IntelliRoute: A Distributed Control Plane for Multi-LLM Orchestration

Anukrithi Myadala, Larry Nguyen, James Pham, Surbhi Singh
Department of Computer Engineering
San José State University
CMPE 273 — Enterprise Distributed Systems, Spring 2026

***Abstract*—Most teams that ship features on top of large language models (LLMs) end up calling a provider SDK directly from application code, wrapped in a try/except that retries the same provider once and then gives up. As soon as a second provider is added, this pattern collapses: routing logic gets sprinkled across services, quotas are fragmented per provider, cost arrives in a monthly bill that nobody can attribute, and a single misrouted batch can bury an interactive chat in queue depth. We argue that LLMs should not be seen as an API, but instead as a distributed system. Therefore, they need a control plane that treats them as such. *IntelliRoute* is our attempt at that control plane. It is a six-service Python/FastAPI system that sits between clients and a pool of mock and real providers (Groq, Gemini), and combines (i) deterministic intent classification, (ii) a multi-objective ranker that weighs latency, cost, capability, and historical success per intent, (iii) a three-replica token-bucket rate limiter with bully-style leader election and a replication log, (iv) per-provider three-state circuit breakers with a fallback chain, (v) a brownout manager with a priority queue and load shedding, and (vi) asynchronous tenant/team/workflow cost accounting with pre-call budget gates. The full stack is approximately 8.7k lines of Python and ships with 259 tests (including 13 end-to-end integration tests that spawn the full ten-process backend on ephemeral ports) plus a replay evaluation harness that compares five routing policies under four scenarios.**

I. Introduction

The 3 a.m. on-call story is a familiar one. A user-facing AI feature paged because its primary provider started rate-limiting, the secondary provider was degraded, and the application's fallback silently routed every request to the cheapest model nobody had benchmarked. By morning, the dashboard reads $4,800 of spending on a single tenant and a broken SLA, while nobody could answer the most basic question: *why did it route there?*

That story is not really about LLMs, but more about a control plane that does not exist. When a service mesh, a load balancer, or a database cluster degrades, on-call engineers have machinery to reach for: circuit breakers, leader election, quotas, brownout, and traceable decisions. LLMs today are usually treated as an HTTP endpoint with a key, and the surrounding infrastructure is whatever the application author had time to write that sprint.

IntelliRoute is a one-semester project that takes the position that LLMs deserve the same infrastructure treatment as any other shared compute resource. Concretely, it answers four questions:

1) *Where should this request go?* An intent classifier and a multi-objective ranker turn a free-form prompt into a ranked list of providers per intent.
2) *Are we allowed to send it?* A distributed token-bucket cluster with a single elected leader serves authoritative quota decisions.
3) *What if it fails?* A three-state per-provider circuit breaker, an adaptive fallback chain, and a brownout manager degrade the system gracefully instead of pushing failures back to the user.
4) *Who paid for it?* An asynchronous cost tracker rolls up per-tenant, per-team, and per-workflow spend, with a pre-call budget gate that demotes to a cheaper provider when the projected cost would push past any active budget.

The remainder of the paper is organized in the structure recommended for the course: problem definition (Section II), existing approaches (Section III), our design (Section IV), implementation notes (Section V), evaluation (Section VI), and limitations (Section VII).

II. Problem Definition

Calling an LLM looks easy, it is one HTTP `POST`. We identified five recurring problems that hide inside that POST and get worse the more an organization comes to depend on it.

P1: Unpredictable latency. The same prompt against the same model can take 80 ms or 4 s depending on the provider's queue, geographical routing, and current load. Application timeouts written against a happy-path measurement are wrong half the time.

P2: Fragmented quotas. Each provider publishes its own counters in its own units (requests per minute, tokens per minute, concurrent connections), and the application has no global view of "this tenant has spent X across all providers in the last minute."

P3: Opaque cost. Cost is a function of tokens, model tier, tenant, team, and workflow, and the bill arrives at the end of the month. By the time a regression is visible it has been running for weeks.

P4: No control plane. Routing logic, which provider, what budget, how many retries, what happens on 429, gets sprinkled across application services. Any change requires touching

TABLE I
DISTRIBUTED-SYSTEMS CONCEPTS AND THEIR LOCATION IN THE INTELLIROUTE SOURCE TREE.

Concept	Module
Service discovery + leases	`router/registry.py`
Leader election (bully)	`rate_limiter/election.py`
Replication log	`rate_limiter/token_bucket.py`
Circuit breaker (3-state)	`health.../circuit_breaker.py`
Adaptive retry / SLA-aware back-off	`router/main.py`
Backpressure + priority queue	`router/queue.py`
Brownout / load shedding	`router/brownout.py`
Multi-objective ranking	`router/policy.py`
Online weight tuning	`router/weight_tuner.py`
Tenant/team/workflow budgets	`cost_tracker/accounting.py`

multiple codebases and there is no single place to ask "why did this request go to provider X?"

P5: No system-level optimization. An interactive chat request and a nightly batch summarization job hit the same quota. Without priority awareness, a long-running batch can bury an interactive request behind it.

These five problems are exactly the kind of thing distributed systems already know how to fix; they are just not currently fixed for LLM traffic. That gap is what IntelliRoute targets.

III. EXISTING APPROACHES

The state of the practice today falls roughly into four camps. We list each and note where it falls short for the problems above.

Direct provider SDK. The application imports the OpenAI or Anthropic SDK and calls it. Quotas, retries, and cost accounting are the application's problem. This solves nothing on the list above.

Per-application `try/except`. A second provider is added behind a hand-written fallback. This addresses the trivial case of P4 (one fallback hop) but does not help with quota visibility, cost, brownout, or routing decisions across more than two providers.

Provider-side rate limiting. Each provider enforces its own limits on its own token-bucket. From the application's perspective this is just a 429 to be retried; it does not give a unified per-tenant view (P2) and offers no admission control across providers.

Naive load balancers. Round-robin, latency-only, or cheapest-first policies. These can be implemented in a hundred lines of Python, but they ignore intent (P5) and have no graceful behavior under partial failure (P4).

There is also a small but growing class of commercial LLM gateways (OpenRouter, Portkey, LangChain's router, etc.). The contribution of IntelliRoute is not to replace those products but to demonstrate, in a course-scoped artifact, the underlying *distributed-systems* machinery they rely on: leader election, replication logs, three-state breakers, brownout, priority queueing, and per-data-path consistency. The full list of mappings between course concepts and the modules that implement them is summarized in Table I.

IV. SYSTEM DESIGN

A. Service Topology

IntelliRoute is six logical services plus a pool of providers and a static dashboard. Figure 1 shows the full topology. Every service is a FastAPI application; every inter-service call is HTTP. Synchronous calls are used where the routing decision cannot proceed without an authoritative answer (rate-limit checks, health state); asynchronous fire-and-forget calls are used where staleness on the order of seconds is acceptable (cost events, success/failure reports).

B. Intent Classification

The router begins by classifying the request into one of four intents: `INTERACTIVE`, `REASONING`, `BATCH`, or `CODE`. We picked these four because they map cleanly to the weight vectors we wanted to express in the ranker (latency-dominated chat, capability-dominated reasoning, cost-dominated batch, capability-and-latency code). The classifier itself is a deterministic keyword/heuristic in `router/intent.py`, it scans the joined message text for tokens like `def`, `import`, `Traceback`, `explain step by step`, `summarize the following document`, etc. We deliberately avoided using a learned model here because the distributed-systems contributions of the project are not in classification, and a deterministic classifier unit-tests cleanly without an external dependency. The classifier also honors an optional client-supplied `intent_hint` so applications that already know their workload can skip the heuristic.

C. Multi-Objective Routing

Once an intent is known, the router asks the policy engine (`router/policy_engine/`) to apply pre-rank rules, batch avoids premium providers, premium requires reasoning or a high complexity score, tenant/team/workflow budget pressure can demote premium-tier providers, brownout can block premium and prefer low-latency providers, and an interactive latency gate filters out providers whose declared p95 SLA exceeds the request's latency budget. Each rule contributes audit metadata that is returned in a `PolicyEvaluationResult` on the response, so we always know *why* a provider was filtered.

The surviving providers go into the multi-objective ranker (`router/policy.py`). For each provider we compute four sub-scores in $[0, 1]$, latency, cost, capability, success rate, and combine them using an intent-specific weight vector. The final score is

$$S_p = w_L \cdot \tilde{L}_p + w_C \cdot \tilde{C}_p + w_K \cdot K_p + w_R \cdot R_p, \qquad (1)$$

where $\tilde{L}_p$ and $\tilde{C}_p$ are min-max normalised latency/cost values (lower is better, hence *higher score* better), K_p is the provider's declared capability tier, and R_p is its EMA success rate from the feedback collector. Table II shows the per-intent weight vectors we shipped as defaults; the online `WeightTuner` in `router/weight_tuner.py` updates them at runtime as it accumulates per-attempt outcomes.

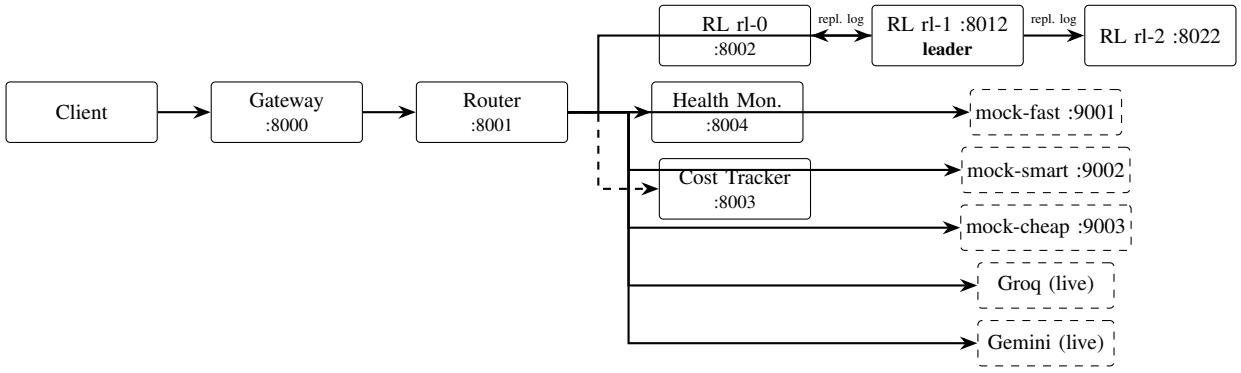

Fig. 1. IntelliRoute service topology. Solid arrows are synchronous HTTP; dashed arrows are asynchronous fire-and-forget. Three rate-limiter replicas form a single bully-elected cluster; the leader serves all `/check` calls authoritatively and fans out a replication log to the followers.

TABLE II
DEFAULT SUB-SCORE WEIGHTS PER INTENT. THE ONLINE WEIGHT TUNER CAN REBALANCE THESE FROM OBSERVED OUTCOMES.

Intent	w_L	w_C	w_K	w_R
Interactive	0.55	0.15	0.20	0.10
Reasoning	0.10	0.10	0.50	0.30
Batch	0.05	0.65	0.15	0.15
Code	0.10	0.05	0.65	0.20

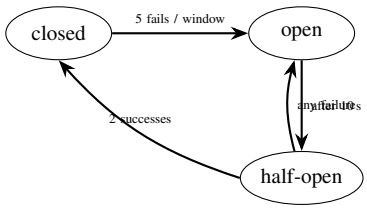

Fig. 2. Per-provider circuit breaker state machine.

D. Distributed Rate Limiting

The rate limiter is the part of the system where consistency matters most. If we over-issue a token the application either incurs unbudgeted spend or hits a 429 storm; either failure is hard to unwind. We therefore run three replicas (`:8002`, `:8012`, `:8022`) with one leader. The leader owns the `TokenBucket` state and serves all `/check` calls. Followers forward writes to the leader and tail its replication log; if the leader cannot be reached, followers fall back to local state by default, or fail closed when `RATE_LIMITER_STRONG_CONSISTENCY=1` is set. Each bucket is keyed by `(tenant_id, provider)` and resolves through a layered configuration: exact pair, per-tenant any-provider, per-provider any-tenant, then a global default.

Leader election is a bully-style protocol in `rate_limiter/election.py`. Replicas exchange `/election/challenge`, `/election/victory`, and `/election/heartbeat` messages, and the highest replica id always wins. We picked bully over Raft for the same reason we picked heuristics over a learned classifier: the goal of the project was to make the consistency tradeoff visible and demonstrable, not to ship a formally verified consensus implementation. Section VII discusses the implications of this choice.

E. Circuit Breakers and the Fallback Chain

The health monitor (`:8004`) runs a per-provider three-state breaker (`closed` → `open` → `half_open`) with a sliding error window of 20 outcomes, a default failure threshold of 5, and a 10-second open-duration cooldown. Figure 2 sketches the state machine. On every provider call, the router pushes a success/failure signal back to the monitor via `/report/provider`. The monitor advances `open` → `half_open` *lazily* when `/snapshot` is read, this avoids a background thread per provider and keeps the breaker side-effect-free between requests.

When the breaker for a candidate is open, the router skips it and moves to the next provider in the ranked list. Retries against the same provider use a jittered exponential back-off bounded by the provider's declared p95 SLA, we did not want a slow provider's back-off to consume the whole interactive latency budget. If *every* candidate trips out (open breakers, exhausted daily quotas, denied by the rate limiter), the router returns a structured HTTP 503 to the client rather than hanging.

F. Brownout, Backpressure, and Priority Queueing

Under sustained overload the system enters *brownout* (`router/brownout.py`). Four signals are tracked in a sliding window: queue depth, p95 latency, error rate, and timeout rate. The manager enters brownout when any of the four exceeds an enter threshold for `enter_consecutive` evaluations in a row, and leaves only when all four sit below their exit threshold for `exit_consecutive` consecutive evaluations. We use this hysteresis to avoid flapping between `normal` and `degraded` on a single noisy sample.

While degraded, the system: (i) drops or delays `LOW`-priority requests, (ii) blocks premium-tier providers for `MEDIUM`/`LOW`, (iii) prefers low-latency providers even at higher cost, and (iv) clamps `max_tokens` for `REASONING`/`BATCH`. The manager runs both globally and per-tenant, so one noisy tenant cannot drag everyone else into brownout.

The priority queue (`router/queue.py`) is the input side of this. `INTERACTIVE` and `CODE` are `HIGH` and bypass the queue entirely; `REASONING` is `MEDIUM`; `BATCH` is `LOW`.

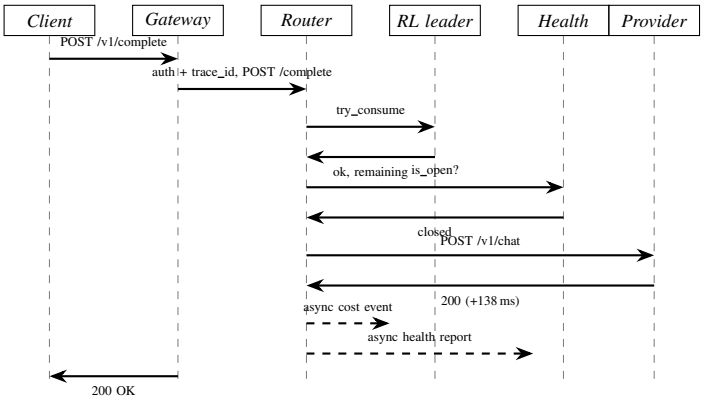

Fig. 3. Happy-path request flow for `POST /v1/complete`. Solid arrows block the routing decision; dashed arrows are fire-and-forget.

Above `shed_threshold`, new `LOW`-priority arrivals are shed instead of enqueued. Four async worker tasks drain the queue.

G. Cost Accounting and Budget Gates

Cost events are published asynchronously from the router to the cost tracker (`:8003`); the router does not wait for the publish to succeed before returning the response. The accountant maintains rollups at three scopes, tenant, team, workflow, and exposes a pre-call gate `/budget/{tenant_id}/check?projected_cost_usd=...`. The router's `/complete` pipeline calls the gate before sending the request, and demotes the head candidate to the cheapest still-pending provider if the projected cost would push past any active budget. This decision is logged with the reason `budget_gate_demoted` in the trace ring buffer.

V. Implementation Notes

The full system is approximately 8.7k lines of Python across the `intelliroute/` package, plus the test suite. It runs on Python 3.10+ with FastAPI and Uvicorn, talks HTTP between services with `httpx`, and uses `pydantic` models as the wire contract. There are no databases, no compiled extensions, and no Docker requirements, a fresh clone runs end-to-end with `pip install -e ".[dev]"` followed by `python3 scripts/start_stack.py`.

A. Trace propagation and observability

The gateway generates an `X-Request-Id` on every request and propagates it to all downstream services. Every log line is JSON (`intelliroute/common/logging.py`) and tagged with the request id, so a single `jq` pipeline reconstructs the full path of a request across services. The router additionally maintains a 50-entry ring buffer of recent traces at `/traces` that the dashboard polls. Figure 3 shows the synchronous and asynchronous calls a single happy-path `/v1/complete` fans out into.

TABLE III
Consistency model per data path.

Data path	Model	Why
Rate-limit decisions	Strong	Tokens are money; over-issue is hard to unwind.
Replication log	Eventual	Followers tail and replay so they can take over as leader.
Cost rollups	Eventual	Async events; small lag is fine.
Provider health	Per-instance	Each health monitor maintains its own breaker state.
Service registry	Per-router	In-process dict + lease TTLs; no cross-router sync.
Brownout state	Per-router	Each router observes its own queue/p95/error window.

B. Consistency model per data path

The consistency model is summarized in Table III. Rate-limit checks are the only data path where we paid the cost of strong consistency, because over-issuing tokens is the failure mode that is hardest to undo. Cost rollups are the dual case, a few seconds of staleness on a dashboard is fine, and an asynchronous publish path lets the router stay on the request critical path.

VI. Evaluation

We evaluated IntelliRoute along three axes: (i) a unit and integration test suite, (ii) a deterministic replay harness that compares routing policies under named scenarios, and (iii) a set of live failure-mode demos.

A. Test suite

The repository ships with 259 tests across 28 `test_*.py` modules. 246 are unit tests against the pure components (token bucket, breaker, election, intent classifier, ranker, accountant, queue, brownout, weight tuner, registry, user feedback store). The remaining 13 are end-to-end integration tests in `tests/test_integration.py` that spawn the full ten-process backend on ephemeral ports and exercise it over real HTTP. The full suite runs in roughly 12 seconds on a laptop, which is the single fastest way we have found to confirm a fresh clone is healthy.

B. Replay harness

The replay harness (`intelliroute/eval_harness/` + `scripts/replay_eval.py`, documented in `docs/REPLAY_EVAL_HARNESS.md`) generates deterministic JSONL workloads for four named scenarios and replays them against `/v1/complete` while sweeping the router through five routing policies. Before each `(scenario, policy)` run it resets all mutable service state (router runtime, cost rollups, breakers, all three rate-limiter replicas, mock fault flags) so cross-run comparisons are fair. Per run it writes a `summary.json`, `metrics.csv`, `config.json`, `reset_report.json`, `timeline.csv`, and `run.log` under `artifacts/replay_runs/`. The matrix run additionally writes an `aggregate_summary.json`,

TABLE IV
REPLAY SCENARIOS.

Scenario	What it stresses
`normal_mixed`	Healthy baseline; mixed interactive/reasoning/-batch load.
`degraded_provider`	One provider degraded after reset; tests the failover and reroute paths.
`budget_pressure`	Constrained tenant/team/workflow budgets force the policy to demote without full collapse.
`overload_brownout`	Burst workload at high concurrency to trip the brownout manager.

`run_index.json`, and a human-readable `report.md`. The metrics computed per `(scenario, policy)` are total requests, success rate, error rate, average / median / p50 / p95 / p99 latency, latency standard deviation, total cost, average cost per request, premium usage rate, fallback / reroute / reject counts, and provider distribution. Table IV summarizes the four scenarios.

C. Live failure-mode demos

Each mock provider exposes four fault-injection hooks (`/admin/force_fail`, `/admin/force_timeout`, `/admin/force_rate_limit`, `/admin/force_malformed`) plus a `/admin/reset` that clears them. The demo scripts (`scripts/demo.py`, `scripts/demo_failure_recovery.py`) walk a breaker through the full `closed` → `open` → `half_open` → `closed` cycle and print the snapshot at each step. The leader-election failover demo is run by hand: identify the current leader replica, kill its process, and observe the surviving replicas elect a new leader within a few seconds.

VII. LIMITATIONS AND FUTURE WORK

Several scoping decisions kept the project achievable in one semester. Each is, in our view, a clean extension point rather than a flaw.

Bully-style election, not Raft. Our election is a working bully-style protocol with a replication log and follower forwarding, enough to demonstrate the consistency-vs.-availability tradeoff and enough to survive a leader kill. A formally verified Raft implementation is the natural next step, especially because we already have the log abstraction in place.

In-memory state. All in-memory state (rate-limit buckets, breakers, registry, queue, brownout window, tuner weights) resets on restart. Only user-feedback rows are persisted, in SQLite at `artifacts/user_feedback.sqlite3`. Swapping in Redis or Postgres for the rest is a single-file change per subsystem because the state is already encapsulated behind narrow interfaces.

Heuristic intent classifier. The classifier is a keyword/heuristic in `router/intent.py`. We chose this so the distributed-systems contributions could be unit-tested without an external model, and so the demo would be deterministic. A small fine-tuned classifier is the obvious upgrade, especially for prompts where the keyword signals are ambiguous.

Mock providers default in the demo. The system ships working Groq and Gemini adapters in `router/provider_clients.py`, but the demo defaults to mock providers because they let us simulate latency, cost, and four distinct failure modes without burning real-provider quota and without a network dependency during grading.

No cross-router state sharing. The brownout window, the weight-tuner state, and the service registry are all per-router. A multi-router deployment would converge independently on each router. Sharing this state across routers is a known unscoped item.

Premium classification is heuristic. "Premium" is decided from the provider name pattern in the policy engine. A real deployment would carry tier metadata from a provider catalog.

VIII. CONCLUSION

The 3 a.m. story we opened with is not really an LLM problem; it is the absence of a control plane for traffic that should obviously *have* one. IntelliRoute is our argument, in code, that the same distributed-systems machinery we already use for shared compute, service discovery, leader election, replication, circuit breakers, brownout, priority queueing, per-data-path consistency, applies just as cleanly to the LLM fleet, and that wiring it together is a one-semester project, not a multi-quarter platform investment. The demo on the dashboard runs the full pipeline in roughly 140 ms on a laptop, the test suite runs in 12 seconds, and every concept we read about in CMPE 273 maps to a specific module we can point at in the source tree.