightarrow$  select) is feedback composition, where the output of the scoring step feeds back as input to the next iteration’s selection.

**Theorem 1** (Adaptive Loop Approximate Convergence). *Let  $DP$  be the feedback composition of the adapter stack, with scoring function  $s : \mathcal{F} \rightarrow [0, 1]$  and feedback map  $\phi(r, f) = r \cup \{s(f)\}$ . If  $s$  is monotone, then the sequence  $s_0, s_1, s_2, \dots$  is monotone nondecreasing and bounded above by 1. For any  $\varepsilon > 0$ , the implementation terminates when  $|s_{n+1} - s_n| < \varepsilon$  or after a configurable maximum number of iterations.*

*Proof.* The scoring function  $s$  maps functionalities to  $[0, 1]$ . The feedback map  $\phi$  is monotone by composition of monotone maps. Therefore the sequence  $s_0 \leq s_1 \leq s_2 \leq \dots$  is monotone nondecreasing and bounded above by 1, so it converges to  $\sup_n s_n = s^*$  by the monotone convergence theorem. The implementation’s `feedback_fixed_point()` has two termination modes: (1) when  $|s_{n+1} - s_n| < \varepsilon$  (default 0.01) for the monitored scoring key, the implementation reports `converged=True`, indicating the tracked value has stabilized within  $\varepsilon$ ; (2) when the iteration count reaches the safety bound (default 100) without satisfying the  $\varepsilon$  criterion, the algorithm halts with `converged=False` and returns the last computed state. The mathematical guarantee (monotone convergence) ensures the sequence approaches  $s^*$ ; the implementation provides a practical decision procedure that reports whether that convergence was achieved within the allocated budget.  $\square$

The implementation in `convergence/codesign.py` provides `compose_series`, `compose_parallel`, and `feedback_fixed_point` with the exact convergence semantics described above.

## 4 Implementation

The convergence architecture is implemented in 14 Python modules under `operon_ai/convergence/`, tested by 216 convergence-specific unit tests as part of a broader suite of 1,404 tests across the entire Operon codebase. Table 2 summarizes the five adapters.

### 4.1 Bidirectional DeerFlow Skill Bridge

DeerFlow’s Markdown-based skill system is structurally similar to Operon’s `PatternTemplate`: both represent reusable workflow patterns with metadata and staged execution steps. The module

Table 2: Convergence adapters: source frameworks and key functions.

Adapter	Source	Parse Function	Template Function
Swarms	Graph workflows	<code>parse_swarm_topology</code>	<code>swarm_to_template</code>
DeerFlow	LangGraph sessions	<code>parse_deerflow_session</code>	<code>deerflow_to_template</code>
AnimaWorks	Supervisor configs	<code>parse_animaworks_org</code>	<code>animaworks_to_template</code>
Ralph	Hat orchestration	<code>parse_ralph_config</code>	<code>ralph_to_template</code>
A-Evolve	Workspace manifests	<code>parse_aevolve_workspace</code>	<code>aevolve_to_template</code>

`convergence/deerflow_skills.py` provides bidirectional conversion: `skill_to_template` parses DeerFlow Markdown skills (extracting frontmatter, workflow steps, and tool references) into scored `PatternTemplate` instances, while `template_to_skill` exports Operon templates as DeerFlow-compatible Markdown. This enables cross-framework template exchange: Operon can import DeerFlow’s community-contributed skill library, and DeerFlow can consume Operon’s structurally validated templates.

## 4.2 Hybrid Assembly

The `hybrid_skill_organism` function implements a two-phase assembly strategy. In the first phase, it queries the pattern library for templates matching the task fingerprint; if a template with a sufficiently high success score exists, it is used directly. In the second phase (fallback), an LLM generator produces a one-shot agent configuration, which is registered in the library for scoring refinement over subsequent runs. This library-first, generator-fallback design ensures that the system improves with experience while maintaining availability for novel tasks.

## 4.3 Cognitive Extensions

Three modules extend the cognitive layer:

- **PrimingView** (in `patterns/priming.py`): a subclass of `SubstrateView` that adds recent outputs, telemetry, experience records, and developmental status as multi-channel context.
- **HeartbeatDaemon** (in `patterns/heartbeat.py`): periodic consolidation of short-term observations into long-term memory, inspired by AnimaWorks’ heartbeat protocol.
- **AsyncOrganizer** (in `convergence/async_thinking.py`): Fork/Join execution within individual stages, inspired by the AsyncThink [2] observation that intra-stage parallel reasoning can reduce latency without increasing token cost.

## 4.4 Catalog Seeding

The `convergence/catalog.py` module seeds the pattern library from multiple sources: `seed_library_from_swarm` registers built-in Swarms workflow patterns; `seed_library_from_deerflow` imports DeerFlow skills; `seed_library_from_ralph` converts Ralph hat configurations; and `seed_library_from_acg_survey` registers templates from the ACG survey’s comparison cards [6], annotated with graph determination time and plasticity mode metadata. The combined catalog provides a rich initial library for hybrid assembly.

## 5 Related Work

**Agent workflow taxonomy.** The ACG survey by Yue et al. [6] provides a comprehensive taxonomy of workflow optimization methods, classifying approaches along two dimensions: graph determination time (offline, pre-execution, in-execution) and graph plasticity mode (static, dynamic). Our work operates within this taxonomy: the adapter architecture is a pre-execution analysis layer, while the watcher provides in-execution monitoring. The ACG framework treats workflow graphs as optimizable structures; we add formal verification and convergence guarantees that the survey identifies as open problems.

**Plural intelligence.** Evans et al. [3] argue that the next advance in AI will be combinatorial societies of specialized agents rather than monolithic models, drawing on primate social scaling and cultural ratchet effects. Our work provides the formal infrastructure—structured composition, verified safety invariants, convergence proofs—for the “plural intelligence” vision they articulate.

**Scaling laws.** Kim et al. [4] derive empirical scaling laws for multi-agent systems, finding that error rates grow with chain depth and coordination overhead dominates at high agent counts. These findings are consistent with our epistemic bounds: the Error Amplification Bound predicts the chain-depth dependence; the Tool Density bound predicts the coordination overhead curve.

**Intra-stage reasoning.** Chi et al.’s AsyncThink [2] demonstrates that allowing agents to reason asynchronously within a stage—forking parallel thought branches and joining at a synchronization point—reduces latency without increasing token cost. Our `AsyncOrganizer` formalizes this pattern with configurable concurrency ratios and critical-path scheduling.

**Co-design theory.** Zardini [7] develops a general theory of co-design based on monotone maps between resource and functionality posets, with series, parallel, and feedback composition operators. We apply this framework to model each adapter as a design problem and prove that the adaptive assembly loop converges to a fixed point.

## 6 Discussion and Future Work

**From analysis to compilation.** The current adapter architecture analyzes structure but does not generate executable workflows. The natural next step is *compilers*: functions like `organism_to_swarms` and `organism_to_deerflow` that compile Operon agent systems into Swarms workflow configurations or DeerFlow LangGraph sessions. For DeerFlow, the deepest integration is a native LangGraph watcher node (`operon_watcher_node`) that provides real-time structural monitoring inside DeerFlow’s execution loop. A Ralph compiler (`organism_to_ralph`) would emit hat configurations with backpressure constraints derived from the epistemic analysis.

**Cross-target evaluation.** A systematic evaluation harness running identical tasks across Operon, Swarms, DeerFlow, AnimaWorks, and Ralph would provide empirical validation of the structural predictions. The ACG survey’s recommendation to measure structural variation across inputs is a valuable addition to such an evaluation protocol, alongside standard metrics (success rate, token cost, latency, intervention count, convergence rate).



**Prompt optimization.** The adapter architecture optimizes topology but leaves node-level content (prompts, demonstrations) fixed. DSPy-style prompt optimization integrated via the A-Evolve mutation loop could address this gap: the evolutionary Solve–Observe–Evolve–Gate–Reload cycle naturally accommodates prompt mutations as a special case of workspace evolution, with the fitness gate ensuring monotonic improvement.

**Limitations.** The adapters analyze structure, not runtime behavior. An architecture that scores well on all four epistemic bounds may still fail due to prompt quality, model capability, or environmental non-stationarity. The TLA+ specifications are verified with small-model parameters; scaling to production-size state spaces would require compositional verification techniques or abstraction refinement. The co-design convergence proof assumes monotonicity of the scoring function, which may not hold under distribution shift in the task stream.

## 7 Conclusion

We have presented a structured adapter architecture for integrating Operon’s structural analysis layer with five external agent orchestration systems—Swarms, DeerFlow, AnimaWorks, Ralph, and A-Evolve—through a single intermediate type, `ExternalTopology`. The architecture contributes four epistemic bounds that serve as a structural linter, four TLA+-verified safety invariants, and a co-design convergence proof showing that the adaptive assembly loop stabilizes within  $\epsilon$ . By operating on serializable dicts with zero external imports, the adapter architecture achieves framework-agnostic analysis while preserving a structured interchange format with top-level frozen dataclass fields (nested dicts remain mutable by Python convention). The 14 convergence modules, 216 convergence tests, 4 TLA+ specifications, and 98 worked examples provide a concrete, verifiable foundation for multi-agent system integration.

## References

- [1] AnimaWorks Contributors. Animaworks: Autonomous agent runtime with priming and heart-beat consolidation. <https://github.com/AnimaWorks/AnimaWorks>, 2025. Operational runtime for autonomous agents with developmental priming.
- [2] Yihang Chi, Zhuoer Lin, and Xiang Yue. AsyncThink: Asynchronous thinking for language model agents. *arXiv preprint*, 2025. Intra-stage parallel reasoning with Fork/Join execution.
- [3] Richard Evans, Benjamin H. Bratton, and Blaise Agüera y Arcas. Agentic AI and the next intelligence explosion. *arXiv preprint arXiv:2603.20639*, 2026. Position paper arguing intelligence is fundamentally plural and social.
- [4] Yubin Kim, Ken Gu, Chanwoo Park, Chunjong Park, Samuel Schmidgall, A. Ali Heydari, Yao Yan, Zhihan Zhang, Yuchen Zhuang, Yun Liu, Mark Malhotra, Paul Pu Liang, Hae Won Park, Yuzhe Yang, Xuhai Xu, Yilun Du, Shwetak Patel, Tim Althoff, Daniel McDuff, and Xin Liu. Towards a science of scaling agent systems. *arXiv preprint arXiv:2512.08296*, 2025. Google Research, Google DeepMind, and MIT.
- [5] Kye Gomez Osagie. Swarms: The enterprise-grade production-ready multi-agent orchestration framework. <https://github.com/kyegomez/swarms>, 2024. Multi-agent orchestration with sequential, concurrent, and graph workflows.
- [6] Xiang Yue, Anil Bhandari, Youngwon Ko, Rishi Patel, Zhuoer Lin, Shuaichen Zhou, Jiacheng Gao, Wenhui Chen, and Xinyu Pan. From static templates to dynamic runtime graphs: A survey of workflow optimization for LLM agents. *arXiv preprint arXiv:2603.22386*, 2026. Comprehensive taxonomy of agent workflow optimization methods.
- [7] Gioele Zardini. *Co-Design of Complex Systems: From Compositionality to Monotone Theory*. PhD thesis, ETH Zurich, 2023. Monotone co-design theory for compositional system design.