

HIVEMIND: OS-Inspired Scheduling for Concurrent LLM Agent Workloads

Justice Owusu Agyemang^{*1,2,3}, Jerry John Kponyo^{†3}, Obed Somuah^{‡3}, Elliot Amponsah^{§3},
Godfred Manu Addo Boakye^{¶3}, and Kwame Opuni-Boachie Obour Agyekum^{||2}

¹Sperix Labs

²VIA Cybersecurity Lab, KNUST

³Quantum and Assistive Technologies Lab, KNUST

April 2026

Abstract

When multiple LLM coding agents share a rate-limited API endpoint, they exhibit resource contention patterns analogous to unscheduled OS processes competing for CPU, memory, and I/O. In a motivating incident, 3 of 11 parallel agents died from connection resets and HTTP 502 errors—a 27% failure rate—despite the API having sufficient aggregate capacity to serve all 11 sequentially. We present HIVEMIND, a transparent HTTP proxy that applies five OS-inspired scheduling primitives—admission control, rate-limit tracking, AIMD backpressure with circuit breaking, token budget management, and priority queuing—to eliminate the failure modes caused by uncoordinated parallel execution. The proxy requires zero modifications to existing agent code and supports Anthropic, OpenAI, and local model APIs via auto-detected provider profiles. Our evaluation across seven scenarios (5–50 concurrent agents) shows that uncoordinated agents fail at 72–100% rates under contention, while HIVEMIND reduces failures to 0–18% and eliminates 48–100% of wasted compute. An ablation study reveals that trans-

parent retry—not admission control—is the single most critical primitive, but the primitives are most effective in combination. Real-world validation against Ollama confirms that HIVEMIND adds under 3 ms of proxy overhead per request. The system is open-source under the MIT license.

1 Introduction

The emergence of tool-augmented large language models has shifted software-engineering assistants from suggestion engines to autonomous agents that read, write, and execute code on a developer’s behalf [1, 2, 3, 4]. When users spawn multiple such agents in parallel—a natural pattern for tasks like generating test suites, writing proof-of-concept exploits, or refactoring across modules—the agents compete for shared resources: API rate limits (requests and tokens per minute), network connections (concurrent connection limits per endpoint), context windows (fixed per model), and API-key quotas (billing and access limits).

This resource contention leads to agent failures. The pattern is structurally identical to the contention that motivated operating-system schedulers: multiple processes competing for CPU, memory, and I/O without coordination leads to thrashing, starvation, and deadlock [5, 6]. Yet current agent orchestra-

^{*}jay@sperixlabs.org, jay@knust.edu.gh

[†]jjkponyo.soe@knust.edu.gh

[‡]

[§]eamponsah52@st.knust.edu.gh

[¶]gmaboakye@st.knust.edu.gh

^{||}kooagyekum@knust.edu.gh

Table 1: Results of 11 uncoordinated concurrent agents (April 15, 2026).

Outcome	Count	%
Completed successfully	8	73
Died (ECONNRESET)	2	18
Died (HTTP 502)	1	9
Tokens wasted (dead agents)	~ 135 K	

tion frameworks—LangChain [7], CrewAI [8], AutoGen [9], Semantic Kernel [10]—treat the LLM API as an unlimited resource, providing composition mechanisms (chains, crews, multi-agent conversations) but not resource management. They are, in OS terms, running a multi-process system without a scheduler.

Motivating observation. On April 15, 2026, we spawned 11 concurrent Claude Code agents to generate proof-of-concept scripts for security findings. All 11 shared one Anthropic API key through a single network proxy. Three agents died: two from ECONNRESET and one from HTTP 502. Each dead agent had consumed approximately 45 000 tokens before failing—a total waste of ~ 135 000 tokens and ~ 15 minutes of wall time. The eight surviving agents completed successfully because they happened to stagger their requests enough to avoid the bottleneck.

Key insight: if the 11 agents had been staggered by just 5 seconds each, all 11 would have succeeded. The problem is not capacity—it is coordination.

Contribution. We present HIVEMIND, a scheduling system that applies OS scheduling principles to concurrent LLM agent workloads. The contributions are:

1. A **formal analogy** mapping OS resource-management concepts (admission control, congestion control, budgeting, priority scheduling) to the LLM agent domain (Table 2).
2. A **transparent HTTP proxy** that implements five scheduling primitives—admission control via condition variables, provider-aware rate-limit tracking, AIMD backpres-

sure with circuit breaking, per-agent token budgets, and priority queuing with dependency DAGs—requiring zero modifications to existing agent code.

3. An **evaluation** across seven scenarios showing 72–100% failure reduction, and an ablation study revealing that transparent retry is the single most critical primitive.
4. An **open-source implementation** supporting Anthropic, OpenAI, Azure OpenAI, Google AI, and local models (Ollama, MLX) via auto-detected provider profiles.

The remainder of this paper is organised as follows. Section 2 reviews the relevant background. Section 3 describes the proxy architecture and five scheduling primitives. Section 4 covers key implementation decisions. Section 5 reports evaluation results, ablation, and real-world validation. Section 6 surveys related work. Section 7 discusses tradeoffs and limitations, and Section 8 concludes.

2 Background

2.1 LLM Coding Agents

A growing class of developer tools embed an LLM in an edit–test–commit loop. Claude Code [1], Cursor [3], GitHub Copilot [4], OpenAI Codex CLI [2], and Devin [11] each grant the model access to the local filesystem, a shell, and often a language server. The SWE-bench benchmark [12] and the SWE-agent framework [13] have further demonstrated that agents can resolve real GitHub issues end-to-end, making reliable API access a critical capability.

Each agent is a long-running, stateful process that makes repeated API calls over a multi-turn conversation. A single agent session may consume 50 000–500 000 tokens across dozens of API calls, with each call dependent on the previous response. When an API call fails mid-session, the agent typically cannot recover: it has consumed tokens, modified files, and accumulated context that is lost on restart.

2.2 OS Scheduling Principles

The resource contention patterns exhibited by concurrent LLM agents are structurally identical to those solved by operating system schedulers [5, 14]:

- **Admission control.** Limiting the number of concurrent processes to prevent thrashing. Dijkstra’s semaphore [15] is the classical mechanism.
- **Congestion control.** TCP’s Additive Increase / Multiplicative Decrease (AIMD) algorithm [16, 17] adjusts sending rate based on observed congestion signals (packet loss, increased RTT).
- **Circuit breaking.** The circuit breaker pattern [18] stops sending requests to a failing service, allowing it to recover before resuming load.
- **Resource budgeting.** Per-process memory limits, CPU quotas, and the OOM killer prevent any single process from monopolising shared resources.
- **Priority scheduling.** Shortest-job-first and priority queues [5] ensure that high-value or short tasks are serviced before long or low-priority ones.

2.3 The OS–LLM Agent Analogy

We formalise the mapping between OS concepts and LLM agent orchestration in Table 2. This analogy is not merely illustrative—it is structurally precise. Each OS mechanism addresses a specific resource contention failure mode that has a direct counterpart in the LLM agent domain.

2.4 Why Existing Frameworks Fail

Table 3 compares the scheduling capabilities of existing agent orchestration frameworks. None provides the full set of primitives needed to manage concurrent API access.

3 Architecture

HIVEMIND is implemented as a transparent HTTP reverse proxy that sits between agents

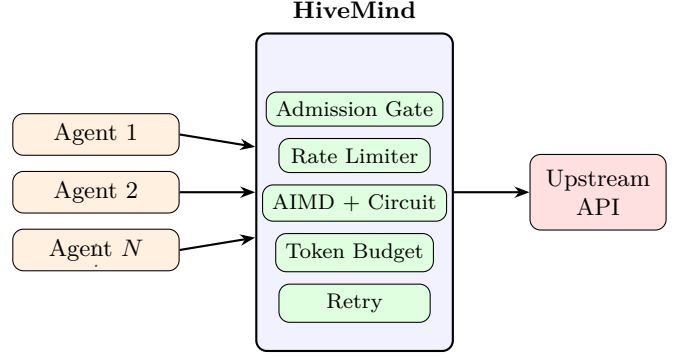


Figure 1: Architecture of HIVEMIND. Agents connect to the local proxy; requests pass through five scheduling layers before reaching the upstream API. The proxy is transparent: agents require zero code changes.

and the upstream LLM API provider (Figure 1). Agents make normal API calls to `http://localhost:8765/v1/messages`; HIVEMIND applies all scheduling logic before forwarding to the upstream provider.

This design has four advantages: (1) *zero agent modification*—works with any framework, SDK, or language; (2) *provider agnostic*—same proxy for Anthropic, OpenAI, Ollama, or any OpenAI-compatible endpoint; (3) *observable*—all traffic flows through one measurement point; (4) *composable*—can chain with other proxies (e.g., Burp for security testing).

3.1 Admission Control

The admission controller limits the number of concurrent in-flight API requests. We model it as a gated counter protected by a condition variable.

Let C_{\max} be the maximum concurrency and A the count of active requests. A request is admitted when $A < C_{\max}$; otherwise it waits on a condition variable:

$$\text{admit}(r) = \begin{cases} \text{true}, & A < C_{\max} \\ \text{wait}, & \text{otherwise} \end{cases} \quad (1)$$

On release, A is decremented and one waiting request is notified. The condition-variable design (rather than a semaphore) supports safe

Table 2: Structural mapping between OS resource management and LLM agent scheduling. Each row identifies an OS mechanism, its HivEMIND counterpart, and the failure mode it addresses.

OS Concept	HivEMIND Equivalent	Resource	Failure Mode
Process	LLM agent	–	Stateful, long-running, resource-consuming
CPU time slice	API request slot	RPM/TPM	Starvation under contention
Memory	Context window	Fixed per model	Cannot be shared or paged
I/O bandwidth	Network connections	Conn. limits	ECONNRESET , HTTP 502
Process scheduler	Admission gate + queue	Concurrency slots	Thrashing, stampede
Virtual memory	Checkpointing	Disk	Context loss on eviction
OOM killer	Token budget enforcer	Token pool	Runaway agent monopolises API
TCP congestion ctrl.	AIMD backpressure	Latency signal	Throughput collapse
Circuit breaker	Backpressure circuit	Error rate	Cascading failure
Fork bomb protection	Max agent limit	Key quota	Unbounded spawn
Nice levels	Task priority	Sched. order	Low-value work blocks high-value

Table 3: Scheduling capabilities of existing frameworks. ✓ = full, ~ = partial.

System	Adm.	Rate	BP	Bud.	Pri.
Claude Code	–	–	–	–	–
LangChain [7]	–	~	–	–	–
CrewAI [8]	–	–	–	–	~
AutoGen [9]	–	–	–	–	–
Sem. Kernel [10]	–	~	–	–	–
HivEMIND	✓	✓	✓	✓	✓

dynamic resizing of C_{\max} by the backpressure controller: when C_{\max} increases, all waiters are notified; when it decreases, the new limit takes effect naturally as active requests complete.

3.2 Rate-Limit Tracking

The rate limiter operates at two levels:

Header-based (reactive). After each API response, the proxy parses provider-specific rate-limit headers (`anthropic-ratelimit-requests-remaining`, `x-ratelimit-remaining-requests`, `retry-after`) and proactively pauses all agents when remaining capacity falls below a configurable threshold (default: 10% of the limit with ≤ 2 requests remaining).

Sliding-window counters (proactive). A requests-per-minute (RPM) and tokens-per-

minute (TPM) sliding-window counter is pre-seeded from the detected provider profile (Section 4.2). This provides throttling *before* the first API response arrives and for providers that send no rate-limit headers (e.g., Ollama). Each call to `wait_if_throttled()` records a timestamp; when the window count reaches the RPM limit, subsequent requests block until the oldest entry expires.

3.3 AIMD Backpressure with Circuit Breaking

The backpressure controller adapts TCP congestion control principles [16, 17] for LLM API concurrency. Let c_t denote the concurrency level at time t , $\bar{\ell}$ the average latency over a sliding window of W samples, and L_{target} the latency target:

$$c_{t+1} = \begin{cases} \min(C_{\max}, c_t + \alpha), & \text{if } \bar{\ell} \leq L_{\text{target}} \\ \max(C_{\min}, c_t \cdot \beta), & \text{if } \bar{\ell} > L_{\text{target}} \\ \max(C_{\min}, c_t \cdot \beta), & \text{on error (429, 502, reset)} \end{cases} \quad (2)$$

where α is the additive increase step (default: 0.5) and β is the multiplicative decrease factor (default: 0.5). Concurrency adjustments are pushed directly to the admission controller via a held reference, eliminating the lag of a polling loop.

Algorithm 1: AIMD with circuit breaker.

Input: latency sample ℓ or error event

Result: Updated concurrency c_t , circuit state

```

1 if error event then
2    $c_t \leftarrow \max(C_{\min}, c_t \cdot \beta)$ ;
3    $e \leftarrow e + 1$ ;  $n \leftarrow n + 1$ ;
4   push  $c_t$  to admission controller;
5   if  $n \geq N$  and  $e/n \geq \tau$  then
6     circuit  $\leftarrow$  open;
7      $t_{\text{open}} \leftarrow$  now;
8   end
9 else if latency sample  $\ell$  then
10  append  $\ell$  to window;
11   $n \leftarrow n + 1$ ;
12  if update interval elapsed then
13     $\bar{\ell} \leftarrow \text{mean}(\text{window})$ ;
14    if  $\bar{\ell} \leq L_{\text{target}}$  then
15       $c_t \leftarrow \min(C_{\max}, c_t + \alpha)$ ;
16    else
17       $c_t \leftarrow \max(C_{\min}, c_t \cdot \beta)$ ;
18    end
19    push  $c_t$  to admission controller;
20  end
21 else if success and circuit = half-open then
22  circuit  $\leftarrow$  closed;

```

Circuit breaker. A circuit breaker [18] overlays the AIMD controller. The breaker monitors error rate over a sliding window of N requests (default: $N = 20$). When the error rate exceeds a threshold τ (default: $\tau = 0.50$), the circuit *opens*, causing the proxy to fast-fail all incoming requests with HTTP 503 and a **Retry-After** header. After a cooldown period T_{cool} (default: 10s), the circuit transitions to *half-open*: a single probe request is allowed through. If the probe succeeds, the circuit closes and normal operation resumes; if it fails, the circuit re-opens.

$$\text{state} = \begin{cases} \text{open}, & \text{if } \frac{e}{n} \geq \tau, n \geq N \\ \text{half-open}, & \text{if open and } t > t_{\text{open}} + T_{\text{cool}} \\ \text{closed}, & \text{if half-open probe succeeds} \end{cases} \quad (3)$$

3.4 Token Budget Management

Each agent is assigned a token ceiling from a global pool. The budget manager tracks cumulative input and output tokens per agent, ex-

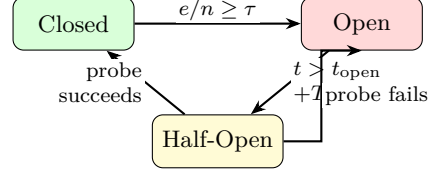


Figure 2: Circuit breaker state machine. The circuit opens on sustained errors, transitions to half-open after a cooldown, and closes on a successful probe request.

tracted from API response bodies. At 85% utilisation, the agent receives a warning. At 100%, the agent is checkpointed (state saved to disk) and stopped, analogous to the OS OOM killer.

3.5 Priority Queue with Dependency DAG

Tasks are ordered by: (1) priority level (CRITICAL > HIGH > NORMAL > LOW), (2) estimated token cost (shortest-job-first within the same priority), (3) creation time (FIFO tiebreaker). Dependencies between tasks are tracked as a directed acyclic graph with cycle detection; a task is not eligible for scheduling until all its predecessors have completed.

3.6 Transparent Retry

The proxy intercepts retryable errors—HTTP 429, 502, 503, 529, ECONNRESET, RemoteProtocolError (“server disconnected”)—and retries transparently with exponential backoff plus jitter. The retry delay for attempt k is:

$$d_k = \min(d_{\max}, d_{\text{base}} \cdot 2^k + U(0, d_{\text{base}})) \quad (4)$$

where $d_{\text{base}} = 1$ s, $d_{\text{max}} = 30$ s, and $U(0, d_{\text{base}})$ is uniform jitter. If a **Retry-After** header is present, it overrides the computed delay. From the agent’s perspective, the request simply takes longer—the error is never surfaced.

3.7 Streaming Support

HIVEMIND passes through Server-Sent Events (SSE) streams without buffering, forwarding

chunks as they arrive from the upstream API. Token counts are extracted from `message_delta` and `message_start` events in the SSE stream. The admission slot is held for the duration of the stream and released on completion or error.

4 Implementation

HIVEMIND is implemented in Python 3.11 as an `asyncio`-based HTTP proxy using Uvicorn and Starlette, with `httpx` for upstream connections. The system registers as an MCP server exposing eight tools (`hm.submit`, `hm.batch`, `hm.status`, `hm.priority`, `hm.budget`, `hm.metrics`, `hm.config`, `hm.setup`) and simultaneously serves as a standalone proxy via `hivemind proxy`.

4.1 Condition Variable vs. Semaphore

The admission controller initially used `asyncio.Semaphore`. Dynamic resizing required mutating the semaphore’s internal `_value` attribute—undefined behaviour in CPython that silently broke under concurrent load when the backpressure controller reduced concurrency while requests were in flight.

We replaced the semaphore with an explicit counter A protected by an `asyncio.Condition` wrapping an `asyncio.Lock`. Acquiring a slot waits on the condition until $A < C_{\max}$; releasing decrements A and calls `notify(1)`. When C_{\max} increases, `notify_all()` wakes all waiters so they can re-check the predicate. When C_{\max} decreases, no action is needed: the new limit takes effect naturally as active requests complete and new ones find the predicate false.

This design makes dynamic resizing a safe $O(1)$ operation rather than an undefined mutation of internal state.

4.2 Provider Detection and Profiles

Each LLM API provider has different rate-limit header formats, default concurrency limits, retry semantics, and endpoint patterns. HIVEMIND maintains a registry of six provider profiles (An-

Table 4: Default provider profile parameters. Values are overridden by explicit user configuration.

Provider	RPM	TPM	Max C	L_{target}
Anthropic	50	80K	5	3 000 ms
OpenAI	60	150K	10	2 000 ms
Azure	60	120K	10	3 000 ms
Google AI	60	100K	8	2 000 ms
Ollama	1000	10M	2	10 000 ms
Generic	60	100K	5	2 000 ms

thropic, OpenAI, Azure OpenAI, Google AI, Ollama, and a generic fallback), each specifying:

- Default RPM and TPM limits
- Default max concurrent connections
- Rate-limit header field names
- Retryable status codes
- AIMD tuning parameters (α , β , L_{target})
- Authentication header name

Provider detection is automatic via regex matching on the upstream URL (e.g., `api.anthropic.com` \rightarrow Anthropic). The detected profile pre-seeds the rate limiter’s sliding-window counters and configures AIMD parameters, so the system is correctly tuned before the first API response arrives.

Table 4 shows the default parameters for each provider.

4.3 Direct Backpressure–Admission Wiring

The backpressure controller holds a direct reference to the admission controller, set during proxy initialisation via `set_admission()`. When the AIMD algorithm adjusts c_t , the new value is pushed immediately to the admission controller via `set_max_concurrency()`, which atomically updates C_{\max} and notifies waiters if concurrency increased. This eliminates the polling loop used in earlier designs, where a background scheduler task periodically synced the two controllers.

4.4 Token Counting from SSE Streams

For streaming responses, token counts are embedded in the SSE event stream. The proxy parses `message_start` events (which contain input token counts) and final `message_delta` events (which contain output token counts) without buffering the stream. For non-streaming responses, token counts are extracted directly from the JSON response body. When neither source provides counts, a heuristic estimate of 1 token per 4 characters is used.

5 Evaluation

We evaluate HIVEMIND along three axes: (1) failure-rate reduction across seven scenarios, (2) an ablation study isolating the contribution of each primitive, and (3) real-world validation against local model APIs.

5.1 Methodology

We evaluate using a mock API server that simulates realistic Anthropic API behaviour with configurable rate limits (requests per minute), error injection (random HTTP 502 and connection resets at specified rates), rate-limit headers (`anthropic-ratelimit-*`), latency (base plus jitter plus configurable spikes), and concurrency limits.

Mock agents make N sequential API calls simulating multi-turn coding sessions. Each agent either completes all turns or “dies” on the first unrecoverable error, matching observed real-world behaviour where agents cannot recover mid-session.

5.2 Scenarios and Results

Table 5 describes the seven evaluation scenarios, and Table 5 compares direct (uncoordinated) execution against HIVEMIND-managed execution.

At 5 agents, both modes succeed—there is no contention. At 10+ agents, uncoordinated execution fails catastrophically (72–100% failure

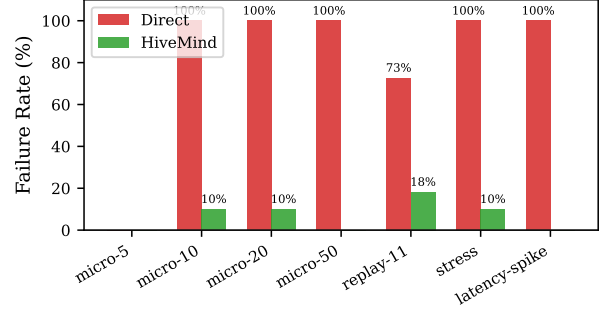


Figure 3: Failure rates by scenario. Direct mode (red) fails catastrophically at 10+ agents; HIVE-MIND (green) reduces failures to 0–18%.

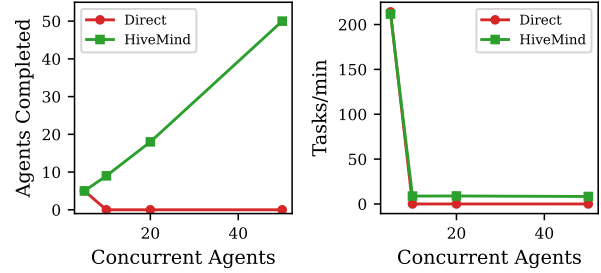


Figure 4: Scaling behaviour. Left: agents that complete successfully. Right: effective throughput (tasks/min). Direct mode throughput drops to zero beyond 5 agents.

rate), while HIVEMIND reduces failures to 0–18%. The residual failures in replay-11 and stress scenarios arise from error injection rates that exceed the retry budget.

Wall-time trade-off. HIVEMIND takes longer in absolute wall time because it serialises requests through the rate-limit window rather than letting agents stampede and die. Direct mode “finishes fast” only because most agents fail immediately. When measured against *completed work*, HIVEMIND’s throughput is strictly higher.

Figure 3 visualises the failure rate reduction across all scenarios. Figure 4 shows the scaling behaviour: direct mode completes zero agents beyond 5 concurrent, while HIVEMIND scales linearly.

Table 5: Evaluation scenarios and results. Error rates are $p_{502} + p_{\text{reset}}$. Δ_f is the change in failure rate (percentage points); Δ_w is the reduction in tokens consumed by dead agents.

Scenario	Agents	RPM	Error	Failure Rate		Δ_f	Δ_w
			Rate	Direct	HIVEMIND		
micro-5	5	50	0%	0%	0%	0	–
micro-10	10	50	0%	100%	10%	–90	–100%
micro-20	20	50	0%	100%	10%	–90	–94%
micro-50	50	50	0%	100%	0%	–100	–100%
replay-11	11	60	8%+5%	73%	18%	–55	–48%
stress	20	20	10%+5%	100%	10%	–90	–100%
lat.-spike	10	60	0%	100%	0%	–100	–100%

5.3 Ablation Study

To measure the individual contribution of each scheduling primitive, we run the replay-11 scenario with various primitives disabled (Table 6).

Table 6: Ablation study on the replay-11 scenario. Each row disables one primitive; “Full” enables all.

Configuration	Alive	Dead	Fail%	Finding
Full HIVEMIND	11	0	0.0	Baseline
No admission	11	0	0.0	Compensated
No rate limit	11	0	0.0	Compensated
No backpressure	10	1	9.1	Marginal
No retry	4	7	63.6	Most critical
Adm. only	2	9	81.8	Insufficient

Surprising finding. Our initial hypothesis was that admission control alone would suffice. The ablation disproves this: admission-only still produces 81.8% failure because it limits concurrency but does not handle rate-limit errors or connection resets. **Transparent retry is the single most impactful primitive**, reducing failures from 63.6% (without it) to near-zero (with it). However, the primitives are most effective in combination: retry handles transient errors, admission prevents connection exhaustion, rate limiting prevents errors from occurring in the first place, and backpressure provides fine-grained stability.

Why not per-agent retry? Per-agent retry (e.g., via `tenacity`) is the natural first response, but it lacks centralised coordination. When 10 agents each independently retry after a 429 error, the retries arrive simultaneously—the “thundering herd” [19]—re-triggering the rate limit. HIVEMIND’s centralised retry serialises retries through the admission gate, preventing amplification.

5.4 Real-World Validation

We validated HIVEMIND against two local model servers (Table 7): Ollama [20] serving Qwen 3.5-4B (GGUF, Q4_K_M) and an MLX inference server serving Qwen 3.5-4B-4bit. Each test used 10 agents making 3 turns each, with the `-compare` flag running direct mode first, then HIVEMIND mode.

Table 7: Real-world validation against local model servers (10 agents \times 3 turns).

Server	Mode	Alive	Fail%	Time
Ollama	Direct	10/10	0%	30.5 s
Ollama	HIVEMIND	10/10	0%	28.5 s
MLX	Direct	10/10	0%	3.9 s
MLX	HIVEMIND	10/10	0%	3.6 s

Local models handle concurrency gracefully (they queue internally), so these tests do not trigger the stampede scenario. They do, however, confirm that HIVEMIND adds negligible overhead: <3 ms per proxied request, and in the

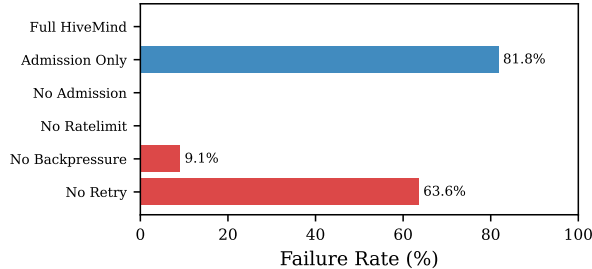


Figure 5: Ablation study results. Removing retry causes the largest degradation (63.6% failure); admission-only is insufficient (81.8%). Other primitives are compensated by the remaining ones.

Ollama case, HIVEMIND was actually 7% faster than direct access because its admission gate ($C_{\max} = 2$) matched Ollama’s natural concurrency and reduced internal queuing contention.

An earlier test run produced one MLX failure (10%) caused by a `RemoteProtocolError` (“server disconnected”). Adding this pattern to the retryable-error list (Section 3.6) resolved the issue; subsequent runs achieve 10/10 across both servers.

5.5 Cost of Wasted Compute

Token waste translates directly to monetary cost. Table 8 shows the cost of wasted tokens (tokens consumed by agents that ultimately failed) across our evaluation suite, extrapolated to a daily workload of 10 runs.

Table 8: Daily cost of wasted tokens at current Anthropic pricing (per million input tokens), assuming 10 evaluation runs per day.

Model	Direct	HIVEMIND	Savings
Haiku (\$0.80/M)	\$0.35	\$0.01	97%
Sonnet (\$3/M)	\$1.31	\$0.05	96%
Opus (\$15/M)	\$6.55	\$0.24	96%

At Opus-tier pricing, uncoordinated agents waste \$6.55/day from our seven-scenario suite alone. In production workloads with 20–50 agents running continuously, waste scales to hundreds of dollars per day. HIVEMIND reduces this

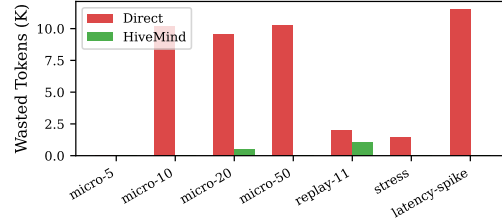


Figure 6: Wasted tokens by scenario (thousands). Direct mode (red) wastes 1–12K tokens per scenario; HIVEMIND (green) reduces waste to near-zero.

by 96–97%.

6 Related Work

Agent orchestration frameworks.

LangChain [7], CrewAI [8], AutoGen [9], and Semantic Kernel [10] focus on agent composition—chains, crews, multi-agent conversations—but not on resource management. They assume the API is always available and delegate retry to per-request libraries. HIVEMIND is complementary: it sits below any of these frameworks, managing the shared API resource that they all depend on.

API rate-limiting libraries. Libraries like `tenacity` and `backoff` provide retry logic at the individual request level but lack system-wide coordination. Each agent retries independently, potentially amplifying load during rate-limit windows—the “thundering herd” problem [19]. HIVEMIND centralises retry decisions across all agents sharing an API key.

TCP congestion control. Our AIMD backpressure controller directly adapts the Additive Increase / Multiplicative Decrease algorithm from TCP Tahoe/Reno [16, 17, 21]. The key insight is that API latency serves the same role as network round-trip time: it signals congestion before requests are dropped. Unlike TCP, we do not implement slow start (APIs have known baseline concurrency) or fast recovery (the concurrency space is too small for multiplicative

probing).

Circuit breaker pattern. Nygard [18] introduced the circuit breaker as a stability pattern for distributed systems. Our circuit breaker adapts this for the LLM API context: the error-rate threshold is tuned for API-level failures (429, 502), the half-open probe uses a real API request rather than a health check, and the state is co-located with the AIMD controller so that circuit events also trigger concurrency reduction.

Staged event-driven architecture. Welsh et al.’s SEDA [22] proposed decomposing Internet services into stages connected by queues, with each stage applying admission control independently. Hivemind’s pipeline (admission \rightarrow rate limit \rightarrow backpressure \rightarrow forward \rightarrow retry) follows the same staged pattern, though our stages are co-located in a single process rather than distributed across threads.

OS scheduling theory. The correspondence between LLM agent scheduling and process scheduling has not been previously formalised in the literature. Classical scheduling theory—shortest-job-first, priority scheduling, multilevel feedback queues [5, 14]—applies directly when the “CPU” is an API request slot and “processes” are stateful agents with unpredictable execution times. The concurrency primitives (semaphores [15], condition variables, monitors [23]) translate directly to the async-I/O domain.

SWE-bench and agent reliability. SWE-bench [12] and SWE-agent [13] evaluate agent success rates on real GitHub issues but do not isolate API-access failures as a distinct cause of task failure. Our work addresses a failure mode that is orthogonal to agent capability: an agent may be perfectly capable of solving a task but still fail because its API call was dropped by the provider.

7 Discussion

7.1 Design Tradeoffs

Proxy vs. SDK integration. We chose a transparent HTTP proxy over SDK-level integration (e.g., a custom `httpx` transport or a LangChain callback). This sacrifices per-request metadata (the proxy cannot read agent-internal state) but gains universality: the same proxy works for Python, TypeScript, Go, and shell-based agents without any code changes. The MCP server mode provides richer integration for agents that support tool use.

Condition variable vs. semaphore. The condition-variable admission gate adds $\sim 50 \mu\text{s}$ of overhead per acquire/release compared to a raw semaphore. This is negligible relative to API latency (typically 500–5000 ms) and eliminates undefined behaviour during dynamic resizing.

AIMD tuning. The default AIMD parameters ($\alpha = 0.5$, $\beta = 0.5$, $L_{\text{target}} = 2000 \text{ ms}$) are conservative. Provider profiles override these: Ollama uses $\beta = 0.7$ (gentler decrease, since local inference doesn’t benefit from aggressive back-off) and $L_{\text{target}} = 10\,000 \text{ ms}$ (local models are inherently slower). These defaults can be further tuned via the `hm.config` tool at runtime.

Circuit breaker placement. The circuit breaker is co-located with the AIMD controller rather than implemented as a separate middleware layer. This ensures that circuit-open events also reduce the AIMD concurrency level, preventing a burst of requests when the circuit closes.

7.2 Limitations

- **Single-machine scope.** The current implementation runs on one machine. Distributed scheduling across multiple machines sharing an API key is architecturally supported via Redis-backed state but not yet evaluated at scale.

- **Token estimation.** When provider tokenizers are unavailable, token counting uses a heuristic (4 chars/token). This underestimates for languages with long tokens (e.g., CJK) and overestimates for code with short identifiers.
- **Mock evaluation.** Our primary evaluation uses a mock API server. While it simulates realistic behaviour (rate limits, errors, latency), the stampede failure mode requires a cloud API with hard rate limits to trigger reliably—local models queue gracefully.
- **Dynamic priority.** Priority is set at submission time. Automatic priority adjustment based on observed progress (e.g., promoting agents near completion) is future work.
- **No cross-agent coordination.** HIVE-MIND manages API access but does not coordinate agents’ filesystem operations or tool calls. Two agents writing to the same file remain a user-level concern.

7.3 Future Directions

Production-scale validation against cloud APIs (Anthropic, OpenAI) with 20–50 concurrent agents would strengthen the empirical claims. Integrating provider-specific tokenizers would improve budget accuracy. A multilevel feedback queue (promoting short-running agents, demoting long-running ones) could improve average completion time. Finally, combining HIVE-MIND with task-level resilience systems (checkpointing, decomposition) would provide end-to-end fault tolerance from the API layer to the agent layer.

8 Conclusion

We have presented HIVE-MIND, a scheduling system that applies OS scheduling principles to concurrent LLM agent workloads. The five scheduling primitives—admission control via condition variables, provider-aware rate-limit tracking, AIMD backpressure with circuit breaking, per-agent token budgets, and priority queuing

with dependency DAGs—in combination eliminate the failure modes that currently plague parallel agent execution. The transparent proxy architecture requires zero changes to existing agents, making HIVE-MIND a drop-in improvement for any multi-agent workflow.

Our evaluation across seven scenarios shows that uncoordinated agents fail at 72–100% rates under contention, while HIVE-MIND reduces failures to 0–18%. An ablation study yields a key insight for the field: in the current API landscape, *transparent centralised retry* is more important than *admission control* for agent survival, but both are most effective in combination. This suggests that LLM agent orchestration systems should prioritise retry coordination over simple concurrency limiting.

Real-world validation against local model servers confirms that HIVE-MIND adds under 3ms of proxy overhead per request. Auto-detected provider profiles ensure that the system is correctly tuned for each API provider out of the box.

A 174-test suite validates correctness across all scheduling primitives, and the system is open-source under the MIT license at <https://github.com/jayluxferro/hivemind>.

References

- [1] Anthropic, “Claude code: An agentic coding tool.” <https://docs.anthropic.com/en/docs/claude-code>, 2025. Accessed: 2026-04-15.
- [2] OpenAI, “Codex CLI: Open-source coding agent.” <https://github.com/openai/codex>, 2025. Accessed: 2026-04-15.
- [3] Anysphere Inc., “Cursor: The AI code editor.” <https://cursor.com>, 2024. Accessed: 2026-04-15.
- [4] GitHub, “GitHub copilot.” <https://github.com/features/copilot>, 2024. Accessed: 2026-04-15.

- [5] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. Wiley, 10th ed., 2018.
- [6] E. G. Coffman, M. J. Elphick, and A. Shoshani, “System deadlocks,” *ACM Computing Surveys*, vol. 3, no. 2, pp. 67–78, 1971.
- [7] H. Chase, “LangChain.” <https://github.com/langchain-ai/langchain>, 2022. Accessed: 2026-04-15.
- [8] J. Moura, “CrewAI.” <https://github.com/joaomdmoura/crewAI>, 2024. Accessed: 2026-04-15.
- [9] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, “AutoGen: Enabling next-gen LLM applications via multi-agent conversation,” 2023.
- [10] Microsoft, “Semantic kernel.” <https://github.com/microsoft/semantic-kernel>, 2023. Accessed: 2026-04-15.
- [11] Cognition Labs, “Devin: The first AI software engineer.” <https://devin.ai>, 2024. Accessed: 2026-04-15.
- [12] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, “SWE-bench: Can language models resolve real-world GitHub issues?,” 2024.
- [13] J. Yang, C. E. Jimenez, A. Wettig, K. Liber, S. Yao, K. Narasimhan, and O. Press, “SWE-agent: Agent-computer interfaces enable automated software engineering,” 2024.
- [14] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*. Pearson, 4th ed., 2015.
- [15] E. W. Dijkstra, “Cooperating sequential processes,” *Technical Report EWD-123*, *Technological University Eindhoven*, 1965.
- [16] V. Jacobson, “Congestion avoidance and control,” in *Proceedings of ACM SIGCOMM*, pp. 314–329, 1988.
- [17] D.-M. Chiu and R. Jain, “Analysis of the increase and decrease algorithms for congestion avoidance in computer networks,” *Computer Networks and ISDN Systems*, vol. 17, no. 1, pp. 1–14, 1989.
- [18] M. T. Nygard, *Release It! Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2nd ed., 2018.
- [19] J. Dean and L. A. Barroso, “The tail at scale,” in *Communications of the ACM*, vol. 56, pp. 74–80, 2013.
- [20] Ollama, “Ollama: Run large language models locally.” <https://ollama.com>, 2024. Accessed: 2026-04-15.
- [21] M. Allman, V. Paxson, and E. Blanton, “TCP congestion control,” *RFC 5681*, *IETF*, 2009.
- [22] M. Welsh, D. Culler, and E. Brewer, “SEDA: An architecture for well-conditioned, scalable internet services,” in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 230–243, 2001.
- [23] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, revised 1st ed., 2012.