HIVEMIND: THE ORCHESTRA OF INTELLIGENCE

TECHNICAL REPORT

Yutai Long
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
yutailon@andrew.cmu.edu

Unmesh Chakravarty
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
ukc@andrew.cmu.edu

August 19, 2026

ABSTRACT

Hivemind is an **Agentic Execution System (AES)** designed around a fundamentally different computational assumption than classical AI systems: **token generation can be distributed**, and **effective compute scales with task decomposition** rather than model size. Instead of relying on a single model to perform end-to-end reasoning, Hivemind represents a task as a **directed acyclic graph (DAG)** of **kernels**, each executed independently across a dynamically managed pool of models.

While prior work has drawn analogies between agent systems and compiler pipelines, this work reframes Hivemind as a **hybrid of compilation and parallel execution**. The compiler perspective provides structure—task decomposition as parsing, DAGs as intermediate representations, and optimization passes over execution graphs—while the parallel perspective defines the true objective: **minimizing execution time through concurrency**.

Unlike classical systems, Hivemind operates under **effectively unbounded parallelism**, **stochastic execution units**, and a cost model dominated not by compute but by **coordination overhead**. The system is therefore governed by three coupled objectives: **latency**, **cost**, and **accuracy**. Latency is reduced through parallel execution and careful **granularity selection**; cost is minimized by enabling **smaller, specialized models to collaborate** instead of relying on a single large model; and accuracy is improved through a **tool-based verification system** that validates intermediate and final outputs.

This introduces a fundamental tradeoff. Fine-grained decomposition increases parallelism and reduces **critical path latency**, but incurs overhead in DAG construction, dependency management, and verification. Coarse-grained execution minimizes overhead but limits achievable speedup and reduces opportunities for cost and accuracy optimization.

Hivemind explicitly models this tradeoff and introduces a system composed of three components—**Canvas**, **Hive**, and **Swarm**—that together optimize decomposition, execution, and coordination. The result is a computational framework in which complex tasks are treated as **schedulable programs**, shifting the bottleneck of AI systems from inference to **structured execution**.

1 Introduction

Modern AI systems are bottlenecked not by model capability, but by **execution structure**. Large language models are typically used in a sequential, monolithic manner: a single prompt produces a single output through an autoregressive process. While effective for small tasks, this paradigm fails to scale for complex workloads, where reasoning, retrieval, transformation, and verification must be composed across multiple interdependent steps.

Hivemind introduces a different paradigm: an **Agentic Execution System (AES)**. Instead of treating a task as a single inference, Hivemind treats it as a **program**. A user request is transformed into a **directed acyclic graph (DAG)** of **kernels**, where each kernel represents a minimal unit of computation. These kernels are executed independently across a pool of models, and their outputs are composed through explicit data dependencies.

This reframing shifts the fundamental computational model. In traditional systems, compute is the limiting resource: CPUs and GPUs have finite parallelism, and performance is constrained by hardware throughput. In an AES, execution units can be instantiated dynamically, making **parallelism effectively unbounded**. As a result, execution time is no longer determined by total work, but by the **critical path** of the task graph.

However, this shift introduces a new bottleneck: **coordination overhead**. Decomposing tasks, managing dependencies, formatting prompts, and verifying outputs all incur cost. Unlike classical systems where overhead is negligible relative to compute, in an AES this overhead can dominate execution if not carefully controlled.

This leads to the central problem of Hivemind: **granularity optimization**. The system must determine how to partition a task into kernels such that parallelism is maximized while overhead remains bounded. Fine-grained decomposition reduces **critical path latency** but increases coordination cost. Coarse-grained execution minimizes overhead but limits parallel speedup and restricts optimization opportunities.

Beyond latency, Hivemind optimizes two additional dimensions: **cost** and **accuracy**. Cost is reduced by enabling **smaller, specialized models to collaborate**, rather than relying on a single large model for end-to-end reasoning. Accuracy is improved through a **tool-based verification system**, which validates intermediate and final outputs and enables corrective re-execution when necessary. These three objectives—**latency**, **cost**, and **accuracy**—are tightly coupled, and must be optimized jointly.

To address this, Hivemind is structured into three core components:

- **Canvas**: the execution substrate, where tasks are represented as DAGs of kernels and optimized through decomposition and verification.
- **Hive**: a **hierarchical interface layer** that maps real-world and digital systems into the DAG abstraction, enabling recursive composition and shared context.
- **Swarm**: the runtime system responsible for **model allocation**, **kernel scheduling**, and **latency hiding** across the execution graph.

While Hivemind draws inspiration from compiler theory—particularly in its use of intermediate representations and optimization passes—it departs in critical ways. Execution units are **stochastic**, resources are **elastic**, and the primary optimization target is **execution time under overhead**, rather than resource utilization. Similarly, while parallels exist with parallel computing and operating systems, the assumption of effectively unbounded parallelism fundamentally alters the design space.

Hivemind should therefore not be understood as an incremental improvement to existing agent frameworks, but as a new computational model for AI systems. By treating tasks as **schedulable programs** and execution as a structured, parallel process, it shifts the bottleneck of AI from inference to **structured execution**, enabling scalable, real-time solutions to complex workloads.

2 Foundations of Agentic Systems

Hivemind is informed by three major areas of systems research: compilation, parallel execution, and operating systems. Each provides a lens for understanding how complex workloads can be structured and executed. However, none of these paradigms directly apply without modification. Hivemind operates under fundamentally different assumptions: **execution units are elastic**, **compute is effectively unbounded**, and **cost is dominated by coordination overhead rather than computation**.

2.1 From Sequential Reasoning to Structured Execution

Most existing approaches to complex reasoning in language models rely on **sequential decomposition**. Techniques such as chain-of-thought prompting [Wei et al., 2022] and more advanced frameworks like [Yao et al., 2022] interleave reasoning steps with tool usage, allowing models to iteratively refine their outputs and interact with external systems.

While effective, these approaches remain fundamentally **linear**. Each step depends on the previous one, forming a chain rather than a graph. This limits scalability, as execution time grows with the number of reasoning steps.

Hivemind generalizes this paradigm. Instead of a sequence of reasoning and acting steps, it constructs a **directed acyclic graph (DAG)** of **kernels**, where reasoning, acting, and verification are represented as independent nodes with explicit dependencies. This allows multiple reasoning paths to proceed in parallel, breaking the sequential bottleneck inherent in prior approaches.

2.2 Compiler-Theoretic Perspective

Compiler theory provides a natural abstraction for structured transformation. A compiler transforms a high-level specification into an executable form through parsing, intermediate representation (IR) construction, optimization, and code generation [Aho et al., 2006, Cooper and Torczon, 2012].

Hivemind mirrors this structure at a high level. A natural language task is transformed into a DAG, which serves as an intermediate representation. Optimization passes operate on this graph, including dependency resolution, redundancy elimination, and execution planning.

This correspondence provides a useful conceptual framework, but the analogy must be applied carefully.

In classical compilers, resources are **finite**. Register allocation, for example, is constrained by a fixed number of physical registers, and spilling is required when demand exceeds supply. This constraint shapes optimization strategies and introduces tradeoffs between memory and compute.

In Hivemind, the situation is fundamentally different. The number of execution units—agents—is **not fixed**. When parallel demand exceeds current capacity, new agents can be instantiated dynamically. As a result, classical resource allocation problems such as register spilling do not arise in the same form.

This shifts the optimization objective. Instead of minimizing resource usage, Hivemind focuses on minimizing **execution time under coordination overhead**. The compiler analogy remains useful for structuring transformations, but the underlying cost model is different.

2.3 Parallel Execution Perspective

Parallel computing provides the closest match to Hivemind's execution model. Tasks are represented as DAGs, where nodes correspond to computation and edges encode dependencies. The goal is to minimize the **makespan**, or total execution time.

In classical parallel systems, scheduling a DAG under resource constraints is known to be computationally hard in the general case [Ullman, 1975], and even simplified variants require careful algorithmic design [Coffman and Graham, 1972]. These systems are fundamentally limited by the number of available processors.

Hivemind operates in a different regime. Because execution units can be scaled dynamically, **parallelism is effectively unbounded**. Execution time is therefore determined primarily by the **critical path length** of the DAG, rather than the total amount of work.

However, this introduces a new limiting factor: **overhead**. Each kernel incurs costs associated with prompt construction, model invocation, dependency tracking, and verification. As the number of kernels increases, these costs accumulate.

This leads to a fundamental tradeoff:

- **Fine-grained decomposition**: reduces critical path length and increases parallelism, but incurs high coordination overhead.
- **Coarse-grained decomposition**: minimizes overhead, but increases critical path length and limits parallel speedup.

This tradeoff defines the central optimization problem of Hivemind and distinguishes it from classical parallel systems, where compute resources are the primary constraint.

2.4 Operating Systems Perspective

Operating systems provide abstractions for process management, scheduling, and resource allocation. In Hivemind, similar responsibilities exist, but under different constraints.

Traditional operating systems manage a fixed set of hardware resources, allocating CPU time and memory across competing processes. Scheduling decisions balance fairness, throughput, and latency.

In Hivemind, the runtime system (Swarm) manages a dynamic pool of model invocations rather than fixed hardware. The primary objective is not fairness, but **latency minimization and coordination efficiency**. Kernels are scheduled based on dependency readiness and optimization objectives, rather than time slicing.

Additionally, Hivemind adopts a **shared-state model** through the Canvas. Instead of isolated processes with private memory, kernels operate over a shared key-value space. Dependencies are enforced through read/write ordering, ensuring that each kernel receives the correct inputs.

This model aligns closely with graph-based intermediate representations such as SSA form [Cytron et al., 1991] and Sea-of-Nodes IR [Click and Paleczny, 1995], where data dependencies are made explicit and enable global optimization.

However, unlike classical systems, execution units in Hivemind are **stochastic**. Intermediate outputs may not satisfy expected constraints, requiring explicit **verification** before results are propagated downstream. This introduces a runtime correctness layer that has no direct analog in traditional operating systems or compilers.

2.5 Synthesis

Taken together, these perspectives define Hivemind as a hybrid system:

- From compilers, it inherits **structured transformation** and optimization over an intermediate representation.
- From parallel computing, it inherits **DAG-based execution** and the focus on critical path minimization.
- From operating systems, it inherits **dynamic scheduling** and runtime coordination.

However, its defining characteristic is the inversion of traditional constraints. **Compute is abundant, but coordination is expensive**. Execution units are **stochastic**, not deterministic. Optimization targets **latency, cost, and accuracy** under overhead, rather than resource utilization.

Because of this, Hivemind does not attempt to replicate any one paradigm exactly. Instead, it selectively borrows principles where they apply, and discards them where they do not. This flexibility is essential to capturing the true nature of **Agentic Execution Systems (AES)**.

3 Canvas: The Execution Substrate

The **Canvas** is the core execution substrate of Hivemind. It defines the representation, transformation, and evaluation of tasks as structured programs. At its core, the Canvas maintains a **directed acyclic graph (DAG)** of **kernels**, where each kernel is a minimal unit of computation operating over a shared state.

The primary objective of the Canvas is to minimize overall execution time while jointly optimizing **latency**, **cost**, and **accuracy**. Unlike classical systems, where compute dominates cost, the Canvas operates in a regime where **coordination overhead is the dominant term**. This section formalizes the computational model and introduces the decomposition and verification mechanisms that govern system performance.

3.1 Kernel and DAG Formalization

Let a task be represented as a DAG $G = (V, E)$, where:

- $V = \{v_1, \ldots, v_n\}$ are **kernels**
- $E \subseteq V \times V$ are directed dependencies

Each kernel v_i is defined as a tuple:

$$v_i = (f_i, I_i, O_i, c_i, \ell_i, p_i)$$

where:

- f_i: function (model + strategy)
- I_i, O_i: input/output keys on Canvas
- c_i: execution cost (USD)
- ℓ_i: latency
- p_i: success probability (accuracy proxy)

Execution time of the DAG is determined by the **critical path**:

$$T(G) = \max_{\pi \in \text{paths}(G)} \sum_{v_i \in \pi} \ell_i$$

Given effectively unbounded parallelism, total work $\sum \ell_i$ is not the bottleneck; only $T(G)$ determines latency.

3.2 Why Structured Execution Outperforms Monolithic Execution

Consider a monolithic execution model (Mono), where a single model processes the entire task.

Let:

$$T_{\text{mono}} = W \cdot \alpha_M \quad \text{and} \quad C_{\text{mono}} = W \cdot \gamma_M$$

where:

- α_M: latency per token for large model
- γ_M: cost per token for large model

In contrast, Hivemind decomposes the task into a DAG:

$$T_{\text{hive}} = \max_{\pi} \sum \ell_i \ll W \cdot \alpha_M$$

due to parallel execution.

Cost becomes:

$$C_{\text{hive}} = \sum c_i \approx \sum W_i \cdot \gamma_i$$

where smaller models satisfy:

$$\gamma_i \ll \gamma_M$$

Thus:

$$C_{\text{hive}} \ll C_{\text{mono}}$$

Accuracy improves through verification:

$$A_{\text{hive}} = 1 - \prod_{i \in \text{critical}} (1 - p_i)$$

which dominates:

$$A_{\text{mono}} = p_M$$

when multiple validated steps are composed.

Conclusion: Hivemind achieves strictly better scaling in latency, cost, and accuracy under decomposition.

3.3 Decomposer

The **Decomposer** is responsible for constructing and refining the execution DAG. Unlike classical partitioning problems, the Decomposer does not operate over arbitrary divisible units. **Kernels are semantically meaningful units of computation**, and therefore the system cannot directly choose the number of kernels n. Instead, it performs an iterative transformation process that moves the DAG toward a region of **optimal granularity** while respecting task structure.

3.3.1 Granularity as a First-Class Constraint

Earlier formulations of decomposition often appeal to the notion of an *irreducible unit* of computation. In practice, this notion is ill-defined. For agentic workloads, even small units (e.g., $\sim$200 tokens) may already incur overhead comparable to their execution cost due to prompt construction, routing, and coordination.

Therefore, the Decomposer replaces irreducibility with a **granularity objective**. Let each kernel v_i have an effective work w_i and an associated overhead β. Then the per-kernel efficiency is:

$$\eta_i = \frac{w_i}{w_i + \beta}.$$

For small w_i, $\eta_i \to 0$, meaning overhead dominates. For large w_i, $\eta_i \to 1$, but parallelism is lost. The goal is therefore not to minimize or maximize kernel size, but to keep all kernels within an **efficiency band**:

$$\eta_i \in [\eta_{\min}, \eta_{\max}].$$

This induces a **feasible granularity region** rather than a single optimal point.

3.3.2 Decomposition as Structured Search

The Decomposer operates as an iterative search over DAG space:

$$G_0 \to G_1 \to \cdots \to G_k,$$

where each transformation applies a local rewrite:

- **Split**: $v_i \to \{v_i^1, v_i^2, \dots\}$
- **Merge**: $\{v_i, v_j\} \to v_{ij}$
- **Rewire**: modify dependency edges to reduce sequential depth
- **Eliminate**: remove redundant or unused kernels

Each step attempts to improve:

$$\mathcal{L}(G) = T(G) + \lambda C(G) - \mu A(G).$$

Unlike classical optimization, the search space is constrained by:

- semantic boundaries of subtasks
- dependency correctness
- model capability requirements

Thus, the Decomposer is not solving a continuous optimization problem, but a **constrained combinatorial search**.

3.3.3 Approaching Optimal Granularity

Let n^* denote the ideal kernel count implied by the continuous approximation:

$$n^* \approx \sqrt{\frac{W\alpha}{\beta}}.$$

The Decomposer cannot directly set $n = n^*$, but can move toward it through controlled transformations.

Define:

$$\Delta T_{\text{split}} \approx \text{reduction in critical path from splitting},$$
$$\Delta C_{\text{split}} \approx \beta \quad (\text{new kernel overhead}),$$

A split is accepted if:

$$\Delta T_{\text{split}} > \lambda \Delta C_{\text{split}}.$$

Similarly, merging kernels yields:

$$\Delta T_{\text{merge}} > 0 \quad (\text{increase in critical path}),$$
$$\Delta C_{\text{merge}} < 0 \quad (\text{reduced overhead}),$$

and is accepted if:

$$|\Delta C_{\text{merge}}| > \frac{1}{\lambda} \Delta T_{\text{merge}}.$$

Thus, the system performs a **discrete approximation of gradient descent** toward the optimal region.

3.3.4 Critical Path Control

A key objective is minimizing:

$$T(G) = \max_{\pi} \sum_{v_i \in \pi} \ell_i.$$

The Decomposer explicitly biases transformations toward reducing critical-path weight:

- Prefer splitting kernels on the current critical path

- Avoid splitting kernels off the critical path unless they enable future parallelism
- Rewire dependencies to increase parallel width

Let π^* be the current critical path. A split of $v_i \in \pi^*$ reduces span if:

$$\ell_i > \ell_i^1 + \ell_i^2 + \beta.$$

Thus, splitting is only beneficial if:

$$\ell_i - (\ell_i^1 + \ell_i^2) > \beta.$$

This formalizes the intuition that splitting must overcome coordination overhead.

3.3.5 Compression and Model Allocation

The decomposer implicitly optimizes a **compression ratio**:

$$\rho = \frac{C(G)}{C_{\text{mono}}}.$$

Decomposition is beneficial only when:

$$\rho < 1.$$

However, compression arises not from splitting alone, but from enabling:

- routing low-complexity kernels to cheaper models
- isolating high-complexity regions for expensive models

Thus, the decomposer must produce kernels with:

$$\text{complexity}(v_i) \approx \text{model capability tiers}.$$

This aligns semantic granularity with economic efficiency.

3.3.6 Anti-Patterns and Constraints

The Decomposer must avoid the following structural failures:

God Kernel A single kernel containing all work:

$$n = 1.$$

Optimal only when:

$$W \text{ is small or tightly coupled.}$$

Trivial Kernels Kernels with $w_i \ll \beta$:

$$\eta_i \approx 0.$$

These should be merged aggressively.

False Parallelism Kernels that appear independent but share hidden context dependencies. These increase error and recomputation cost.

Excessive Depth Chains where:

$$|\pi^*| \gg \log n,$$

indicating poor parallelization.

3.3.7 Verification-Aware Decomposition

Verification is not uniform across kernels. We distinguish:

- **Correctness kernels**: produce boolean validity signals (tool-based, deterministic where possible)
- **Debug kernels**: perform re-analysis or regeneration (expensive, invoked conditionally)

Let:

$$c_{\text{correct}} \ll c_{\text{debug}}, \quad p_{\text{correct}} \approx 1.$$

The optimal policy is:

- Always apply correctness kernels to critical outputs
- Invoke debug kernels only upon failure

This yields expected verification cost:

$$C_{\text{verify}} \approx c_{\text{correct}} + (1 - p_i)c_{\text{debug}}.$$

Thus, verification cost scales with error probability rather than always incurring full cost.

3.3.8 Branch Kernels and Recursive Structure

Some kernels are marked as **branch kernels**, representing unresolved subproblems. These induce recursive decomposition:

$$v_i \to G_i$$

with depth constraint:

$$\text{recursion depth} \leq 3.$$

This enables hierarchical DAG construction while preventing explosion in coordination complexity.

3.3.9 Complexity of Decomposition

Let K be the number of refinement steps. Each step involves:

- local graph transformation: $O(1)$–$O(\deg(v))$
- global bookkeeping (topological updates): $O(n + m)$

Thus total decomposition overhead is:

$$O(K(n + m)).$$

Since K is typically small (bounded refinement), decomposition cost remains subdominant relative to execution cost.

3.3.10 Summary

The Decomposer is not a static planner, but a **dynamic optimization process** that:

- respects semantic kernel boundaries
- moves the DAG toward an optimal granularity region
- minimizes critical path while controlling overhead
- aligns kernel complexity with model capability tiers

Its role is to approximate an intractable global optimization problem through structured, local transformations that balance **parallelism**, **coordination cost**, and **probabilistic correctness**.

3.4 Granularity Optimization

Granularity emerges from the action of the Decomposer. It is not a directly controlled parameter, but the result of a sequence of split, merge, and rewiring operations applied under semantic and dependency constraints. The role of this subsection is to formalize how granularity influences system behavior, and to quantify the benefit of moving toward an optimal granularity region.

3.4.1 Granularity as Induced by Decomposition

Let $G = (V, E)$ be the execution DAG after k decomposition steps, with $n = |V|$ kernels and total task work:

$$W = \sum_{i=1}^{n} w_i.$$

Each kernel incurs overhead β, so total effective cost is:

$$C(G) = \sum_{i=1}^{n} (c_i^{\text{base}} + \beta).$$

Granularity is therefore characterized by the distribution $\{w_i\}$ and induced kernel count n, rather than by n alone.

Define the **granularity profile**:

$$\mathcal{G}(G) = \{w_1, w_2, \ldots, w_n\}.$$

The Decomposer acts on $\mathcal{G}(G)$ through local transformations.

3.4.2 Effect of Split on Latency and Cost

Consider a kernel v with work w on the critical path π^*. Splitting it into k subkernels:

$$w \to \{w_1, \ldots, w_k\}, \quad \sum w_i = w.$$

Latency Effect If the split produces parallelizable children, the new contribution to critical path becomes:

$$\max_i \ell_i = \alpha \max_i w_i.$$

Thus latency improvement is:

$$\Delta T_{\text{split}} = \alpha(w - \max_i w_i).$$

In the balanced case:

$$w_i = \frac{w}{k}, \quad \Delta T_{\text{split}} = \alpha w \left(1 - \frac{1}{k}\right).$$

Cost Effect Each new kernel introduces overhead:

$$\Delta C_{\text{split}} = (k-1)\beta.$$

Net Benefit Condition Define utility:

$$\Delta U_{\text{split}} = -\Delta T_{\text{split}} + \lambda \Delta C_{\text{split}}.$$

Split is beneficial iff:

$$\alpha(w - \max_i w_i) > \lambda(k-1)\beta.$$

Balanced case:

$$\alpha w \left(1 - \frac{1}{k}\right) > \lambda(k-1)\beta.$$

Solving approximately for $k = 2$:

$$w > \frac{2\lambda\beta}{\alpha}.$$

Thus, kernels smaller than this threshold should not be split.

3.4.3 Effect of Merge on Latency and Cost

Consider two kernels v_i, v_j with work w_i, w_j.

Cost Reduction

$$\Delta C_{\text{merge}} = -\beta.$$

Latency Increase If previously parallel, merged kernel contributes:

$$\ell_{ij} = \alpha(w_i + w_j).$$

Latency increase:

$$\Delta T_{\text{merge}} = \alpha \min(w_i, w_j).$$

Merge Condition Merge is beneficial iff:

$$\beta > \lambda^{-1}\alpha \min(w_i, w_j).$$

Thus, sufficiently small kernels should be merged.

3.4.4 Granularity Band

Combining split and merge conditions yields a stable region:

$$\frac{2\lambda\beta}{\alpha} \quad w_i \quad \frac{\beta}{\lambda^{-1}\alpha}.$$

Kernels outside this band are transformed:

- above band $\rightarrow$ split
- below band $\rightarrow$ merge

This defines a **granularity attractor region**.

3.4.5 Global Advantage of Optimal Granularity

Let $T_{\text{mono}} = \alpha W$.

Under decomposition into n kernels with balanced structure:

$$T(G) \approx \alpha \frac{W \log n}{n}.$$

Total overhead:

$$C_{\text{overhead}} = n\beta.$$

Thus total effective latency:

$$T_{\text{total}}(n) \approx \alpha \frac{W \log n}{n} + n\beta.$$

Improvement over monolithic:

$$\Delta T = \alpha W - T_{\text{total}}(n).$$

For $n \ll W$, improvement grows approximately:

$$\Delta T \sim \alpha W \left(1 - \frac{\log n}{n}\right).$$

Maximum improvement occurs near:

$$n^* \sim \sqrt{\frac{W\alpha}{\beta}}.$$

At this point:

$$T_{\text{total}}(n^*) \sim 2\sqrt{W\alpha\beta \log n^*}.$$

Thus:

$$\frac{T_{\text{mono}}}{T_{\text{hive}}} \sim \frac{\alpha W}{\sqrt{W\alpha\beta}} = \sqrt{\frac{W\alpha}{\beta}}.$$

Speedup scales sublinearly but grows with task size.

3.4.6 Interaction with Critical Path Structure

Granularity alone is insufficient; the DAG structure matters.

Let:

$$S(G) = \text{critical path length in nodes.}$$

Then:

$$T(G) = \sum_{i \in \pi^*} \ell_i \approx \alpha \sum_{i \in \pi^*} w_i.$$

Poor decomposition leads to:

$$S(G) \gg \log n,$$

causing:

$$T(G) \approx \alpha W.$$

Thus:

- good granularity requires both balanced w_i and shallow depth
- decomposer must jointly optimize **width and depth**

3.4.7 Context and Fragmentation Constraints

Let s_i denote context size required for kernel v_i.

Splitting increases duplication:

$$\sum s_i \geq s_{\text{original}}.$$

Excessive splitting leads to:

- context redundancy
- degraded p_i
- increased verification cost

Thus granularity must satisfy:

$$\sum s_i = O(W), \quad \text{not } \omega(W).$$

3.4.8 Summary

Granularity determines the tradeoff between parallel speedup and coordination cost. The Decomposer moves the system toward a stable region where:

- splitting yields diminishing latency returns
- merging begins to harm parallelism
- overhead remains bounded relative to useful work

Within this region, Hivemind achieves maximal practical advantage over monolithic execution, with speedup scaling as:

$$O\left(\sqrt{\frac{W\alpha}{\beta}}\right)$$

under ideal conditions.

3.5 Verification System

Execution in Hivemind is fundamentally **stochastic**: kernels produce outputs with non-zero probability of error. The purpose of the verification system is to transform this stochastic execution into a controlled process with bounded failure probability, while minimizing additional latency and cost.

Verification is not treated as a uniform post-processing step. Instead, it is integrated into the DAG as a structured set of kernels with distinct roles, costs, and guarantees.

3.5.1 Verification as a First-Class Graph Component

Let $G = (V, E)$ be the execution DAG. The verification system induces an augmented DAG:

$$G' = (V \cup V_{\text{ver}}, E \cup E_{\text{ver}}),$$

where V_{ver} are verification kernels and E_{ver} encode verification dependencies.

Each kernel v_i is associated with a verification pipeline:

$$v_i \to v_i^{\text{check}} \to v_i^{\text{debug}}.$$

However, not all stages are executed unconditionally. Verification is structured to minimize expected cost.

3.5.2 Verification Kernel Types

We distinguish two classes of verification kernels:

Correctness Kernels A correctness kernel produces a boolean validity signal:

$$v_i^{\text{check}} : O_i \to \{0, 1\}.$$

Key properties:

- **Low cost**: $c_{\text{check}} \ll c_i$
- **Low latency**: $\ell_{\text{check}} \ll \ell_i$
- **High reliability**: $p_{\text{check}} \approx 1$

Correctness kernels are designed to rely on **external tools** whenever possible (e.g., compilation, unit tests, schema validation), rather than model-based reasoning. This allows them to approximate deterministic validation.

Debug Kernels A debug kernel performs corrective computation:

$$v_i^{\text{debug}} : (I_i, O_i) \to O_i'.$$

Key properties:

- **High cost**: $c_{\text{debug}} \gg c_{\text{check}}$
- **High latency**: $\ell_{\text{debug}} \gg \ell_{\text{check}}$
- **Corrective capability**: can regenerate or refine outputs

Debug kernels are invoked only when correctness kernels detect failure.

3.5.3 Conditional Verification Execution

Verification follows a conditional execution model:

$$v_i \xrightarrow{\text{execute}} O_i$$

$$O_i \xrightarrow{v_i^{\text{check}}} \begin{cases} \text{accept} & \text{if } 1 \\ v_i^{\text{debug}} & \text{if } 0 \end{cases}$$

This yields expected cost:

$$C_i^{\text{ver}} = c_{\text{check}} + (1 - p_i)c_{\text{debug}},$$

and expected latency:

$$T_i^{\text{ver}} = \ell_{\text{check}} + (1 - p_i)\ell_{\text{debug}}.$$

Thus, expensive debug operations are only incurred on failure.

3.5.4 Retry and Effective Execution Model

Let p_i be the probability that kernel v_i produces a correct output prior to verification.

With verification and retry, the effective success probability per attempt becomes:

$$\tilde{p}_i = p_i + (1 - p_i)p_{\text{debug}}.$$

Expected number of attempts:

$$r_i = \frac{1}{\tilde{p}_i}.$$

Thus effective cost:

$$c_i^{\text{eff}} = r_i \left(c_i + c_{\text{check}} + (1 - p_i)c_{\text{debug}}\right),$$

and effective latency:

$$\ell_i^{\text{eff}} = r_i \left(\ell_i + \ell_{\text{check}} + (1 - p_i)\ell_{\text{debug}}\right).$$

This formulation integrates verification directly into the execution model.

3.5.5 Selective Verification Placement

Verification is not applied uniformly across all kernels. Let $\Pi^\star$ denote the set of correctness-critical kernels (those whose outputs influence final results).

The system assigns verification according to:

- **Full verification**: for $v_i \in \Pi^\star$
- **Partial or deferred verification**: for non-critical kernels

Let $q_i \in \{0, 1\}$ indicate whether verification is applied. Then:

$$C_{\text{total}} = \sum_i c_i^{\text{eff}}(q_i),$$

$$T_{\text{total}} = S_{\text{exec}}(G) + S_{\text{check}}(G, q).$$

Optimal placement satisfies:

$$\frac{\partial A}{\partial q_i} \gg \frac{\partial C}{\partial q_i}, \frac{\partial T}{\partial q_i}$$

for critical kernels, and the reverse for non-critical ones.

3.5.6 Parallel Verification and Aggregation

Let n_v be the number of verification kernels. Verification results:

$$x_i = v_i^{\text{check}}(O_i)$$

must be aggregated to determine system-level correctness.

Using parallel reduction (e.g., Blelloch scan [Blelloch, 1989]), aggregation can be performed in:

$$O(\log n_v) \text{ span}, \quad O(n_v) \text{ work}.$$

Thus:

$$S_{\text{check}}(G) = S_{\text{check,local}} + O(\log n_v).$$

Importantly, only aggregation benefits from logarithmic depth; individual checks remain kernel-local.

3.5.7 Verification-Induced DAG Transformations

Verification modifies the execution DAG in two ways:

- introduces additional nodes and edges,
- creates conditional execution paths (failure branches).

Let G be the original DAG. The augmented DAG G' satisfies:

$$|V'| = n + n_v, \quad |E'| = m + O(n_v).$$

Thus verification increases graph size linearly in the number of checked kernels.

3.5.8 Interaction with Granularity

Granularity affects verification in two competing ways:

- finer granularity $\Rightarrow$ more kernels $\Rightarrow$ more verification overhead,
- finer granularity $\Rightarrow$ smaller $w_i \Rightarrow$ higher p_i (simpler tasks).

Thus:

$$p_i = p(w_i), \quad \frac{dp}{dw_i} > 0,$$

and expected verification cost becomes:

$$C_{\text{ver}}(G) = \sum \left(c_{\text{check}} + (1 - p(w_i))c_{\text{debug}}\right).$$

This introduces a second-order tradeoff: splitting reduces error probability but increases verification overhead.

3.5.9 Guarantees and Limits

The verification system provides probabilistic guarantees:

$$A(G) = \prod_{i \in \Pi^\star} \tilde{p}_i.$$

In high-reliability regimes, verification can drive:

$$A(G) \to 1,$$

but at the cost of increased latency and computation.

Thus, full correctness is achievable asymptotically, but not without tradeoffs.

3.5.10 Summary

The verification system transforms unreliable kernel execution into a structured, probabilistically controlled process. Its design is governed by:

- separation of cheap correctness checks and expensive debug corrections,
- conditional execution to minimize expected cost,
- selective placement on critical regions,
- parallel aggregation to control latency.

This allows Hivemind to achieve high accuracy without sacrificing the latency and cost advantages of decomposition.

3.6 Tradeoff: Latency, Cost, and Accuracy

The core optimization problem of the Canvas is not the minimization of a single scalar objective, but the navigation of a **three-way tradeoff** between **latency**, **cost**, and **accuracy**. These quantities are coupled through decomposition, model assignment, and verification policy. Improving one usually requires sacrificing at least one of the others.

Let a task be decomposed into a DAG $G = (V, E)$ with $n = |V|$ kernels. For each kernel v_i, let:

$$w_i \text{ be its task work (e.g. effective token workload)}, \qquad m_i \in \mathcal{M} \text{ be the chosen model},$$

$$\ell_i = \ell(w_i, m_i), \qquad c_i = c(w_i, m_i), \qquad p_i = p(w_i, m_i)$$

where ℓ_i is latency, c_i is monetary cost, and p_i is probability that the kernel output is acceptable before verification-triggered retry.

We also introduce a verification policy $q_i \in [0, 1]$ for each kernel, where q_i parameterizes verification strength. In practice, q_i may represent whether the kernel is unverified, weakly verified, or strongly verified; analytically, it is convenient to treat it as continuous. Stronger verification increases runtime cost and latency, but also increases effective reliability.

Let the expected number of attempts for kernel v_i be

$$r_i = \frac{1}{\tilde{p}_i},$$

where $\tilde{p}_i = \tilde{p}_i(m_i, q_i)$ is the probability that one execution-plus-verification cycle succeeds. Then the expected effective cost of v_i is

$$c_i^{\text{eff}} = r_i \, c_i = \frac{c_i}{\tilde{p}_i},$$

and the expected effective latency contribution of v_i is

$$\ell_i^{\text{eff}} = r_i \, \ell_i + v_i^{\text{check}}(q_i),$$

where $v_i^{\text{check}}(q_i)$ is the direct verification overhead.

The expected total cost is therefore

$$C(G) = \sum_{i=1}^{n} \frac{c_i}{\tilde{p}_i},$$

while expected latency is governed by the weighted critical path:

$$T(G) = \max_{\pi \in \text{paths}(G)} \sum_{v_i \in \pi} \left(\frac{\ell_i}{\tilde{p}_i} + v_i^{\text{check}}(q_i) \right) \; + \; H_{\text{sched}}(G),$$

where $H_{\text{sched}}(G)$ is Canvas-level scheduling and coordination overhead.

Accuracy must also be modeled at the system level rather than per-kernel in isolation. Let $\Pi^\star$ denote a set of correctness-critical kernels whose failure can corrupt the final answer. If we assume independence as a first approximation, then end-to-end success probability is

$$A(G) \approx \prod_{v_i \in \Pi^\star} \tilde{p}_i.$$

Equivalently, the system failure probability is

$$1 - A(G) \approx 1 - \prod_{v_i \in \Pi^\star} \tilde{p}_i.$$

For high-accuracy regimes, it is often more informative to work in log-space:

$$-\log A(G) = \sum_{v_i \in \Pi^\star} -\log \tilde{p}_i.$$

This expression makes the accumulation of reliability debt explicit: every kernel on the correctness-critical region contributes additively to total error budget.

3.6.1 A Decomposition Parameterization

To analyze the tradeoff more concretely, let g denote a **decomposition level**: larger g means finer granularity. As g increases, the number of kernels $n(g)$ increases, average per-kernel work decreases, and overhead rises. We write:

$$n = n(g), \qquad \bar{w}(g) \approx \frac{W}{n(g)},$$

where W is total task work.

At a high level, finer decomposition tends to produce the following effects:

$T(g) \downarrow$ through reduced per-kernel latency and reduced critical-path work,
$C(g) \uparrow$ through more prompts, more coordination, and more verification calls,
$A(g) \uparrow$ or $\downarrow$ depending on whether specialization/verification dominates fragmentation error.

This last point is important: accuracy is *not* monotone in granularity. Decomposition can improve accuracy by allowing specialized models and targeted verification, but excessive fragmentation can destroy global context and introduce interface mismatch between kernels.

To make this explicit, we decompose accuracy into two competing effects:

$$A(g) \approx A_{\text{spec}}(g) \cdot A_{\text{cohere}}(g),$$

where:

- $A_{\text{spec}}(g)$ captures gains from specialization, tool use, and localized verification,
- $A_{\text{cohere}}(g)$ captures losses from fragmentation, boundary mismatch, and context breakage.

Typically,

$$A'_{\text{spec}}(g) > 0, \qquad A'_{\text{cohere}}(g) < 0,$$

so overall $A(g)$ may peak at an intermediate decomposition level.

3.6.2 Marginal Tradeoff Rates

The relevant question is not merely whether there is a tradeoff, but **how much one objective must be sacrificed to improve another by a given magnitude**. This is captured by marginal rates.

Suppose g is the control variable. Then:

$$\frac{dC}{dT} = \frac{dC/dg}{dT/dg}, \qquad \frac{dA}{dT} = \frac{dA/dg}{dT/dg}, \qquad \frac{dA}{dC} = \frac{dA/dg}{dC/dg}.$$

These ratios measure local exchange rates:

- $\frac{dC}{dT}$: additional cost paid per unit latency reduction,
- $\frac{dA}{dT}$: additional accuracy gained per unit latency sacrifice,
- $\frac{dA}{dC}$: additional accuracy gained per unit cost increase.

If the goal is to reduce latency by a small amount $\Delta T < 0$, then the first-order cost increase is

$$\Delta C \approx \frac{dC}{dT}\,\Delta T,$$

and the corresponding accuracy change is

$$\Delta A \approx \frac{dA}{dT}\,\Delta T.$$

Likewise, if the goal is to increase accuracy by $\Delta A > 0$, the first-order latency and cost sacrifices are

$$\Delta T \approx \frac{dT}{dA}\,\Delta A, \qquad \Delta C \approx \frac{dC}{dA}\,\Delta A.$$

These derivatives become especially informative near the operating frontier. If $\frac{dC}{dT}$ becomes very large in magnitude, then further latency reduction is entering a coordination-dominated regime and is no longer economical. If $\frac{dT}{dA}$ becomes very large, then the system is attempting to buy the last increments of reliability at excessive runtime cost.

3.6.3 Asymptotic Sacrifice Under a Simple Granularity Model

To make the exchange rates explicit, consider a stylized model where

$$T(g) = a\frac{\log n(g)}{n(g)} + b\, n(g),$$

$$C(g) = c\, W + d\, n(g) + e\, V(g),$$

$$-\log A(g) = \frac{\kappa}{n(g)^{\rho}} + \eta\, n(g)^{\sigma}.$$

Here:

- the first term in $T(g)$ captures reduced critical-path work under finer decomposition,
- the second term in $T(g)$ captures per-kernel overhead,
- cW is the base inference spend,
- $dn(g)$ is prompt / scheduling / bookkeeping overhead,
- $V(g)$ is verification effort,
- the first term in $-\log A(g)$ models specialization gains,
- the second term in $-\log A(g)$ models fragmentation/coherence loss.

For simplicity, suppose $n(g)$ itself is the effective control variable. Then:

$$\frac{dT}{dn} = a\frac{1-\log n}{n^2} + b,$$

$$\frac{dC}{dn} = d + e\, V'(n),$$

$$\frac{d(-\log A)}{dn} = -\kappa\rho n^{-(\rho+1)} + \eta\sigma n^{\sigma-1}.$$

This yields:

$$\frac{dC}{dT} = \frac{d + e\, V'(n)}{a\frac{1-\log n}{n^2} + b},$$

and

$$\frac{d(-\log A)}{dT} = \frac{-\kappa\rho n^{-(\rho+1)} + \eta\sigma n^{\sigma-1}}{a\frac{1-\log n}{n^2} + b}.$$

These expressions expose the asymptotics directly:

- In the **under-decomposed regime**, n is too small, so $dT/dn < 0$ with large magnitude. Increasing n yields large latency gains per additional kernel, so the sacrifice in cost per unit speedup is relatively small.
- Near the **latency-optimal region**, $dT/dn \approx 0$, which means $\frac{dC}{dT}$ blows up in magnitude. This is the point where small additional latency gains require disproportionately large cost increases.
- In the **over-decomposed regime**, the bn overhead term dominates, so adding more kernels worsens latency directly; both latency and cost increase together, and accuracy may also fall due to fragmentation.

Thus, if one insists on reducing latency by a factor of $(1-\varepsilon)$ once the system is already near the latency optimum, the required cost increase is superlinear in the distance to the optimum because the denominator dT/dn approaches zero. Put differently, the last increments of speed are the most expensive.

3.6.4 Tradeoff With Verification Strength

Verification introduces a second control axis independent of granularity. Let q denote verification strength. Then typically:

$$\frac{\partial T}{\partial q} > 0, \qquad \frac{\partial C}{\partial q} > 0, \qquad \frac{\partial A}{\partial q} > 0.$$

If stronger verification improves kernel success probability from $\tilde{p}_i$ to $\tilde{p}_i + \delta_i(q)$, then expected retry cost falls while direct checking overhead rises. For a single kernel:

$$\frac{\partial}{\partial q}\left(\frac{c_i}{\tilde{p}_i(q)}\right) = -\frac{c_i \tilde{p}_i'(q)}{\tilde{p}_i(q)^2},$$

while direct verification cost contributes

$$\frac{\partial v_i^{\text{check}}(q)}{\partial q} > 0.$$

Therefore stronger verification is worthwhile precisely when the decrease in retry burden dominates the increase in direct check overhead. This gives a per-kernel threshold rule:

$$\text{increase } q_i \quad \text{iff} \quad \frac{c_i \tilde{p}_i'(q_i)}{\tilde{p}_i(q_i)^2} \text{ and } \frac{\ell_i \tilde{p}_i'(q_i)}{\tilde{p}_i(q_i)^2}$$

are sufficiently large relative to the marginal cost and latency of stronger verification.

This also suggests a structural policy:

- use **strong verification** on correctness-critical kernels and expensive kernels,
- use **weaker or deferred verification** on cheap, easily retryable, or non-critical branches.

3.6.5 Practical Operating Frontier

The correct design target is therefore not a single optimum, but a **Pareto frontier**. A configuration is Pareto-optimal if no other configuration improves one of $\{T, C, A\}$ without worsening at least one of the others.

In practice, Canvas should operate on this frontier rather than optimize a fixed scalarization prematurely. Different workloads prefer different regions:

- interactive workloads prefer lower T,
- batch or low-margin workloads prefer lower C,
- high-stakes workflows prefer higher A.

The role of the decomposer is thus to move the DAG toward the appropriate region of the latency–cost–accuracy frontier while respecting semantic constraints on what can meaningfully be turned into a kernel.

3.7 Complexity Summary

The complexity of Canvas must be expressed in terms of both **total task work** and **graph structure**. Kernel count alone is not a meaningful asymptotic variable, because kernels are semantic units rather than arbitrary equal-sized fragments. Two DAGs with the same number of kernels may differ drastically in total work, critical-path length, verification burden, and coordination overhead.

We therefore use the following quantities:

- $W = \sum_{i=1}^{n} w_i$: total task work,
- $n = |V|$: number of kernels,
- $m = |E|$: number of dependency edges,
- $L(G)$: weighted critical-path length (span) of the execution DAG,
- $L_{\text{check}}(G)$: weighted critical-path length of the verification DAG,
- $H_{\text{coord}}(n, m)$: Canvas coordination overhead.

3.7.1 Execution Work and Span

The total **execution work** of the DAG is

$$W_{\text{exec}}(G) = \sum_{i=1}^{n} \ell_i,$$

or more precisely in expected form under retries,

$$\mathbb{E}[W_{\text{exec}}(G)] = \sum_{i=1}^{n} \frac{\ell_i}{\tilde{p}_i}.$$

This is the total amount of model-runtime consumed across all kernels.

However, under parallel execution, latency is governed not by total work but by **span**:

$$S_{\text{exec}}(G) = \max_{\pi \in \text{paths}(G)} \sum_{v_i \in \pi} \frac{\ell_i}{\tilde{p}_i}.$$

This is the weighted critical path of the execution DAG.

Accordingly, Hivemind's wall-clock latency satisfies

$$T_{\text{exec}}(G) = S_{\text{exec}}(G) + H_{\text{coord}}(n, m),$$

up to scheduler and queueing effects deferred to Swarm.

3.7.2 Decomposition and Graph-Optimization Complexity

Let the decomposer start from an initial graph G_0 and apply K local transformations (split, merge, reroute, eliminate, fuse, verify-insert). If each transformation examines at most local neighborhoods plus global graph bookkeeping, then one decomposition pass costs

$$O(n + m)$$

for traversal and bookkeeping, plus the cost of any model-assisted decomposition calls.

If there are K refinement steps, then the algorithmic graph-manipulation cost is

$$O(K(n + m)),$$

excluding LLM inference cost. In practice this term is usually dominated by model calls, but it matters because it is the irreducible systems overhead even when model latencies shrink.

For compiler-like graph optimizations:

- reachability / dead-kernel elimination is $O(n + m)$,
- topological sorting is $O(n + m)$,
- common-subexpression detection ranges from near-linear with hashing to superlinear depending on canonicalization policy,
- critical-path recomputation is $O(n + m)$ on a DAG after topological ordering.

3.7.3 Verification Complexity

Let the verification layer define a verification DAG G_{check} over kernel outputs. If verification is purely per-kernel and aggregated globally, then total verification work is

$$W_{\text{check}} = \sum_{i=1}^{n} v_i^{\text{check}}.$$

If one aggregates verdicts using parallel tree reduction, the aggregation work remains linear in the number of verified items, while aggregation span becomes logarithmic in that count. Concretely, for n_v verified outputs:

$$\text{aggregation work} = O(n_v), \qquad \text{aggregation span} = O(\log n_v),$$

assuming balanced parallel reduction.

This does *not* mean all verification is $O(\log n)$. The correct statement is:

$$T_{\text{check}}(G) = S_{\text{check,local}}(G) + O(\log n_v),$$

where $S_{\text{check,local}}(G)$ is the span of the actual local checking procedures themselves. If local checks are expensive, they dominate; the logarithmic term applies only to verdict aggregation and failure propagation, not to the semantic act of checking.

3.7.4 End-to-End Complexity

The total expected work of Canvas is therefore

$$\mathbb{E}[W_{\text{total}}] = \underbrace{\sum_{i=1}^{n} \frac{\ell_i}{\widetilde{p}_i}}_{\text{execution work}} + \underbrace{\sum_{i=1}^{n} v_i^{\text{check}}}_{\text{verification work}} + \underbrace{O(K(n+m))}_{\text{graph optimization / coordination bookkeeping}}.$$

The end-to-end latency is controlled by total span:

$$T_{\text{total}} = \underbrace{S_{\text{exec}}(G)}_{\text{execution critical path}} + \underbrace{S_{\text{check,local}}(G)}_{\text{local verification span}} + \underbrace{O(\log n_v)}_{\text{parallel verdict aggregation}} + \underbrace{H_{\text{coord}}(n,m)}_{\text{scheduling, synchronization, prompt overhead}}.$$

This is the responsible asymptotic picture:

- **work scales with task work** W, expected retries, and verification effort,
- **latency scales with span**, not raw work, under sufficient parallelism,
- **coordination overhead scales with graph size** (n, m),
- **logarithmic terms arise only in parallel reductions / aggregations**, not in the semantic execution of the task itself.

3.8 Discussion

The Canvas formalizes agentic execution as an optimization problem over directed acyclic graphs, where the fundamental quantities are **total work**, **span (critical-path latency)**, and **coordination overhead**. These quantities are not independent; they are coupled through the structure of the decomposition.

Let $W = \sum w_i$ denote total task work, and let $G = (V, E)$ with $n = |V|$ and $m = |E|$ denote the execution graph. The system operates under the decomposition:

$$T_{\text{total}}(G) = S_{\text{exec}}(G) + S_{\text{check}}(G) + H_{\text{coord}}(n, m),$$

where:

- $S_{\text{exec}}(G)$ is the execution span (critical-path work),
- $S_{\text{check}}(G)$ is the verification span,
- $H_{\text{coord}}(n, m)$ is coordination overhead.

Unlike classical systems, where minimizing total work is primary, the dominant objective here is minimizing span while controlling the growth of H_{coord}.

3.8.1 Structural Effects of Decomposition

Decomposition transforms a task of size W into a graph of size (n, m). While W is intrinsic to the task, (n, m) are induced by the Decomposer and determine the structure over which execution occurs.

At a minimum, coordination overhead satisfies:

$$H_{\text{coord}}(n, m) = \Omega(n + m),$$

reflecting scheduling, dependency resolution, and prompt construction costs.

For bounded-degree DAGs where $m = O(n)$:

$$H_{\text{coord}}(n, m) = \Omega(n).$$

Thus, increasing kernel count linearly increases coordination cost, independent of model execution.

At the same time, increasing n enables reduction in execution span. In the ideal balanced case:

$$S_{\text{exec}}(G) \approx \alpha \frac{W \log n}{n},$$

while in the degenerate sequential case:

$$S_{\text{exec}}(G) \approx \alpha W.$$

Therefore, decomposition introduces a structural tradeoff:

- increasing n reduces span sublinearly,
- but increases coordination cost linearly,
- and may increase span if dependencies are poorly structured.

3.8.2 Granularity as Equilibrium

Granularity determines how work W is distributed across kernels. As shown previously, splitting and merging operations induce a stable region in which:

- per-kernel work is sufficient to amortize overhead,
- further splitting yields diminishing latency returns,
- merging begins to degrade parallelism.

This corresponds to an equilibrium condition:

$$\frac{\partial S_{\text{exec}}}{\partial n} \approx -\frac{\partial H_{\text{coord}}}{\partial n}.$$

At this point, the marginal reduction in critical-path latency from increasing kernel count is balanced by the marginal increase in coordination overhead.

3.8.3 Verification as a Second-Order Constraint

Verification introduces additional structure into the graph. Let G_{check} denote the verification DAG. Its contribution to latency is:

$$S_{\text{check}}(G) = S_{\text{check,local}}(G) + O(\log n_v),$$

where n_v is the number of verification operations and the logarithmic term arises from parallel aggregation.

Verification affects the system in two ways:

- increases coordination and execution overhead,
- improves effective success probability, reducing expected recomputation.

Thus, verification modifies both S_{exec} (through retries) and H_{coord}, and must be placed selectively on correctness-critical regions.

3.8.4 Regimes of Operation

The system exhibits three distinct regimes depending on decomposition:

Under-decomposed regime

$$n \text{ small}, \quad S_{\text{exec}} \approx \alpha W.$$

Parallelism is underutilized; latency is dominated by total work.

Balanced regime

$$n \approx n^*, \quad S_{\text{exec}} \ll \alpha W, \quad H_{\text{coord}} \text{ moderate}.$$

Span reduction and coordination overhead are balanced; this is the optimal operating region.

Over-decomposed regime

$$n \text{ large}, \quad H_{\text{coord}} \gg S_{\text{exec}}.$$

Overhead dominates; latency and cost both increase, and accuracy may degrade due to fragmentation.

3.8.5 Governing Principle

The analysis yields a single governing principle:

Agentic execution is efficient when decomposition reduces critical-path work faster than it increases coordination overhead.

Equivalently, the system must ensure:

$$\Delta S_{\text{exec}} < -\Delta H_{\text{coord}}$$

for any transformation that increases kernel count.

This principle unifies the behavior of the Decomposer, the granularity dynamics, and the verification system. It defines the feasible region in which Hivemind achieves advantage over monolithic execution, and characterizes the design space of Agentic Execution Systems.

4 Hive: The Hierarchical Interface Layer

The **Hive** is the interface layer that connects the Canvas to real-world and digital environments. While the Canvas defines how tasks are executed as structured programs, the Hive defines **what is being operated on**. It provides a unified abstraction over heterogeneous systems—files, APIs, devices, software, and shared human workflows—by organizing them into a recursive, hierarchical structure.

Formally, the Hive maps external state into a structure that can be consumed by the Canvas as inputs and targets for execution. It enables **compositionality**, **reuse**, and **shared context** across tasks, and allows agentic execution to extend beyond isolated prompts into persistent systems.

4.1 Hive Structure

A Hive is defined as a recursive container:

$$H = (S, T, \{H_j\}),$$

where:

- S is the internal state space (logical representation of data),
- T is the set of **Tactiles** (interfaces to external systems),
- $\{H_j\}$ are sub-Hives.

Each Hive can contain arbitrary sub-Hives, forming a tree:

$$H_0 \supset H_1 \supset \cdots \supset H_k.$$

This hierarchy mirrors real-world structure:

- a user's environment (root Hive),
- sub-Hives for projects, systems, or domains,
- further decomposition into components or resources.

The recursive nature of Hives enables **local reasoning with global consistency**: kernels operate on localized state while remaining composable within the larger system.

4.2 Tactiles: Interface Primitives

Interaction with the external world occurs through **Tactiles**. A Tactile is an interface of the form:

$$\tau = (\text{read}, \text{write}, \text{schema}, \text{cost}, \text{latency}),$$

where:

- read and write define operations,
- schema defines input/output structure,

- cost and latency characterize execution.

We distinguish:

- **In-Tact**: read-only interfaces (e.g., file read, API query),
- **Ex-Tact**: write or action interfaces (e.g., file write, API call, device control).

Tactiles are exposed to the Canvas through kernels of type `AgentAction`. These kernels bridge the abstract DAG execution with real-world side effects.

4.3 Hive–Canvas Coupling

The Hive and Canvas interact through a mapping:

$$\Phi : H \to \mathcal{C},$$

where $\mathcal{C}$ is the Canvas state space.

Specifically:

- Hive state is materialized as Canvas inputs,
- Kernel outputs are written back to Hive through Tactiles,
- Dependencies between kernels correspond to data dependencies within the Hive.

This establishes a **bidirectional flow**:

- **Read path**: Hive $\to$ Canvas (context ingestion),
- **Write path**: Canvas $\to$ Hive (action / persistence).

Thus, the Canvas executes over a projection of the Hive, while the Hive persists and structures the results.

4.4 Hierarchical Decomposition and Sub-Canvases

The recursive structure of Hives enables hierarchical execution. A kernel may operate not only on raw data, but on an entire sub-Hive. In such cases, execution proceeds by spawning a **sub-Canvas**:

$$v_i \to G_i,$$

where G_i is a DAG defined over the sub-Hive H_i.

This enables:

- localized decomposition within subdomains,
- reuse of decomposition patterns,
- bounded complexity through depth-limited recursion.

Let d_H denote Hive depth. To prevent coordination explosion:

$$d_H = O(1)$$

in practice (typically ≤ 3).

Thus, the system maintains hierarchical expressivity without unbounded recursion.

4.5 Shared Context and Multi-Agent Consistency

A key advantage of the Hive abstraction is **shared context**. Multiple users or processes may operate on the same Hive or sub-Hive:

$$H_{\text{shared}}.$$

This allows:

- reuse of intermediate results,
- elimination of redundant computation,
- consistent state across concurrent executions.

Formally, if two tasks G_1 and G_2 operate on the same Hive, their combined work is:

$$W_{\text{combined}} \leq W_1 + W_2,$$

with strict inequality when subgraphs overlap.

This is analogous to common subexpression elimination at the system level.

4.6 Comparison to Monolithic and Flat Agent Systems

Traditional agent systems operate over flat contexts:

$$\text{context} = \text{prompt} + \text{history}.$$

Such systems lack:

- persistent structured state,
- compositional reuse,
- explicit data dependencies.

In contrast, the Hive provides:

- **persistent structured state** via hierarchical storage,
- **compositional interfaces** via Tactiles,
- **explicit dependency structure** via Canvas integration.

Let R denote reusable work. In flat systems:

$$R \approx 0,$$

while in Hive-based systems:

$$R = \Omega(W_{\text{shared}}),$$

where W_{shared} is overlapping work across tasks.

Thus, the Hive enables asymptotic reduction in redundant computation across repeated or collaborative workflows.

4.7 AIoT Interpretation

The Hive abstraction generalizes naturally to an **AIoT (Artificial Intelligence of Things)** model. Each device, service, or system is represented as a sub-Hive with associated Tactiles. This allows:

- unified control across heterogeneous systems,
- consistent abstraction across physical and digital domains,
- integration of sensing, reasoning, and actuation within the same framework.

Unlike traditional IoT systems, which are device-centric, the Hive is **task-centric**: devices and services are organized around their role in computation rather than their physical identity.

4.8 Discussion

The Hive extends the Canvas from a pure execution model to a **persistent, compositional system**. It enables:

- hierarchical structuring of environments,
- reuse and sharing of computation,
- integration of external systems into the execution graph.

Together, the Canvas and Hive define a unified model in which:

- tasks are represented as DAGs,
- environments are represented as hierarchical state spaces,
- execution consists of transformations between the two.

This establishes the foundation for scalable, real-time agentic systems operating over complex environments.

5 Swarm: The Execution Runtime

The **Swarm** is the runtime system responsible for executing the DAG defined by the Canvas over the state defined by the Hive. It manages model allocation, kernel scheduling, concurrency, and latency hiding. While the Canvas defines the structure of computation, the Swarm determines how that structure is realized in time.

The primary objective of the Swarm is to minimize **wall-clock latency**:

$$T_{\text{wall}} = \text{time from task submission to completion},$$

under the constraints of model availability, network latency, and coordination overhead.

5.1 Execution Model

Let $G = (V, E)$ be the execution DAG. Each kernel $v_i \in V$ is executed as a remote model invocation with latency:

$$\ell_i = \ell(w_i, m_i),$$

where w_i is kernel work and m_i is the assigned model.

Define:

- $S_{\text{exec}}(G)$: execution span (critical path),
- Q_i: queueing delay before execution,
- O_i: per-kernel overhead (serialization, routing, etc.).

Then wall-clock latency is:

$$T_{\text{wall}} = \max_{\pi \in \text{paths}(G)} \sum_{v_i \in \pi} \left(\ell_i + Q_i + O_i\right).$$

The role of the Swarm is to minimize:

$$\sum Q_i + \sum O_i$$

along the critical path.

5.2 Model Pool and Parallel Execution

Let $\mathcal{M}$ denote the set of available models. The Swarm maintains a **model pool** with effectively elastic capacity:

$$|\mathcal{M}| \to \infty \quad \text{(idealized)}.$$

At time t, let $R(t)$ denote the set of ready kernels (all dependencies satisfied). The Swarm schedules:

$$\forall v_i \in R(t), \quad \text{execute immediately}.$$

Thus, in the ideal case:

$$Q_i = 0.$$

This realizes the theoretical assumption of unbounded parallelism from the Canvas.

5.3 Latency Hiding via Asynchronous Execution

In practice, kernel execution includes:

- network latency L_{net},
- model inference latency L_{inf},
- serialization overhead L_{ser}.

Sequential execution incurs:

$$T_{\text{seq}} = \sum_{i=1}^{n} (\ell_i + L_{\text{net}} + L_{\text{ser}}).$$

Under Swarm execution, these costs are overlapped across kernels. For a layer of width k:

$$T_{\text{layer}} = \max_{i=1}^{k} (\ell_i + L_{\text{net}} + L_{\text{ser}}).$$

Thus total latency becomes:

$$T_{\text{wall}} \approx \sum_{\text{layers}} \max_{v_i \in \text{layer}} \ell_i.$$

This collapses additive latency into **layer-wise maxima**, achieving latency hiding.

5.4 Queueing Effects and Throughput

In realistic systems, model capacity is finite. Let μ be the service rate (kernels per second per model), and let λ be the arrival rate of ready kernels.

Using an M/M/k approximation, expected queueing delay is:

$$\mathbb{E}[Q] \approx \frac{\rho^{\sqrt{2(k+1)}-1}}{k\mu(1-\rho)}, \quad \rho = \frac{\lambda}{k\mu}.$$

To maintain low latency:

$$\rho \ll 1.$$

Thus, Swarm must maintain:

$$k \gg \frac{\lambda}{\mu}.$$

This justifies maintaining an **overprovisioned model pool** to eliminate queueing on the critical path.

5.5 Critical Path Acceleration

Let π^* denote the critical path. Only kernels on π^* directly affect latency:

$$T_{\text{wall}} \approx \sum_{v_i \in \pi^*} (\ell_i + Q_i + O_i).$$

Thus, Swarm prioritizes:

- scheduling critical-path kernels immediately,
- assigning faster or more reliable models to critical nodes,
- deprioritizing non-critical branches.

Formally, let d_i be the distance of v_i to the sink node. Then priority is:

$$\text{priority}(v_i) \propto d_i.$$

This is analogous to critical-path scheduling in parallel systems.

5.6 Model Allocation and Cost-Latency Tradeoff

Each kernel may be executed on different models:

$$m_i \in \mathcal{M}.$$

Let:

$$\ell_i(m), \quad c_i(m), \quad p_i(m)$$

be latency, cost, and success probability for model m.

The Swarm solves:

$$\min_{m_i} \quad \ell_i(m_i) + \lambda c_i(m_i) - \mu p_i(m_i).$$

This yields:

- high-performance models on critical path,
- cheaper models on parallel branches,
- adaptive assignment based on kernel importance.

5.7 Comparison to Monolithic Execution

In monolithic execution:

$$T_{\text{mono}} = \alpha_M W,$$

where all work is processed sequentially.

In Swarm execution:

$$T_{\text{hive}} \approx S_{\text{exec}}(G) + H_{\text{coord}}.$$

Under balanced decomposition:

$$S_{\text{exec}}(G) \approx \alpha \frac{W \log n}{n}.$$

Thus speedup:

$$\frac{T_{\text{mono}}}{T_{\text{hive}}} \approx \frac{\alpha_M W}{\alpha \frac{W \log n}{n}} = \frac{\alpha_M}{\alpha} \cdot \frac{n}{\log n}.$$

For sufficiently large n, this yields substantial improvement.

5.8 Latency Regimes

The Swarm exhibits three regimes:

Compute-bound

$$T_{\text{wall}} \approx S_{\text{exec}}(G).$$

Coordination-bound

$$T_{\text{wall}} \approx H_{\text{coord}}.$$

Queue-bound

$$T_{\text{wall}} \approx \sum Q_i.$$

Optimal operation maintains:

$$Q_i \approx 0, \quad H_{\text{coord}} \ll S_{\text{exec}}.$$

5.9 Discussion

The Swarm transforms the theoretical parallelism of the Canvas into practical performance. Its key function is to ensure that:

- kernels are executed as soon as they become ready,
- latency components are overlapped rather than accumulated,
- critical-path execution is prioritized.

This yields the central principle:

Latency is determined by the critical path only if all other latency is successfully hidden.

By maintaining high concurrency and low queueing, the Swarm ensures that Hivemind approaches its theoretical performance limits, enabling real-time execution of complex, structured tasks.

6 Experiment

6.1 Experimental Setup

To evaluate whether the theoretical advantages of Hivemind translate to practical LLM workloads, we evaluate the system on **eight tasks of varying difficulty and structural complexity**. The tasks are designed to require different amounts of reasoning, implementation, and decomposition, allowing the experiment to measure both the efficiency and capability of Hivemind across workloads with different cognitive requirements.

We compare three configurations:

- **GPT-5.5 Monolithic**: GPT-5.5 receives the complete task and independently performs planning, reasoning, implementation, and generation.
- **DeepSeek V3.1 Monolithic**: DeepSeek V3.1 receives the same complete task and independently attempts to solve it.
- **Hivemind**: GPT-5.5 serves as the **Planner**, while multiple instances of DeepSeek V3.1 serve as parallel **Workers**. The Planner decomposes the original task into kernels and specifies the responsibilities, dependencies, and interfaces of each kernel. The resulting kernels are then dispatched to Workers for execution according to the generated DAG.

This configuration separates **planning capability** from **execution cost**. GPT-5.5 provides the stronger reasoning capability necessary to identify the structure of the task and construct an appropriate decomposition, while the majority of implementation and generation is delegated to the less expensive DeepSeek V3.1 Workers.

For each configuration, we measure three primary quantities:

- **Quality**: the degree to which the generated result successfully completes the assigned task;
- **Cost**: the total monetary inference cost required to complete the task;
- **Execution Time**: the wall-clock time between submission of the task and completion of the generated result.

For Hivemind, both Planner and Worker inference are included in the reported cost. Similarly, execution time includes planning, kernel dispatch, and Worker execution rather than measuring parallel execution in isolation.

6.2 Aggregate Results

Across the eight experimental tasks, Hivemind achieves approximately **equal average output quality** to monolithic GPT-5.5 while requiring substantially less inference cost and execution time.

The average monetary cost per task for monolithic GPT-5.5 is

$$C_{\text{Mono}} = \$1.19,$$

while the average cost for Hivemind is

$$C_{\text{Hive}} = \$0.417.$$

The resulting relative cost reduction is therefore

$$R_C = 1 - \frac{C_{\text{Hive}}}{C_{\text{Mono}}} = 1 - \frac{0.417}{1.19} \approx 0.650.$$

Thus, Hivemind reduces average inference cost by approximately

$$65.0\%$$

while maintaining approximately equal output quality.

Equivalently, the cost ratio between the two systems is

$$\frac{C_{\text{Mono}}}{C_{\text{Hive}}} = \frac{1.19}{0.417} \approx 2.85.$$

At the observed average cost, the inference budget required for one monolithic GPT-5.5 execution is therefore sufficient for approximately 2.85 Hivemind executions.

Hivemind also reduces average wall-clock execution time by approximately **28%**. Let T_{Mono} denote the average execution time of monolithic GPT-5.5. The observed Hivemind execution time can then be expressed as

$$T_{\text{Hive}} \approx 0.72 T_{\text{Mono}}.$$

This result is consistent with the DAG-based execution model described in the previous section. The Planner introduces an additional serial component before Worker execution. However, after the DAG has been constructed, kernels without unresolved dependencies may execute concurrently. Consequently, the Worker portion of execution is bounded primarily by the critical path rather than by the sum of all kernel execution times.

The reduction in cost is substantially larger than the reduction in execution time. This difference follows naturally from the two distinct mechanisms responsible for the observed improvement. Parallel execution reduces wall-clock latency by exploiting concurrency within the DAG, whereas heterogeneous model assignment additionally reduces the cost of each unit of computation. Hivemind therefore benefits simultaneously from **computational parallelism** and **model specialization**.

6.3 Planner Cognitive Load

During the initial experiments, Hivemind did not provide a substantial cost advantage over monolithic GPT-5.5. This result initially appeared inconsistent with the expected cost advantage of delegating kernel execution to the substantially cheaper DeepSeek V3.1 Workers.

Analysis of the generated Planner traces revealed that the primary source of the problem was not Worker cost, but the **cognitive load retained by the Planner**.

When GPT-5.5 received the complete task with a sufficiently large reasoning budget, it frequently performed substantially more work than was required for decomposition. Instead of only determining how the task should be divided, the Planner reasoned through significant portions of the solution before constructing the Worker instructions.

The resulting computation can be approximated as

$$\text{Solve}_{\text{Planner}} + \text{Execute}_{\text{Workers}},$$

rather than the intended architecture,

$$\text{Decompose}_{\text{Planner}} + \text{Solve}_{\text{Workers}}.$$

In the first configuration, the expensive Planner continues to perform much of the cognitive work that would have been performed by monolithic GPT-5.5. Worker inference is subsequently added on top of this computation. The architecture therefore distributes *generation* without necessarily distributing *cognition*.

This distinction is important because parallelism alone does not imply cost reduction. If the Planner internally solves each kernel before dispatching it, the system may execute the same cognitive work twice: once implicitly within the Planner and once explicitly within the Workers.

To address this problem, we modify the Planner in two ways.

First, we impose a stricter **reasoning-token budget**, limiting the amount of computation the Planner can spend reasoning about the underlying solution.

Second, we modify the Planner prompt such that its objective is explicitly restricted to decomposition. The Planner is instructed to identify kernels, dependencies, interfaces, constraints, and expected outputs without attempting to solve the kernels themselves.

The desired Planner transformation is therefore

$$P(x) \to G,$$

where x is the original task and G is the resulting task DAG, rather than

$$P(x) \to (G, \hat{y}),$$

where $\hat{y}$ represents an implicit solution constructed by the Planner during decomposition.

After restricting Planner cognition in this manner, the cost advantage of Hivemind becomes substantially more pronounced. The resulting system preserves the high-level decomposition capability of GPT-5.5 while transferring a larger fraction of the actual problem-solving workload to DeepSeek V3.1.

This result suggests that the efficiency of a heterogeneous AES depends not only on **which models participate in the computation**, but also on **how cognitive work is distributed between them**.

6.4 Capability–Cost Tradeoff

The comparison against monolithic DeepSeek V3.1 demonstrates the opposite side of this tradeoff.

A straightforward method for reducing inference cost is to execute the entire task using a cheaper model. However, this approach reduces both cost and the maximum cognitive capability available to every stage of the computation.

At approximately equal aggregate inference cost, Hivemind successfully completes an average of **2.4 additional tasks** compared with monolithic DeepSeek V3.1.

The three configurations therefore represent different points on a capability–cost spectrum.

Monolithic GPT-5.5 applies high cognitive capability throughout the entire computation, including portions of the task that may not require it. This provides strong task performance but incurs high cost.

Monolithic DeepSeek V3.1 applies a cheaper model throughout the computation. This minimizes the cost of individual tokens but may provide insufficient reasoning capability for the most difficult components of a task.

Hivemind instead selectively applies GPT-5.5 to the high-level decomposition problem and delegates the resulting lower-level problems to DeepSeek V3.1. The system therefore retains access to stronger cognition without requiring that stronger cognition to generate the majority of the solution.

These results indicate that the relevant optimization problem is not simply to identify the model with the lowest cost,

$$\min_{m \in M} C(m),$$

nor the model with the greatest capability,

$$\max_{m \in M} A(m),$$

but to determine how models of different capabilities should be assigned to different portions of the total cognitive workload.

6.5 Cognitive Cascade

The Planner experiment suggests a more general hypothesis about heterogeneous AES design. We refer to this hypothesis as the **Cognitive Cascade**.

Suppose a task is decomposed into a set of kernels

$$K = \{k_1, k_2, \ldots, k_n\},$$

and let

$$M = \{m_1, m_2, \ldots, m_r\}$$

denote the available model pool.

Each kernel k_i requires some minimum cognitive capability r_i for reliable execution. Similarly, each model m_j possesses some effective cognitive capability q_j.

A valid assignment function

$$\phi : K \to M$$

must assign kernels to models such that the assigned model possesses sufficient capability:

$$q_{\phi(i)} \geq r_i.$$

If the cost of executing kernel k_i using model m_j is denoted by

$$C(k_i, m_j),$$

then the ideal assignment is

$$\phi^* = \operatorname*{argmin}_{\phi} \sum_{i=1}^{n} C(k_i, m_{\phi(i)})$$

subject to

$$q_{\phi(i)} \geq r_i \qquad \forall i \in \{1, \ldots, n\}.$$

Under this formulation, assigning a model whose capability substantially exceeds r_i represents **cognitive overprovisioning**. The kernel succeeds, but the system pays for capability that the kernel does not require.

Conversely, assigning a model such that

$$q_{\phi(i)} < r_i$$

represents **cognitive underprovisioning**. Although the immediate inference cost may be lower, the probability of failure increases, potentially introducing verification, retry, and escalation costs.

The optimal assignment should therefore place each kernel on the **least expensive model possessing sufficient cognitive capability to reliably execute it**.

This produces a cascade of model capability rather than a fixed Planner–Worker hierarchy. Simple kernels may be assigned to extremely inexpensive models, intermediate kernels to stronger general-purpose Workers, and difficult kernels to frontier reasoning models. If verification determines that a kernel has failed, the kernel may move upward through the cascade to a more capable model.

For example, consider an ordered model pool

$$q_1 < q_2 < \cdots < q_r.$$

An initial model assignment may be expressed as

$$\phi(k_i) = \min \{m_j \in M \mid q_j \geq r_i\}.$$

If execution by m_j fails verification, the kernel can be escalated to m_{j+1} without affecting unrelated kernels in the DAG. The system therefore allocates additional cognition only where evidence indicates that additional cognition is required.

6.6 Specialized Cognitive Capability

The scalar representation q_j is useful for describing the basic Cognitive Cascade, but model capability is unlikely to be one-dimensional.

Different kernels may require fundamentally different forms of cognition. One kernel may require mathematical reasoning, another may primarily require code generation, while another may depend on long-context comprehension or visual reasoning.

We therefore generalize model capability into a vector

$$\mathbf{q}_j = \begin{bmatrix} q_j^{\text{reason}} \\ q_j^{\text{code}} \\ q_j^{\text{context}} \\ \vdots \end{bmatrix},$$

and similarly define the cognitive requirement of kernel k_i as

$$\mathbf{r}_i = \begin{bmatrix} r_i^{\text{reason}} \\ r_i^{\text{code}} \\ r_i^{\text{context}} \\ \vdots \end{bmatrix}.$$

A model is suitable for a kernel when its relevant capabilities satisfy the kernel's requirements. In the simplest component-wise formulation,

$$\mathbf{q}_{\phi(i)} \geq \mathbf{r}_i.$$

This extension implies that the optimal model for a kernel is not necessarily the strongest general-purpose model available. A smaller model specialized for the kernel's dominant cognitive requirement may satisfy $\mathbf{r}_i$ at substantially lower cost.

The Cognitive Cascade may therefore operate across both **capability levels** and **capability specializations**. Rather than constructing a hierarchy consisting only of increasingly powerful general-purpose models, the Model Pool may contain models specialized for different forms of computation.

6.7 Implications and Next Experiment

The present experiment does not establish the optimal Cognitive Cascade or demonstrate that the cognitive requirement of an arbitrary kernel can be reliably estimated before execution. Instead, it provides empirical motivation for investigating such an assignment.

The observed results establish three preliminary findings.

First, distributing generation across cheaper Workers can preserve the quality of a substantially more expensive monolithic model while reducing both cost and execution time.

Second, simply introducing a strong Planner is insufficient. If excessive cognitive work remains within the Planner, the cost advantage of heterogeneous execution can disappear. Efficient decomposition therefore requires explicit control over the distribution of cognition itself.

Third, the comparison with monolithic DeepSeek V3.1 indicates that replacing an expensive model uniformly with a cheaper model is also insufficient. Some portions of the workload benefit substantially from access to stronger cognition.

Together, these observations suggest the existence of an intermediate optimum in which different portions of a task receive different amounts and types of model capability.

The next experiment will investigate this hypothesis directly. Rather than using a fixed GPT-5.5 Planner and DeepSeek V3.1 Worker configuration, we will introduce multiple models with different costs, capabilities, and specializations into the Model Pool. Kernels will then be assigned across this pool according to their estimated cognitive requirements.

The resulting experiment will test whether the total cost of a task can be further reduced while preserving output quality by approximating

$$\phi^* = \operatorname*{argmin}_{\phi} C_{\text{total}}(\phi)$$

subject to a required system-level accuracy

$$A_{\text{Hive}}(\phi) \geq A_{\text{target}}.$$

If such an assignment can be estimated reliably, the Cognitive Cascade would extend Hivemind from a system that parallelizes computation into one that also **allocates cognition as a computational resource**.

References

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, et al. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2202.11464*, 2022.

Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

Keith D Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2012.

Jeffrey D Ullman. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

Edward G Coffman and Ronald L Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3): 200–213, 1972.

Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.

Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. In *ACM SIGPLAN Workshop on Intermediate Representations*, 1995.

Guy E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989. doi:10.1109/12.42122.