Power Lever: On-Demand Speculative Decoding, Dynamic GPU Orchestration, and Agentic Routing

Kesavan Ramakrishnan Feolu Kolawole Nick Rui

Feb 15, 2026

Abstract

This report documents *Power Lever*, a three-tier inference system that routes each prompt to a right-sized GPU worker and configures speculative decoding on-demand. The system consists of a Next.js frontend, a FastAPI gateway, and four Modal-hosted GPU workers running vLLM. In AUTO mode, a Claude-based router agent (implemented both as an Anthropic tool-use loop and via the Claude Agent SDK with MCP tools and structured output) classifies prompt complexity and selects a tier. Responses stream back to the frontend using Server-Sent Events (SSE) with configuration, trace, token, and metrics events. We describe the code-level architecture, routing workflow, tier registry, and the speculative decoding and orchestration mechanisms as implemented in the repository.

1 System Overview

Power Lever addresses the mismatch between *prompt complexity* and *inference hardware*: many queries do not require frontier-class GPUs. The system exposes a "power" control (0–100) and an AUTO mode. Requests are mapped to one of four tiers, each with distinct GPU and model configurations (including draft models for speculative decoding where feasible).

1.1 Repository Architecture

The codebase is organized into three primary components:

- **Frontend** (`frontend/`): Next.js 14 App Router app. Proxies requests to the gateway and renders streamed SSE events.
- **Gateway** (`gateway/`): FastAPI service that runs routing, computes sustainability metrics, dispatches to Modal workers, and streams SSE.
- **Modal Workers** (`modal_workers/`): Modal application defining four GPU worker classes (Eco/Balanced/Performance/Ultra) wrapping vLLM engines with tier-specific settings and fallback chains.

A typical deployed topology is:

Browser → Vercel (Next.js) → Modal Gateway (FastAPI) → Modal GPU Workers (vLLM).

2 End-to-End Request Workflow

2.1 SSE Streaming Contract

The gateway endpoint `POST /api/inference` returns an SSE stream. Events are emitted in a structured JSON envelope (“`type`” + “`data`”).

Listing 1: SSE event sequence used by the gateway and frontend.

```
data: {"type":"config","data":{"tier":"balanced","gpu":"A10G",
  "target_model":"...","draft_model":"...",
  "spec_decoding":{"enabled":true,"K":7,"tau":0.5}}}

data: {"type":"trace","data":{"step":"classify_prompt","result":{...},"duration_ms":2}}

data: {"type":"reasoning","data":{"text":"..."}}

data: {"type":"token","data":{"text":"The ","logprob":0.0,"spec_accepted":true}}

data: {"type":"metrics","data":{"ttft_ms":..., "tokens_per_sec":..., ...}}

data: {"type":"done"}
```

On the frontend, `frontend/lib/streamUtils.ts` posts JSON to a URL and incrementally parses SSE by splitting on `\n\n` and decoding `data:` lines into typed events.

2.2 Gateway Inference Endpoint

In `gateway/main.py`, `POST /api/inference`:

1. Calls `route_request(prompt, power, speed, mode)` to obtain an `InferenceConfig`, trace steps, and natural-language reasoning.
2. Emits `config`, `trace`, and optionally `reasoning`.
3. Dispatches to a Modal worker via `allocate_worker`, which selects a worker class by name and iterates `generate_stream` using Modal's `remote_gen`.
4. Emits `token` events as they arrive.
5. Computes sustainability metrics and emits a terminal `metrics` event followed by `done`.

The gateway supports a mock mode (for frontend development) controlled by `INFERENCE_MODE` and request-level `use_modal`. In mock mode, the gateway itself emits synthetic tokens.

3 Tier Registry and Configuration

Tier definitions are centralized in `gateway/config.py` and include: power range, GPU type, target model, optional draft model, speculative decoding parameters, and the Modal worker class name.

Speed knob. The gateway maps a `speed` value (0–100) to sampling parameters (`max_tokens`, `temperature`, `top_p`) via `SPEED_CONFIGS` in `gateway/config.py`.

4 Speculative Decoding On-Demand

4.1 Implementation in Modal Workers

Speculative decoding is enabled per tier by configuring vLLM `AsyncEngineArgs` with a `speculative_config` dictionary. The shared helper `modal_workers/common/engine.py` builds engine kwargs:

- `model`: points to `/models/<hf_model_id>` on a shared Modal volume.
- `speculative_config`: when enabled, uses `method="draft_model"`, `model=<draft_path>`, `num_speculative_tok`

Each worker class (`EcoWorker`, `BalancedWorker`, `PerformanceWorker`, `UltraWorker`) loads a vLLM engine at container startup (`@modal.enter()`), then provides:

- `generate`: batch generation, returns all tokens and telemetry.
- `generate_stream`: yields token dictionaries for streaming.

4.2 Fallback Chains

Workers implement explicit fallback chains to handle VRAM constraints or model availability:

- **Eco**: tries (draft=TinyLlama, K=10) then falls back to target-only.
- **Balanced**: tries draft=Nemotron-Mini, then TinyLlama fallback, then reduced `max_model_len`, then target-only.
- **Performance**: tries AWQ 70B + 8B draft (K=4), then AWQ 70B alone, then 8B+ TinyLlama, then 8B alone.
- **Ultra**: no speculative decoding; tries 70B FP8, else falls back to 8B if the 70B FP8 weights are missing from the volume.

Note on `tau`. The gateway schema and tier config include `tau` as an "acceptance threshold" field. In the current implementation, workers pass `num_speculative_tokens` to vLLM but do not wire `tau` into engine arguments; thus `tau` is informational at the API layer and UI.

Token acceptance telemetry. Workers currently mark `spec_accepted` conservatively as a boolean "speculative decoding enabled" flag (always true for tiers with a draft model; false for Ultra). The code does not parse per-token accept/reject signals from vLLM outputs; this is a known limitation for fine-grained acceptance-rate reporting.

5 Dynamic GPU Orchestration with Modal

5.1 Modal App and Deployment

Modal workers are grouped under the Modal app name `"power-lever"` (`modal_workers/app.py`). The gateway can be deployed separately as `"power-lever-gateway"` (`gateway/modal_gateway.py`) as a Modal ASGI app.

Workers are registered by importing them in `modal_workers/deploy.py`. Demo mode is supported via `DEMO_MODE=1`, which sets `min_containers=1` for each tier to reduce cold-start latency.

5.2 Model Weight Caching via Modal Volumes

Model weights are cached in a persistent Modal volume (`power-lever-models`) mounted at `/models`. The seeding workflow (`modal_workers/volumes.py` or `scripts/seed_models.py`) uses HuggingFace `snapshot_download` to materialize:

- "Small" models for Eco/Balanced.
- Optionally, "Large" models for Performance/Ultra (70B AWQ + 70B FP8).

After downloads, the volume is committed so subsequent container starts reuse cached weights.

5.3 Runtime Dispatch and Availability Checks

`gateway/tools/allocator.py` dispatches by resolving a Modal class name from the config (e.g., `EcoWorker`) using `modal.Cls.from_name("power-lever", cls_name)` and iterating its `generate_stream.remote_` generator.

GPU availability is approximated by checking if worker classes exist (i.e., are deployed) in `gateway/tools/gpu_pool.py`. If a class lookup fails, the tool reports the tier as unavailable and returns a cold-start estimate (seconds) per tier.

6 Router Agent with Claude and Claude Agent SDK

Power Lever implements routing in AUTO mode using a two-stage design:

- **Deterministic mapping tools** for classification and config construction (fast, testable).
- **Claude-generated reasoning** for transparency (optional).

6.1 Heuristic Classifier and Config Builder

The gateway includes local tools:

- `classify_prompt` (`gateway/tools/classify.py`): heuristic rules for safety-critical, code, math, creative writing, simple Q&A, and general reasoning. Produces a difficulty score (1–10).
- `build_config` (`gateway/tools/config_builder.py`): maps `power` to a tier, merges tier config with speed sampling params, and optionally emits an `escalation` directive when difficulty is high but tier is low.

AUTO mode maps difficulty → power using a fixed lookup table (`AUTO_DIFFICULTY_TO_POWER` in `gateway/config.py`).

6.2 Anthropic Tool-Use Agent (Two Modes)

`gateway/router_agent.py` supports:

- **Single-shot (default)**: runs local tools deterministically, then makes one Claude API call to generate a brief explanation of the routing decision.
- **Multi-turn**: runs a full tool-use loop where Claude decides which tools to invoke and in what order (primarily for demos).

The system is designed to fall back to deterministic routing on agent failure or timeout.

6.3 Claude Agent SDK Router (MCP + Structured Output)

`gateway/agent_router.py` provides a second implementation using the Claude Agent SDK:

- Exposes tools as an MCP server in `gateway/agent_tools.py`.
- Constrains outputs via a JSON schema (`OUTPUT_SCHEMA`) to ensure the agent returns a valid configuration object with a `reasoning` field.
- Defines a small "classifier" subagent specialized for difficulty and category labeling.

7 Routing Algorithm (As Implemented)

Algorithm 1 Gateway routing and dispatch (AUTO vs MANUAL) summarized from `gateway/main.py` and `gateway/router_agent.py`.

```
Require: prompt, mode ∈ {auto, manual}, power, speed
 1: $trace \leftarrow [\,]$
 2: if mode == auto then
 3:     $c \leftarrow$ ClassifyPrompt($prompt$)
 4:     $d \leftarrow c$.difficulty
 5:     $p \leftarrow$ DifficultyToPower($d$)
 6:     $s \leftarrow 50$
 7: else
 8:     $p \leftarrow$ power (default 50)
 9:     $s \leftarrow$ speed (default 50)
10:     $d \leftarrow 5$
11: end if
12: $g \leftarrow$ CheckGpuPool()                        ▷ availability signal for the agent/UI
13: $cfg \leftarrow$ BuildConfig($p, s, d, mode$)         ▷ tier + models + spec + sampling
14: emit SSE config + trace (+ optional reasoning)
15: for token ∈ ModalStream(cfg.modal_function, prompt, cfg.generation) do
16:     emit SSE token
17: end for
18: compute savings; emit SSE metrics then done
```

8 Sustainability and Cost Modeling

The gateway computes energy, water, CO_2, and cost metrics in `gateway/sustainability.py`. GPU "active wattage" and hourly cost are taken from `gateway/config.py`. For inference time t and active power P:

$$E_{\text{kWh}} = \frac{P \cdot t}{3600 \cdot 1000}.$$

Water and CO_2 are derived using fixed multipliers:

$$W_{\text{L}} = 1.8 \cdot E_{\text{kWh}}, \quad C_{\text{kg}} = 0.39 \cdot E_{\text{kWh}}.$$

Savings are computed relative to an H100 "frontier baseline" using a separate baseline inference time.

9 Limitations and Engineering Notes

- **Spec acceptance granularity**: per-token accept/reject is not currently extracted from vLLM outputs; acceptance rate is approximated.
- **`tau` is not enforced**: API/config includes `tau` but workers do not wire it into vLLM runtime configuration.
- **GPU pool reporting**: availability is inferred from worker class lookup; warm container counts are not queried from Modal runtime state.
- **SSE parsing**: frontend parsing assumes `data: <json>\n\n` framing; this matches the gateway emitter.

10 Related Work

Speculative decoding is a known method for accelerating decoding by combining a draft model with a target verifier model. Serverless GPU scheduling and right-sizing inference workloads aligns with broader trends in elastic inference infrastructure.

11 Conclusion

Power Lever demonstrates a practical, hackathon-scale implementation of (i) on-demand speculative decoding configured per tier, (ii) dynamic GPU orchestration using Modal classes and volumes, and (iii) an LLM router agent implemented both with the Anthropic Messages API and with the Claude Agent SDK (MCP + structured output). The system provides an end-to-end SSE streaming interface with traceability and sustainability metrics, enabling interactive demonstrations of quality-preserving efficiency gains through right-sized inference.

References

[1] Y. Leviathan, M. Kalman, and Y. Matias, "Fast Inference from Transformers via Speculative Decoding," 2023.

[2] vLLM: High-throughput LLM serving. https://github.com/vllm-project/vllm

[3] Modal: Serverless GPU infrastructure. https://modal.com

[4] Anthropic Claude API and Agent SDK documentation. https://docs.anthropic.com

[5] W3C, "Server-Sent Events," https://html.spec.whatwg.org/multipage/server-sent-events.html