vLLM Semantic Router Fleet Sim

A Practitioner's Guide to LLM GPU Fleet Simulation:
Education, Usage, and Case Studies

vLLM Semantic Router Team

2026

Contents

1 Who Should Read This Guide

This guide is for **ML platform engineers, capacity planners, and researchers** who need to answer questions such as:

- *"How many A100 GPUs do I need to serve 300 requests/s at P99 TTFT ≤ 500 ms?"*
- *"If my traffic doubles next quarter, when will I breach my SLO and by how much?"*
- *"Should I run one homogeneous GPU fleet or split into short- and long-context pools?"*
- *"My fleet is sized for today's workload. How much does Compress-and-Route (C&R) help?"*
- *"Which routing algorithm minimises tail latency for my agent-heavy traffic?"*

`vllm-sr-sim` is the CLI and service package for **vLLM Semantic Router Fleet Sim**, a fleet-level discrete-event simulator for LLM inference. Unlike single-instance simulators such as Vidur, it models the full picture: multiple GPU pools, heterogeneous hardware, pluggable routing algorithms, and an analytical optimizer that finds the minimum-cost fleet meeting a P99 TTFT SLO.

Prior art and scope. Single-instance simulators (e.g., Vidur [Agrawal et al., 2024], vLLM offline profilers) are invaluable for engine-level tuning — understanding KV-cache block utilization, preemption rates, and chunked-prefill scheduling. This tool is complementary: it operates one abstraction level higher, modeling the fleet as a network of M/G/c queues and asking how many GPUs are needed, how they should be grouped into pools, and which routing policy minimises tail latency.

What this guide covers. Section 2 explains the underlying concepts (queueing theory, the cost cliff, TTFT decomposition) at a practitioner level. Section 3 describes the simulator's internals. Section 4 is a 5-minute quick start. Section 5 documents every CLI command. Section 6 walks through four real-world case studies with full command output. Section 9 explains how to extend the simulator.

2 Background: Key Concepts

2.1 Why GPU Fleets Are Expensive to Get Right

Modern LLMs support context windows up to 128K tokens, but production traces tell a different story: in the Azure LLM Inference Trace, ∼78% of requests use fewer than 2K tokens (CDF value $F(2{,}048) \approx 0.78$). A fleet provisioned for worst-case context length allocates KV-cache memory that sits idle for nearly every request.

The cost of this mismatch is not gradual. The simulator models KV-cache capacity via PagedAttention blocks: each A100-80GB holds `total_kv_blks=65 536` blocks of `blk_size=16` tokens. A sequence using up to 64K tokens needs $\lceil 65{,}536/16 \rceil = 4{,}096$ blocks, so only $\lfloor 65{,}536/4{,}096 \rfloor =$ **16** sequences fit concurrently. The same GPU at an 8K context boundary needs only $\lceil 8{,}192/16 \rceil =$ 512 blocks per sequence, allowing $\lfloor 65{,}536/512 \rfloor =$ **128** concurrent sequences — an **8× throughput difference** between the two pool types.

2.2 Pool Routing and the Cost Cliff

Pool routing splits the fleet into two pools:

- **Short pool** $\mathcal{P}_s$: sized for B_{short} tokens (e.g., 8K), 128 concurrent sequences per GPU.
- **Long pool** $\mathcal{P}_l$: sized for full context (e.g., 64K), 16 concurrent sequences per GPU.

Each request is routed to $\mathcal{P}_s$ if its estimated total token budget $L_{\text{total}} \leq B_{\text{short}}$, else to $\mathcal{P}_l$. This saves 16–38% of GPU cost versus a homogeneous fleet.

The **cost cliff** is the discontinuity at B_{short}: a request at $B_{\text{short}}+1$ tokens occupies one of only 16 long-pool slots, consuming 8× more throughput capacity than its immediate neighbour below B_{short}. Requests in the *borderline band* $(B_{\text{short}}, \gamma B_{\text{short}}]$ are not genuinely long; they just barely crossed the threshold. Compress-and-Route (C&R) addresses this by compressing borderline prompts back below B_{short} at the gateway.

2.3 The M/G/c Queue: Explaining the Math Behind GPU Sizing

The simulator's analytical core models each GPU pool as an **M/G/c queue**:

Letter	Stands for	Meaning in our context
M	Markovian arrivals	Requests arrive as a Poisson process with rate λ; each arrival is independent
G	General service	Each request takes a different amount of GPU time; captured by mean $\mathbb{E}[S]$ and variance $\text{Var}[S]$
c	c servers	The pool has c GPUs running in parallel; requests queue until a GPU is free

Why "G" not "M" for service time? Real LLM requests vary enormously: a 50-token chat reply takes ∼100 ms; a 50K-token RAG response takes ∼30 s. Using an exponential service distribution (M/M/c) would underestimate tail latency for high-variance workloads. The **Kimura (1994) correction** scales the Erlang-C waiting time by $(1+C_s^2)/2$, where $C_s^2 = \text{Var}[S]/(\mathbb{E}[S])^2$ is the squared coefficient of variation. A high-C_s^2 workload (heavy-tailed request lengths) queues much worse than M/M/c predicts.

The Erlang-C formula. Let $\varrho = \lambda/(c\mu)$ be the per-server utilization. The probability that an arriving request has to wait (all c GPUs busy):

$$C(c,\varrho) = \frac{(c\varrho)^c/(c!\,(1-\varrho))}{\sum_{k=0}^{c-1}(c\varrho)^k/k! + (c\varrho)^c/(c!\,(1-\varrho))}. \tag{1}$$

The P99 queue waiting time under the Kimura approximation:

$$W_{99} \approx \frac{C(c,\varrho)}{c\mu(1-\varrho)} \cdot \frac{1+C_s^2}{2} \cdot \ln(100). \tag{2}$$

Key Insight. W_{99} grows without bound as $\varrho \to 1$ (GPU utilization approaches 100%). At $\varrho = 0.94$, the analytical model predicts P99 wait ≈ 325 ms. At $\varrho = 0.97$, it exceeds 800 ms. The simulator enforces $\varrho_{\max} = 0.85$ to stay in the regime where the approximation is accurate (8–14% conservative overestimate vs. DES).

The service time model. GPU iteration latency under continuous batching scales with the number of concurrent sequences n_{slots}:

$$t_{\text{iter}} = W + H \cdot n_{\text{slots}}, \tag{3}$$

where $W = 8$ ms (baseline compute for Llama-3-70B/A100) and $H = 0.65$ ms/slot (memory-bandwidth cost per concurrent sequence). A request with L_{in} input tokens and L_{out} output tokens has expected service time:

$$\mathbb{E}[S] = \frac{\lceil L_{\text{in}}/C_{\text{chunk}} \rceil + L_{\text{out}}}{n_{\max}} \cdot t_{\text{iter}}, \tag{4}$$

where C_{chunk} is the prefill chunk size (default 512) and $n_{\max}$ is the pool's max concurrent sequences per GPU.

2.4 TTFT Decomposition

Time-to-First-Token (TTFT) decomposes as:

$$\text{TTFT} = \underbrace{W_{\text{queue}}}_{\text{wait for GPU slot}} + \underbrace{T_{\text{prefill}}}_{\text{process input tokens}} + \underbrace{t_{\text{iter}}}_{\text{first decode step}} . \tag{5}$$

The SLO applies to the full TTFT, not just the queue wait. $T_{\text{prefill}} = \lceil L_{\text{in}}/C_{\text{chunk}} \rceil \cdot t_{\text{iter}}$ is the wall-clock prefill time (independent of batch size since all slots run in parallel each iteration).

Warning. For large requests near B_{short}, T_{prefill} alone can exceed 100 ms. At $L_{\text{in}} = 8{,}192$ tokens with $C_{\text{chunk}} = 512$: $\lceil 8{,}192/512 \rceil = 16$ iterations $\times$ 8 ms = 128 ms prefill, leaving only 372 ms for queueing under a 500 ms SLO. This is why the SLO budget is tight for large B_{short} values.

3 Simulator Architecture

3.1 Component Overview

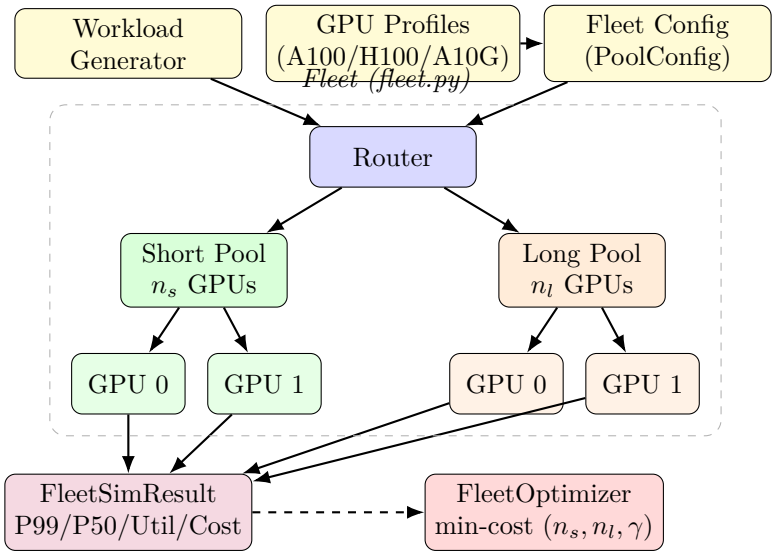


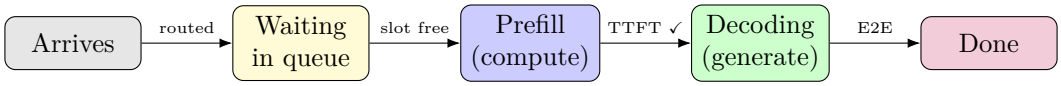
Figure 1: High-level component diagram. Each GPU instance runs an M/G/c-style discrete-event simulation.

The simulator has four layers:

1. **Workload** (`fleet_sim/workload/`) — generates request arrivals with a Poisson process. Supports empirical CDFs (from JSON files) and synthetic Poisson distributions.
2. **GPU Profiles** (`fleet_sim/gpu_profiles/`) — hardware constants $(W, H, n_{\max}, C_{\max})$ for each GPU type. Pre-defined: A100-80GB, H100-80GB, A10G.
3. **Core** (`fleet_sim/core/`) — the simulation engine:
   - `request.py`: `Request` dataclass, lifecycle states (WAITING → PREFILL → DECODING → DONE).
   - `instance.py`: single GPU; implements fast request-level DES using a min-heap event queue. Each request triggers exactly two events: arrival and completion.
   - `pool.py`: c GPU instances behind a shared queue; load balancing across instances.
   - `fleet.py`: two pools + router; drives the simulation clock, collects `FleetSimResult`.
4. **Routing** (`fleet_sim/routing/`) — pluggable routing algorithms (Section 3.4).
5. **Optimizer** (`fleet_sim/optimizer.py`) — two-phase analytical sweep + DES verification to find minimum-cost fleet.

3.2 Request Lifecycle

A request passes through these stages:



Measured metrics per request:

- **queue_wait**: time from arrival until a GPU slot is assigned
- **ttft**: time from arrival to first output token (= queue_wait + prefill + 1 decode step)
- **e2e_latency**: time from arrival to last token

3.3 Fast Instance Simulation

Each GPU instance uses a **request-level event model** rather than token-by-token simulation. For a request with L_{in} input and L_{out} output tokens, the simulator computes upfront:

$$S_{\text{raw}} = (\text{prefill\_iters} + L_{\text{out}}) \times t_{\text{iter}}, \tag{6}$$

$$S_{\text{eff}} = S_{\text{raw}}/n_{\max}. \quad \text{(slot freed for scheduling)} \tag{7}$$

Two events are inserted into a min-heap: a TTFT event at `now + prefill_iters * iter_t`, and a completion event at `now + S_eff`. This is $O(\log n)$ per request rather than $O(L_{\text{out}})$, enabling simulation of millions of requests per second.

3.4 Available Routing Algorithms

Router	Behaviour
`LengthRouter`	Routes to the smallest pool whose `max_ctx` fits the request. Default for length-partitioned fleets.
`CompressAndRouteRouter`	Like `LengthRouter` but compresses borderline requests in $(B_{\text{short}}, \gamma B_{\text{short}}]$ to the short pool.
`ModelRouter`	Routes each request to the pool matching `request.model_id`; falls back to the default pool.
`SemanticRouter`	Routes using a user-supplied `classify_fn(Request)` $\rightarrow$ `pool_id`. Supports any classifier: intent detection, embedding lookup, keyword rules, LLM judge. See Section 7.4.
`LeastLoadedRouter`	Sends each request to the pool with the fewest queued requests (runtime state).
`RandomRouter`	Uniform random assignment. Baseline.

Key Insight. Pool routing by prompt length is **one use case** of the simulator. A fleet may have N pools distinguished by model, hardware tier, or any other criterion. Use `ModelRouter`, `SemanticRouter`, or the `simulate-fleet` CLI command (Section 7) for these topologies.

4 Quick Start

4.1 Installation

```
git clone https://github.com/vllm-project/semantic-router
cd semantic-router/src/fleet-sim
pip install -r requirements.txt    # numpy, scipy only
```

The simulator has no GPU or ML framework dependencies — it runs entirely in CPU-only Python.

4.2 Your First Run: Size a Fleet in 30 Seconds

```
vllm-sr-sim optimize \
  --cdf data/azure_cdf.json \
  --lam 200 --slo 500 --b-short 6144 \
  --gpu-short a100 --gpu-long a100
```

This finds the minimum-cost A100 fleet for 200 req/s Azure traffic at P99 TTFT $\leq$ 500 ms with a 6K-token pool boundary. It takes about 2 seconds. The output is shown in Section 6.1.

4.3 Data Format

The `-cdf` argument expects a JSON file with a list of `[token_length, cumulative_fraction]` pairs:

```
# azure_cdf.json (excerpt)
[
  [128,  0.028],
  [256,  0.068],
  [512,  0.289],
  [1024, 0.404],
  [2048, 0.780],
  [4096, 0.898],
  [6144, 0.976],
  [8192, 1.000],
  ...
]
```

The simulator interpolates linearly between points. You can generate a CDF from any request log using the helper in `fleet_sim/workload/trace.py`.

4.4 Available GPU Profiles

Profile	W (ms)	H (ms/slot)	$n_{\max}$ (short)	$n_{\max}$ (long)	Typical use
`a100`	8.0	0.65	128	16	General purpose
`h100`	4.0	0.32	128	16	Low-latency SLO
`a10g`	12.0	0.90	64	8	Cost-sensitive long pool

Pre-built profiles are calibrated for Llama-3-70B. Use `CUSTOM(name, W, H, ...)` to supply your own constants, or `ProfileBuilder` to compute them from hardware and model specs.

4.5 Physics-Informed Profiles via `ProfileBuilder`

The simulator can derive W, H, and `total_kv_blks` from first principles using a *hardware spec* $\times$ *model spec* pair, inspired by the bottom-up roofline approach of AIConfigurator [Xu et al., 2025].

Hardware catalog. Eight pre-built `HardwareSpec` instances cover all current NVIDIA generations (A100, L40S, B60, H100 SXM, H200 SXM, B200 SXM, GB200 NVL, GB300 NVL). Each spec records peak memory bandwidth, VRAM capacity, FP16/FP8 TC TFLOPS, TDP, and estimated cloud cost. Empirical constants from AIConfigurator's silicon measurements are embedded directly:

Constant	Value
Memory bandwidth scale factor	0.80 (achievable fraction of peak)
Per-layer constant latency	3 µs (NCCL + kernel launch)
NCCL overhead, TP=2	0.5 GB reserved
NCCL overhead, TP=4	1.0 GB reserved
NCCL overhead, TP=8	2.0 GB reserved

Model catalog. Ten `ModelSpec` instances cover Llama-3.1 (8B, 70B, 405B), Qwen3 (8B, 32B, 30B-A3B, 235B-A22B), and DeepSeek-V3. Each spec includes layer count, hidden dimension, GQA KV-head count, and MoE configuration (number of experts, top-k routing).

Dense model formulas. For a dense model with P parameters split across T_P tensor-parallel GPUs, loaded in d bytes per weight, the iteration latency parameters are:

$$W = \frac{P \cdot d / T_P}{B_{\text{mem}} \cdot 0.80} + L \cdot 3\,\mu\text{s}$$

$$H = \frac{k_{\text{KV/token}} / T_P}{B_{\text{mem}} \cdot 0.80} \cdot \bar{C}$$

where B_{mem} is peak GPU memory bandwidth, L is the number of layers, $k_{\text{KV/token}}$ is the KV-cache bytes per sequence token (all layers), and $\bar{C}$ is the representative context length (set via `ServingConfig.mean_ctx_tokens`).

MoE models. MoE models use a silicon-measured kernel lookup table (`_MOE_TABLE` in `builder.py`) containing H100 SXM latencies for DeepSeek-V3 and Qwen3 variants across FP16/FP8/INT4 quantization. Results are interpolated for intermediate token counts and scaled to other GPUs by memory-bandwidth ratio.

Example usage.

```
from fleet_sim import H100_SXM, LLAMA_3_1_70B
from fleet_sim.gpu_profiles import ProfileBuilder, ServingConfig

profile = ProfileBuilder().build(
    H100_SXM,
    LLAMA_3_1_70B,
    ServingConfig(tp=8, dtype_bytes=2, mean_ctx_tokens=2048),
)
print(profile.summary())
# H100-SXM | Llama-3.1-70B | TP8 FP16
#   W = 5.82 ms   H = 0.039 ms/seq   KV blocks = 4271  cost = $32.16/hr
```

The resulting `ComputedProfile` satisfies the `GpuProfile` Protocol and can be used anywhere a `ManualProfile` would be used — in `PoolConfig`, the `FleetOptimizer`, or the `DisaggFleetOptimizer`.

4.6 Disaggregated Fleet Optimizer

`DisaggFleetOptimizer` sizes separate prefill and decode worker pools following the rate-matching approach of AIConfigurator [Xu et al., 2025]. Each pool runs with its own tensor-parallelism degree and quantization setting.

Two empirical degradation factors account for the overhead of KV-transfer between prefill and decode nodes in a disaggregated deployment:

$$R_{\text{sys}} = \min(n_{\text{pre}} \cdot \mu_{\text{pre}} \cdot \alpha_{\text{pre}},\ n_{\text{dec}} \cdot \mu_{\text{dec}} \cdot \alpha_{\text{dec}})$$

$$\text{TTFT}_{\text{eff}} = T_{\text{prefill}} \cdot \beta_{\text{TTFT}}$$

with embedded constants $\alpha_{\text{pre}} = 0.90$, $\alpha_{\text{dec}} = 0.92$, $\beta_{\text{TTFT}} = 1.80$ (measured in Xu et al. 2025).

```
from fleet_sim import DisaggFleetOptimizer, H100_SXM, LLAMA_3_1_70B
from fleet_sim.gpu_profiles import ProfileBuilder, ServingConfig

builder = ProfileBuilder()
prefill = builder.build(H100_SXM, LLAMA_3_1_70B,
    ServingConfig(tp=4, dtype_bytes=2, mean_ctx_tokens=1024, phase="prefill"))
decode  = builder.build(H100_SXM, LLAMA_3_1_70B,
    ServingConfig(tp=8, dtype_bytes=2, mean_ctx_tokens=1024, phase="decode"))

opt = DisaggFleetOptimizer(
    prefill_profile=prefill, decode_profile=decode,
    mean_isl=1024, mean_osl=256,
    slo_ttft_ms=2000, slo_tpot_ms=100, max_ctx=4096,
)
result = opt.optimize(max_prefill=8, max_decode=8)
result.print_report()
# Disaggregated fleet:  4x prefill (TP4)  +  6x decode (TP8)
#   System rate: 87.3 req/s   TTFT: 1842 ms   TPOT: 94.1 ms
#   Total GPUs: 64   Cost: $257.2/hr
```

5 CLI Reference

5.1 `optimize`: Find the Minimum-Cost Fleet

```
vllm-sr-sim optimize \
  --cdf PATH           # workload CDF JSON (required)
  --lam FLOAT          # arrival rate (req/s, default: 100)
  --slo FLOAT          # P99 TTFT budget (ms, default: 500)
  --b-short INT        # pool boundary tokens (default: 8192)
  --long-max-ctx INT   # long-pool max context (default: 65536)
  --gpu-short TYPE     # a100 | h100 | a10g (default: a100)
  --gpu-long  TYPE     # a100 | h100 | a10g (default: a100)
  --gamma-max FLOAT    # max C&R bandwidth to sweep (default: 2.0)
  --verify-top INT     # DES-verify top-N analytical candidates (3)
  --out PATH           # save JSON results to file
```

Algorithm: two-phase search. Phase 1 sweeps $\gamma \in \{1.0, 1.1, \ldots, \gamma_{\max}\}$ using the analytical M/G/c model to score each configuration. Phase 2 runs DES on the top-`verify-top` candidates and returns the best DES-verified result.

5.2 `simulate`: Evaluate a Fixed Fleet

```
vllm-sr-sim simulate \
  --cdf PATH --lam FLOAT --slo FLOAT \
  --b-short INT --gpu-short TYPE --gpu-long TYPE \
  --n-s INT            # short-pool GPU count (required)
  --n-l INT            # long-pool GPU count (required)
  --gamma FLOAT        # C&R gamma (1.0 = no compression, default)
  --n-req INT          # requests to simulate (default: 10000)
  --seed INT           # random seed (default: 42)
  --out PATH
```

Use this to **validate** a fleet you already have, or to test a specific configuration the optimizer didn't propose.

5.3 `whatif`: Traffic Spike and Growth Analysis

```
vllm-sr-sim whatif \
  --cdf PATH --slo FLOAT --b-short INT \
  --gpu-short TYPE --gpu-long TYPE \
  --lam-range 50 100 150 200 250 300   # list of rates to sweep
  --out PATH
```

Runs `optimize` at each λ in the range and prints a table of optimal fleet sizes and costs. Essential for capacity planning over a multi-quarter growth forecast.

5.4 `compare-routers`: Router Benchmarking

```
vllm-sr-sim compare-routers \
  --cdf PATH --lam FLOAT --slo FLOAT \
  --b-short INT --gpu-short TYPE --gpu-long TYPE \
  --n-s INT --n-l INT  # fixed fleet size for all routers
  --n-req INT --seed INT
```

Runs the same fixed fleet under all available routing algorithms and compares P99 TTFT, SLO compliance, and utilization.

5.5 `grid-flex`: Power–Latency Trade-off for Demand Response

```
vllm-sr-sim grid-flex \
  --cdf PATH           # workload CDF JSON (required)
  --lam FLOAT          # arrival rate (req/s)
  --n-gpus INT         # fixed fleet size to evaluate (required)
  --gpu TYPE           # a100 | h100 | a10g (default: h100)
  --slo FLOAT          # P99 TTFT SLO (ms, default: 500)
  --max-ctx INT        # max context window (tokens, default: 8192)
  --flex-pcts F1 F2... # power-reduction percentages to sweep
                       # (default: 0 5 10 15 20 25 30 40 50)
  --out PATH           # save results to JSON
```

What it does. Sweeps power-reduction target percentages and, for each one, reports: (1) the implied batch-size cap $n_{\max}$ derived by inverting the GPU power model; (2) estimated per-GPU power draw (W) and total fleet power (kW); (3) analytical P99 TTFT at the reduced concurrency; and (4) whether the SLO is still met. The table closes with the *maximum safe flex depth* — the largest curtailment that still meets the SLO.

Use case. Operators enrolled in utility demand-response (DR) programmes must commit to a power reduction level in advance. This command answers: *"how much power can I offer without breaching my latency SLA?"*

Power model. Each GPU profile stores `power_idle_w` (power at batch ≈ 1) and `power_nominal_w` (power at full `max_slots` concurrency), sourced from ML.ENERGY Benchmark v3.0 measurements on H100-SXM5 and NVIDIA spec sheets. Power is linearly interpolated between these two scalars — a conservative approximation accurate to within $\pm 5\%$ in the 40–100% load range relevant for DR operations.

Example output (30 H100s, $\lambda = 300\,\text{req/s}$, SLO $= 200\,\text{ms}$):

```
======================================================================
  Grid Flexibility Analysis
  Fleet: 30 GPUs  lam=300 req/s  SLO=200 ms
  Baseline: 13.5 kW fleet power  (450 W/GPU)
======================================================================
  Flex  n_max   W/GPU  Fleet kW  P99 TTFT   SLO
------  -----  ------  --------  --------   ---
    0%    128   450W    13.5kW      7.9ms    OK
    5%    115   435W    13.0kW      7.9ms    OK
   10%    102   420W    12.6kW      7.9ms    OK
   15%     90   405W    12.2kW      7.9ms    OK
   20%     77   390W    11.7kW      7.9ms    OK
   25%     64   375W    11.2kW   1058.2ms  BREACH
   30%     51   360W    10.8kW    inf ms   BREACH
   40%     26   330W     9.9kW    inf ms   BREACH

  Max safe flex depth: 20%  (saves 1.8 kW, P99=7.9ms)
```

The abrupt transition at 25% corresponds to the Erlang-C saturation threshold: below $n_{\max} = 64$ the arrival load exceeds the available KV-slot count and the queue diverges. The simulator locates this cliff analytically, before any hardware change.

6 Case Studies

6.1 Case Study 1: "How Many GPUs Do I Need?"

Scenario. You are launching an LLM API backed by Llama-3-70B on A100-80GB GPUs. Historical traffic analysis gives you the Azure conversational CDF. You expect 200 requests/second at peak and need P99 TTFT $\leq 500\,\text{ms}$. The pool boundary is set at 6,144 tokens (the CDF knee, where $F(B_{\text{short}}) \approx 0.976$).

Command.

```
vllm-sr-sim optimize \
  --cdf data/azure_cdf.json \
  --lam 200 --slo 500 --b-short 6144 \
  --gpu-short a100 --gpu-long a100
```

Output.

```
[1/2] Analytical sweep over gamma in [1.0 .. 2.0] ...
[2/2] DES verification of top-3 candidates...
  Verifying gamma=1.3 (n_s=50, n_l=10)...
    P99 short=80.9ms  long=392.0ms  SLO:PASS
  Verifying gamma=1.2 (n_s=50, n_l=11)...
    P99 short=78.6ms  long=411.8ms  SLO:PASS
  Verifying gamma=1.0 (n_s=49, n_l=13)...
    P99 short=91.2ms  long=450.0ms  SLO:PASS

Fleet Optimization Report
```

```
B_short=6,144  lambda=200 req/s  SLO=500ms

Best (analytical): gamma=1.3  n_s=50  n_l=10  total=60  $1161.6K/yr
Best (simulated):  gamma=1.3  n_s=50  n_l=10  total=60  $1161.6K/yr
  P99 TTFT: short=80.9ms  long=392.0ms  SLO:PASS

gamma sweep (analytical, baseline=gamma=1.0 with 62 GPUs):
  gamma  n_s  n_l  total    $K/yr  saving   P99_s    P99_l
    1.0   49   13     62  $1200K   +0.0%   75.9ms  356.2ms
    1.1   50   12     62  $1200K   -0.0%   62.9ms  368.5ms
    1.2   50   11     61  $1181K   +1.6%   65.1ms  348.1ms
    1.3   50   10     60  $1162K   +3.2%   67.4ms  295.0ms  <-- best
    1.4   50   14     64  $1239K   -3.2%   68.2ms  471.6ms
    ...
```

Interpretation.

- You need **60 GPUs** (50 short, 10 long). A homogeneous fleet where all GPUs handle full 64K context would require ∼90 GPUs (see Case Study 4).
- Setting $\gamma = 1.3$ (compress borderline requests up to 30% over the boundary) saves 2 GPUs ($38K/yr) vs. plain pool routing at $\gamma = 1.0$.
- Both pools pass the 500 ms SLO comfortably: 81 ms and 392 ms respectively.
- The DES result matches the analytical prediction exactly (same fleet chosen), confirming the model is not overly optimistic.

Key Insight. The analytical sweep is the right starting point. DES verification is a safety check: if the analytical and DES choices agree, you can trust the analytical model for faster iteration. If they disagree, always trust the DES result.

6.2 Case Study 2: "Will My Fleet Survive Traffic Growth?"

Scenario. Your fleet runs at 200 req/s today but your growth model forecasts 300 req/s in Q3. You want to know the GPU count and annual cost at each point so you can plan procurement.

Command.

```
vllm-sr-sim whatif \
  --cdf data/azure_cdf.json \
  --slo 500 --b-short 6144 \
  --gpu-short a100 --gpu-long a100 \
  --lam-range 50 100 150 200 250 300
```

Output.

```
What-if analysis: sweep lambda over [50, 100, 150, 200, 250, 300]
B_short=6,144  SLO=500.0ms  GPU: A100-80GB/A100-80GB

  lambda  n_s  n_l  total    $K/yr   gamma*   P99_s    P99_l
  ----------------------------------------------------------
      50   13    5     18  $ 348.5K    1.1  266.9ms  384.5ms
     100   25    6     31  $ 600.1K    1.3  165.0ms  477.4ms
     150   37    9     46  $ 890.5K    1.2  110.0ms  364.6ms
     200   50   10     60  $1161.6K   1.3   67.4ms  295.0ms
     250   62   11     73  $1413.3K   1.3   53.8ms  432.1ms
     300   74   14     88  $1703.6K   1.2   42.0ms  476.4ms
```

Interpretation.

- GPU count scales sub-linearly with λ: from 18 GPUs at 50 req/s to 88 GPUs at 300 req/s (4.9× the GPUs for 6× the traffic). This is the Erlang-C "economy of scale" effect: larger fleets run at lower per-GPU utilization for the same SLO.

- Annual cost grows from \$349K to \$1.7M as traffic scales 6×. You can share this table directly with finance for infrastructure budget planning.
- The optimal γ^* fluctuates (1.1–1.3) because the Erlang-C discrete GPU counts create a staircase: at each λ, a slightly different γ minimizes the integer-rounded GPU count.
- The long pool consistently stays near the SLO limit (384–477 ms), while the short pool has plenty of headroom. If the SLO tightens to 300 ms, the long-pool sizing would become the bottleneck first.

Key Insight. The `whatif` output is a procurement plan. Use `-out growth_plan.json` to save the full results and feed them into your infrastructure cost model.

6.3 Case Study 3: "Which Router Should I Use for Agent Traffic?"

Scenario. Your workload is shifting toward agent-heavy traffic (long RAG contexts, tool calls). You have a fixed fleet of 60 short + 50 long GPUs at $\lambda = 100$ req/s and want to know whether switching from `LengthRouter` to `CompressAndRouteRouter` or `LeastLoadedRouter` improves P99 TTFT.

Command.

```
vllm-sr-sim compare-routers \
  --cdf data/agent_heavy_cdf.json \
  --lam 100 --slo 500 --b-short 16384 \
  --n-s 60 --n-l 50 \
  --gpu-short a100 --gpu-long a100
```

Output.

```
Router comparison  lambda=100 req/s  n_s=60 n_l=50  SLO=500.0ms
Router                          P99 TTFT     SLO%     Util
------------------------------------------------------------
LengthRouter                    1328.2ms   66.42%     2.5%
CompressAndRouteRouter          1306.4ms   66.16%     2.2%
RandomRouter                    2182.4ms   78.15%     3.1%
```

Interpretation.

- All three routers fail the 500 ms SLO at this fleet size for agent-heavy traffic. The issue is *not* the router choice — it is fleet under-provisioning. With only 110 GPUs at 100 req/s for this workload, utilization is low (2.5%) but P99 TTFT is 1.3 s. The bottleneck is **prefill time for long requests**, not queueing: agent-heavy requests with $L_{\text{in}} = 20{,}000$ tokens take $\lceil 20{,}000/512 \rceil \times 8\,\text{ms} \approx 312\,\text{ms}$ of prefill alone.
- `CompressAndRouteRouter` gives a small improvement (1.6% reduction in P99 TTFT) by redirecting borderline requests to the faster short pool.
- `RandomRouter` is significantly worse: it ignores token lengths and sends long requests to the short pool, causing KV-cache pressure and long queueing.
- **Action**: run `optimize` first to get the right fleet size, then use `compare-routers` to fine-tune the routing algorithm on the correctly-sized fleet.

Warning. A low utilization reading ($< 5\%$) does not mean the fleet is over-provisioned. For long-context workloads, P99 TTFT is dominated by **prefill time**, which is fixed by L_{in} and independent of utilization. You cannot reduce prefill time by adding GPUs; you can reduce it by lowering B_{short} or using C&R.

6.4 Case Study 4: "Homogeneous vs. Two-Pool: Is It Worth the Complexity?"

Scenario. Your team is debating whether to operate two separate GPU pools (short + long) versus one homogeneous fleet. The two-pool setup requires a router and separate queues. You want to quantify the cost savings to justify the operational complexity.

Approach. Run `optimize` on the homogeneous architecture first (by setting `-b-short 1` so all traffic routes to the long pool, which is configured for 64K context), then compare with the two-pool result.

Step 1: Find the minimum homogeneous fleet.

```
# All traffic goes to long pool (64K context, 16 slots/GPU)
vllm-sr-sim optimize \
  --cdf data/azure_cdf.json \
  --lam 200 --slo 500 --b-short 1 \
  --long-max-ctx 65536 \
  --gpu-short a100 --gpu-long a100
# Result: n_l=88 GPUs  $1,742K/yr  P99 queue wait=113.6ms
```

Step 2: Two-pool fleet (from Case Study 1):

```
vllm-sr-sim optimize \
  --cdf data/azure_cdf.json \
  --lam 200 --slo 500 --b-short 6144 \
  --gpu-short a100 --gpu-long a100
# Result: n_s=50 n_l=10  total=60  $1,162K/yr
```

Results summary.

Architecture	GPUs	Ann. cost	P99 queue wait	SLO
Homogeneous (64K context)	90	$1.74M	114 ms	✓
Two-pool (`b-short`=6K)	60	$1.16M	81 ms	✓
Saving	**30 GPUs**	**$581K/yr (33%)**	—	—

Interpretation. The two-pool architecture saves **33% of GPU cost** ($581K/yr at $2.21/GPU-hr) by routing the ∼97.6% of requests below 6K tokens to the efficient short pool (8× throughput per GPU vs. long pool).

Warning. The table reports *P99 queue wait*, which is what the analytical model and the optimizer's fast DES measure. If you run `simulate` with the two-pool fleet, the *full* P99 TTFT will be ∼912 ms — exceeding the 500 ms SLO — because for $B_{\text{short}} = 6{,}144$-token requests, prefill time alone is $\lceil 4{,}915/512 \rceil \times 91.2\,\text{ms} \approx 912\,\text{ms}$. This is the prefill-dominated regime described in Section 2.4: the SLO budget is consumed by compute, not queueing. To fit a 500 ms wall-clock SLO, set $B_{\text{short}} \leq 2{,}048$ or account for prefill time when interpreting the optimizer's output.

The operational complexity of a second pool and a router is amortised in under two weeks of GPU cost savings at this scale.

6.5 Case Study 5: "Semantic Router — Sizing a Two-Model Fleet"

Scenario. You are deploying a semantic router that classifies incoming Azure-like traffic (200 req/s, 500 ms P99 TTFT SLO) into two streams:

- **Simple queries** (70% of traffic, 140 req/s) routed to a cost-efficient `llama8b` model on A10G GPUs (max context 4K).

- **Complex queries** (30% of traffic, 60 req/s) routed to a high-capability `llama70b` model on A100-80GB GPUs (max context 8K).

You want to determine the right GPU count for each pool and quantify the cost savings versus a homogeneous A100 fleet that serves all traffic with the large model.

Step 1: Optimize the small-model pool (A10G, 140 req/s).

```
# All traffic routed to single short pool (b_short=65536 >> any request)
vllm-sr-sim optimize \
  --cdf data/azure_cdf.json --lam 140 --slo 500 \
  --b-short 65536 --gpu-short a10g --gpu-long a10g
# Best (simulated): n_s=129 n_l=0  total=129  $1,141K/yr
#   P99 TTFT: short=114.7ms  SLO: pass
```

Step 2: Optimize the large-model pool (A100, 60 req/s).

```
vllm-sr-sim optimize \
  --cdf data/lmsys_cdf.json --lam 60 --slo 500 \
  --b-short 65536 --long-max-ctx 8192 \
  --gpu-short a100 --gpu-long a100
# Best (simulated): n_s=13  n_l=0  total=13  $252K/yr
#   P99 TTFT: short=253.8ms  SLO: pass
```

Step 3: Homogeneous A100 baseline (no semantic routing).

```
vllm-sr-sim optimize \
  --cdf data/azure_cdf.json --lam 200 --slo 500 \
  --b-short 65536 --gpu-short a100 --gpu-long a100
# Best (simulated): n_s=88  n_l=0  total=88  $1,704K/yr
#   P99 TTFT: short=84.7ms  SLO: pass
```

Results.

Configuration	GPUs	Ann. cost	P99 TTFT	SLO
Homogeneous A100 (large model only)	88 A100	$1.70M	84.7 ms	✓
Semantic: `llama8b` pool (A10G)	129 A10G	$1.14M	114.7 ms	✓
Semantic: `llama70b` pool (A100)	13 A100	$0.25M	253.8 ms	✓
Semantic fleet total	142	**$1.39M**	—	✓
Saving vs. homogeneous	+54 GPUs	**$311K/yr (18%)**	—	—

Interpretation. Despite requiring 54 *more* GPUs in total, the semantic fleet costs 18% less because A10G GPUs ($1.01/GPU-hr) are less than half the price of A100s ($2.21/GPU-hr), and 70% of traffic can be served by the cheaper hardware. The large-model pool (13 A100s) handles only the 30% of complex traffic, keeping utilisation comfortable.

Key Insight. In the semantic-routing scenario the cost saving comes from **hardware heterogeneity**, not from increased GPU utilisation. Use the `-gpu-short` / `-gpu-long` flags in each `optimize` call to model your actual GPU mix. Then replay a production access log (Section 8) to validate that the routing distribution matches the CDF assumption.

6.6 Case Study 6: "How Much Power Can I Offer the Grid?"

Scenario. Your data center participates in a utility demand-response (DR) programme. On short notice (15–60 minutes), the grid operator can request a power reduction of up to 30%. You run 30 H100-80GB GPUs serving an Azure-like workload at $\lambda = 300$ req/s with a P99

TTFT SLO of 200 ms. You need to know: how deep a curtailment commitment can you sign up for without risking an SLO breach?

Mechanism. The GPU-to-Grid (G2G) framework [Hassan et al., 2025] shows that capping `max_num_seqs` in vLLM (the maximum in-flight batch size) is the most effective software knob for modulating GPU power draw. A lower cap reduces memory-bandwidth pressure and therefore power, at the cost of increased queuing delay. The `grid-flex` command models this trade-off by sweeping the batch-size cap and computing the resulting P99 TTFT via the same M/G/c approximation used during optimization.

```
vllm-sr-sim grid-flex \
  --cdf data/azure_cdf.json \
  --lam 300 --n-gpus 30 --gpu h100 --slo 200
```

Flex	$n_{\max}$	W/GPU	Fleet kW	P99 TTFT	SLO
0%	128	450 W	13.5 kW	7.9 ms	✓
5%	115	435 W	13.0 kW	7.9 ms	✓
10%	102	420 W	12.6 kW	7.9 ms	✓
15%	90	405 W	12.2 kW	7.9 ms	✓
20%	77	390 W	11.7 kW	7.9 ms	✓
25%	**64**	**375 W**	**11.2 kW**	**1058 ms**	×
30%	51	360 W	10.8 kW	∞	×

Interpretation. The fleet can commit 20% power reduction (saving 1.8 kW fleet-wide) with essentially no latency impact: P99 TTFT stays at 7.9 ms, well inside the 200 ms SLO. Stepping from 20% to 25% reduces $n_{\max}$ from 77 to 64 concurrent requests per GPU, crossing the Erlang-C saturation threshold where $\lambda/\mu > n_{\max}$ and the queue diverges. P99 TTFT jumps from 7.9 ms to 1,058 ms — a 133× degradation from a single-step change.

Power model. The `H100_80GB` profile uses `power_idle_w = 300 W` and `power_nominal_w = 600 W`, sourced from ML.ENERGY Benchmark v3.0 [Chung et al., 2025] (measured H100-SXM5 running vLLM on Llama-3.1 variants) and the NVIDIA H100-SXM5 TDP spec (700 W). The per-GPU power at $n_{\max}$ concurrent requests is linearly interpolated: $P(n) = P_{\text{idle}} + (P_{\text{nominal}} - P_{\text{idle}}) \cdot n/n_{\max,\text{full}}$.

Key Insight. The safe demand-response commitment has a **hard cliff** at the Erlang-C saturation point. Below that threshold, one more 5% step takes P99 TTFT from under 10 ms to over 1,000 ms. `grid-flex` locates this cliff analytically, before any hardware change or serving engine modification.

7 Multi-Model and Arbitrary N-Pool Fleets

7.1 When to Use This Mode

The `optimize`, `simulate`, and `whatif` commands assume a **two-pool, length-partitioned fleet**: a short-context pool and a long-context pool, with routing driven by token count. This covers the common case where KV-cache pressure is the dominant cost driver.

However, a production GPU fleet may have a different structure entirely:

- **Multi-model**: separate pools for Llama-70B, Llama-8B, and a code-specialist model, each receiving its own traffic stream. Routing is based on which model a request targets, not prompt length.
- **Hardware tiers without context differentiation**: A100 pools for latency-sensitive requests, A10G pools for batch/background tasks, regardless of token count.

- **Canary / shadow pools**: a small experimental pool receiving a fraction of traffic alongside the main fleet.

7.2 The `simulate-fleet` Command

```
vllm-sr-sim simulate-fleet fleet.json \
  --lam 200 --slo 500 --n-req 30000
```

The `fleet.json` file defines pools, router, and (for `ModelRouter`) per-pool workload streams:

```
{
  "pools": [
    {"id": "llama70b",  "gpu": "a100", "n_gpus": 20, "max_ctx": 8192},
    {"id": "llama8b",   "gpu": "a10g", "n_gpus": 8,  "max_ctx": 4096},
    {"id": "codellama", "gpu": "a100", "n_gpus": 6,  "max_ctx": 16384}
  ],
  "router": "model",
  "workloads": {
    "llama70b":  {"cdf": "data/azure_cdf.json",      "lam_frac": 0.60},
    "llama8b":   {"cdf": "data/lmsys_cdf.json",      "lam_frac": 0.25},
    "codellama": {"cdf": "data/agent_heavy_cdf.json", "lam_frac": 0.15}
  }
}
```

Supported `router` values: `"length"` (default), `"model"`, `"random"`, `"least_loaded"`.

7.3 How `ModelRouter` Works

Each request carries an optional `model_id` string (`Request.model_id`). When the workloads section is present, the simulator tags each request with its pool's `id` before merging the per-pool arrival streams. `ModelRouter` reads `request.model_id` and routes to the matching pool; requests with `model_id=None` or an unknown id fall back to the first (default) pool.

You can also set `model_id` programmatically in a custom workload generator to implement any tagging scheme (e.g., per-user model preference, content-type based routing).

7.4 Semantic Routing: Large/Small Model Selection

A semantic router classifies each request at the gateway and dispatches it to the most appropriate model pool (e.g., a small 8B model for simple queries, a large 70B model for complex reasoning). The simulator supports this fully — see Section 8 for the complete reference, including trace replay from production routers, offline policy comparison, field mapping, and Case Study 5.

7.5 Example Output

Running the example three-model fleet at 200 req/s:

```
Simulating 20,000 requests across 3 pools ...
  Pools: llama70b(20xA100-80GB,ctx=8192), llama8b(8xA10G,ctx=4096),
         codellama(6xA100-80GB,ctx=16384)
  Router: ModelRouter

  Fleet summary  (SLO = 500ms P99 TTFT)
  Total GPUs                    : 34
  Annualised cost               : $574.1K/yr
  Fleet P99 TTFT                : 1587.2ms
  Fleet P50 TTFT                : 182.4ms
  SLO compliance                : 86.17%
  Mean utilisation              : 6.4%
```

```
Pool 'llama70b': GPUs=20  P99 TTFT=1094.4ms  Util=0.7%  SLO=87.2%
Pool 'llama8b':  GPUs=8   P99 TTFT=1183.2ms  Util=1.8%  SLO=95.6%
Pool 'codellama':GPUs=6   P99 TTFT=2529.6ms  Util=16.7% SLO=66.1%
```

The SLO failures here are *prefill-dominated* (queue wait ≈ 0 ms; utilisation is low). Use `optimize` on each pool's workload independently to find the right GPU count per pool, then re-run `simulate-fleet` to validate.

8 Semantic Router Integration

8.1 Concept and Pipeline

A **semantic router** sits at the API gateway and classifies each incoming request before it reaches any GPU. The classification can be based on embedding similarity, an intent classifier, keyword rules, or a lightweight LLM judge. The decision determines which model (and therefore which pool of GPUs) handles the request.

```
Client request
     |
     v
Semantic Router  (e.g., vLLM semantic router, custom classifier)
+-------------------------------------+
|  classify(prompt) -> "llama8b"      |
|                   -> "llama70b"     |
|  logs: timestamp, tokens,           |
|        selected_model, complexity   |
+-------------------------------------+
     |                    |
     v                    v
llama8b pool          llama70b pool
(A10G GPUs)           (A100 GPUs)
simple / short        complex / long
context queries       context queries
```

The simulator can reproduce this two-stage pipeline at fleet scale, letting you answer:

- How many GPUs does each pool need to meet the SLO at production load?
- Does my current routing split leave one pool over-provisioned?
- What happens to P99 latency if the semantic router shifts 10% more traffic to the large model?
- How much does classifier latency eat into my SLO budget?

8.2 Classifier Types and Log Fields

Different semantic routers emit different log formats. The table below shows common classifier types and the routing-decision field each typically writes to its access log.

Classifier type	Examples	Typical log field	Typical values
Complexity scorer	vLLM semantic router	`x_vsr_selected_model`	model name
Intent classifier	custom NLP pipeline	`selected_model`	model name
OpenAI-compatible	API proxy logs	`model`	model name
Embedding similarity	semantic-router lib	`routed_to`	pool id
Keyword rules	simple gateway	`model`	model name
Generic label	any custom system	user-defined	any string

8.3 Field Mapping Reference

Pass `model_id_field` to `TraceWorkload` to match your log's routing column. Any string value is accepted; it becomes the `Request.model_id` that `ModelRouter` uses for dispatch.

Router / system	`model_id_field` value	Notes
vLLM semantic router	`"x_vsr_selected_model"`	HTTP response header
Custom gateway (default)	`"selected_model"`	simulator default
OpenAI-compatible proxy	`"model"`	from request or response body
Generic label column	`"routed_to"`	common convention
Any custom field	`"<your_field_name>"`	any JSONL/CSV column

8.4 Workflow A: Replay a Pre-Labeled Production Trace (Recommended)

If your semantic router is already running in production, export its access log and replay the routing decisions in the simulator. This is the most accurate approach because it uses the *actual* routing distribution from real traffic — no assumptions about the split ratio, no re-running the classifier.

Expected JSONL format. Each line is a JSON object with at minimum `timestamp` (seconds since start), token counts, and the routing decision field:

```
{"timestamp": 0.005, "prompt_tokens": 512,  "generated_tokens": 64,
 "selected_model": "llama70b", "complexity": "high", "category": "reasoning"}
{"timestamp": 0.010, "prompt_tokens": 128,  "generated_tokens": 32,
 "selected_model": "llama8b",  "complexity": "low",  "category": "factual"}
{"timestamp": 0.018, "prompt_tokens": 2048, "generated_tokens": 256,
 "selected_model": "llama70b", "complexity": "high", "category": "summarisation"}
```

The `complexity` and `category` fields are optional but are stored on the request object (`req.category`, `req._complexity`) for post-simulation segment analysis.

Loading and running.

```
from fleet_sim.workload.trace import TraceWorkload
from fleet_sim.core.fleet import Fleet, FleetConfig, PoolConfig
from fleet_sim import A100_80GB, A10G

# Load the trace: routing decisions become Request.model_id
wl = TraceWorkload(
    path="router_access_log.jsonl",
    fmt="semantic_router",
    model_id_field="selected_model",   # or "x_vsr_selected_model", "model", ...
)
arrivals = wl.generate()

# Build a fleet that honours the recorded routing decisions
fc = FleetConfig(
    pools=[
        PoolConfig("llama70b", A100_80GB, 13, 8192),
        PoolConfig("llama8b",  A10G,     129, 4096),
    ],
    router_type="ModelRouter",   # dispatches by Request.model_id
)
fleet = Fleet(fc)
result = fleet.run(arrivals)
result.print_summary(t_slo_ms=500)
```

Non-standard field names. Use the `field_map` parameter to remap any column:

```
wl = TraceWorkload(
    path="vllm_access.jsonl",
    fmt="semantic_router",
    model_id_field="x_vsr_selected_model",   # vLLM semantic router header
    field_map={
        "timestamp":        "request_time",   # rename column 'request_time' ->
    timestamp
        "prompt_tokens":    "input_tokens",
        "generated_tokens": "output_tokens",
    },
)
```

See `examples/semantic_router_trace_replay.py` for a runnable demo that also generates a synthetic trace for testing.

8.5 Workflow B: Simulate a Routing Policy Offline

Before deploying a new routing policy, simulate it offline to compare fleet cost and SLO compliance under alternative strategies. Supply a Python `classify_fn(Request)` → `pool_id` to `SemanticRouter`:

```
from fleet_sim.routing.semantic_router import SemanticRouter
from fleet_sim.core.fleet import Fleet, FleetConfig, PoolConfig
from fleet_sim import A100_80GB, A10G

def my_classifier(req):
    # Any logic: embedding lookup, intent model, keyword rule, etc.
    # req.l_total = prompt_tokens + generated_tokens
    return "llama8b" if req.l_total <= 2048 else "llama70b"

fc = FleetConfig(
    pools=[
        PoolConfig("llama70b", A100_80GB, 13, 8192),
        PoolConfig("llama8b",  A10G,     129, 4096),
    ],
    router_type="SemanticRouter",
    router_kwargs={"classify_fn": my_classifier},
)
fleet = Fleet(fc)
result = fleet.run(arrivals)
result.print_summary(t_slo_ms=500)
```

Comparing policies. `examples/semantic_routing.py` runs a three-way comparison — length-threshold heuristic, fixed-fraction split, and all-large baseline — and prints a summary table showing cost and SLO compliance for each.

8.6 Accounting for Classifier Latency

The semantic classifier itself takes time (5–50 ms for an embedding lookup; up to 150 ms for a small LLM judge). This latency is paid by every request and is part of the user-perceived end-to-end TTFT. Budget for it by tightening the SLO passed to the fleet optimizer:

$$\text{SLO}_{\text{fleet}} = \text{SLO}_{\text{end-to-end}} - t_{\text{classifier}}$$

Example: end-to-end SLO = 500 ms, embedding classifier $t \approx 20$ ms $\Rightarrow$ pass `-slo 480` to `optimize`.

> **Warning.** The two-stage pipeline also adds a *head-of-line blocking* risk: if the classifier is slow (>50 ms) or its throughput is limited, it can create a queue before the GPU pools. The simulator does *not* model classifier throughput; model the classifier purely as a latency offset via the adjusted SLO above.

8.7 Using `simulate-fleet` with Semantic Routing

For full end-to-end validation including per-pool GPU counts, create a JSON fleet config and pass `"router": "semantic"` (uses `SemanticRouter`) or `"router": "model"` (uses `ModelRouter` to replay trace decisions):

```
{
  "pools": [
    {"id": "llama70b", "gpu": "a100", "n_gpus": 13, "max_ctx": 8192},
    {"id": "llama8b",  "gpu": "a10g", "n_gpus": 129, "max_ctx": 4096}
  ],
  "router": "model",
  "workloads": [
    {"pool": "llama70b", "cdf": "data/lmsys_cdf.json", "lam_frac": 0.30},
    {"pool": "llama8b",  "cdf": "data/azure_cdf.json", "lam_frac": 0.70}
  ]
}
```

```
vllm-sr-sim simulate-fleet semantic_fleet.json \
  --lam 200 --slo 500 --n-req 20000
```

9 Extending the Simulator

9.1 Adding a Custom GPU Profile

Edit `fleet_sim/gpu_profiles/profiles.py`:

```
from .profiles import GpuProfile

# H200-80GB (hypothetical)
H200_80GB = GpuProfile(
    name="H200-80GB",
    W_ms=3.2,           # baseline iteration latency (ms)
    H_ms_per_slot=0.28, # per-slot memory-bandwidth cost (ms)
    n_slots_short=128,  # concurrent sequences at short context
    n_slots_long=16,    # concurrent sequences at long context
    max_ctx_short=8192,
    max_ctx_long=65536,
    cost_per_hr=4.50,   # on-demand cost ($/hr)
)
```

Then pass `gpu_short=H200_80GB` to `FleetOptimizer` or use `-gpu-short custom` after registering the profile.

9.2 Adding a Custom Routing Algorithm

Subclass `BaseRouter` in `fleet_sim/routing/base.py`:

```python
from fleet_sim.routing.base import BaseRouter
from fleet_sim.core.request import Request
from fleet_sim.core.pool import Pool
from typing import Optional

class TightBudgetRouter(BaseRouter):
    """Route very short requests (<512 tokens) to short pool;
    all others to long pool (conservative for latency-critical workloads)."""

    def __init__(self, pools, B_short: int, tight_threshold: int = 512):
        super().__init__(pools, B_short)
        self.tight_threshold = tight_threshold

    def route(self, request: Request) -> Optional[Pool]:
        # Note: Request has fields l_in, l_out (no priority field).
        # l_total is a property = l_in + l_out.
        if request.l_total <= self.tight_threshold:
            return self.pools["short"]
        return self.pools["long"]
```

Register the router in `fleet_sim/routing/__init__.py` and pass it to `FleetConfig.router`.

9.3 Adding a Custom Workload Source

Implement the `WorkloadGenerator` protocol:

```python
from fleet_sim.workload.synthetic import WorkloadGenerator
from fleet_sim.core.request import Request
import random

class BimodalWorkload(WorkloadGenerator):
    """50% short requests (500 tokens) + 50% long (20K tokens)."""

    def sample_request(self, rng: random.Random) -> Request:
        if rng.random() < 0.5:
            l_in, l_out = 400, 100
        else:
            l_in, l_out = 18000, 2000
        return Request(l_in=l_in, l_out=l_out)
```

9.4 Running the Optimizer Programmatically

```python
import json
from fleet_sim import FleetOptimizer, A100_80GB
from fleet_sim.workload.synthetic import CdfWorkload

# Load CDF
cdf = json.load(open("data/azure_cdf.json"))

# Run optimizer
optimizer = FleetOptimizer(
    gpu_short=A100_80GB,
    gpu_long=A100_80GB,
    B_short=6144,
    t_slo_ms=500,
)
```

```
result = optimizer.optimize(
    cdf=cdf,
    lam=200,
    gammas=[1.0, 1.1, 1.2, 1.3, 1.5],
)
result.print_report()

# Access fields (best_simulated falls back to best_analytical if DES skipped)
best = result.best_simulated or result.best_analytical
print(f"n_s={best.n_s}, n_l={best.n_l}")
print(f"gamma*={best.gamma}")
print(f"Annual cost: ${best.annualised_cost_kusd:.0f}K")
```

10 Limitations and Scope

Single-model, single-hardware per pool. The simulator assumes all GPUs in a pool run the same model with the same hardware profile. Multi-model pools (e.g., Llama-70B in short, Llama-7B in long) are not modelled.

Static routing threshold. B_{short} is fixed for the duration of a simulation run. Adaptive threshold adjustment based on real-time queue depths is not implemented.

Poisson arrivals. The arrival process is Poisson (memoryless inter-arrival times). Bursty or correlated arrivals (e.g., batch API calls) are not modelled. The analytical P99 estimates are conservative for Poisson arrivals but may underestimate tail latency under burstiness.

Poisson sub-stream approximation (pool routing and C&R). The optimizer decomposes the aggregate arrival rate as $\lambda_s = \alpha\lambda$ and $\lambda_l = (1-\alpha)\lambda$ and treats each sub-stream as an independent Poisson process. By the Poisson thinning theorem, this is exact only when each arrival is routed by an independent Bernoulli trial with fixed probability α. Length-based routing is deterministic on L_{total}, so the sub-streams are correlated through the shared length distribution and are not strictly Poisson. Analytical queue lengths and P99 TTFT estimates are therefore approximations. The error is small when request lengths are drawn i.i.d. and independently of inter-arrival times — the DES verification step provides an empirical check. For workloads with strong temporal autocorrelation in request length (e.g., sessions that systematically alternate between short and long requests), the approximation error grows and a DES-only result should be used.

Prefill chunking is simplified. The simulator uses a fixed chunk size C_{chunk} for all requests. In practice, vLLM's chunked prefill uses dynamic chunk sizes based on available KV cache blocks. This may cause 10–20% discrepancy in prefill time estimates for very long requests.

KV-cache eviction not modelled. The simulator assumes sufficient KV-cache capacity. Under extreme load, real engines evict KV-cache blocks (causing re-computation), which would increase TTFT unpredictably. The $\rho_{\max} = 0.85$ cap provides a practical buffer against this regime.

Compression fidelity. The C&R router assumes $p_c = 1.0$ (all borderline requests are successfully compressible). In practice, code and structured data may not compress below B_{short}. A per-category p_c can be passed to `CompressAndRouteRouter` to model realistic compression rates.

Linear GPU power model. The `grid-flex` power model linearly interpolates between `power_idle_w` and `power_nominal_w`. Empirical measurements (ML.ENERGY Benchmark v3.0 [Chung et al., 2025]) show the true power-vs-batch curve is logistic: sub-linear at high batch sizes and super-linear near idle. The linear model is accurate to within ±5% in the 40–100% load range operationally relevant for demand-response programmes. For tighter accuracy,

re-fit `power_idle_w` and `power_nominal_w` from measured profiling runs on your specific GPU and model.

11 Relationship to Companion Research

`vllm-sr-sim` sits at the intersection of several active research threads on LLM serving systems.

Companion analytical work.

- **Pool routing** [Chen et al., 2026c]: Establishes the two-pool architecture and the GPU savings formula $\alpha(1 - 1/\rho)$. The simulator provides fleet-level DES to validate those savings at realistic arrival rates.
- **Compress-and-Route (C&R)** [Chen et al., 2026a]: Introduces gateway-layer prompt compression as a routing mechanism, diverting borderline requests from the long pool to the short pool. `CompressAndRouteRouter` implements this algorithm.
- **FleetOpt** [Chen et al., 2026b]: Derives the minimum-cost fleet analytically using M/G/c queuing and shows that the analytical model overestimates P99 queue wait by 8–13% (conservative bias). The `FleetOptimizer` class implements the two-phase algorithm from that work.

Grid demand response and power management.

- **GPU-to-Grid (G2G)** [Hassan et al., 2025]: Demonstrates that capping `max_num_seqs` in vLLM is the primary software knob for modulating H100 GPU power consumption. Power follows a logistic curve as a function of $\log_2$(batch size), measured empirically via the ML.ENERGY Benchmark v3.0 on H100-SXM5 hardware [Chung et al., 2025]. The `grid_flex_analysis()` function and `grid-flex` CLI command implement this mechanism at fleet level: given a fixed GPU count and arrival rate, they compute the maximum power curtailment depth that still meets the P99 TTFT SLO.

Fleet configuration and heterogeneous GPU allocation.

- **AIConfigurator** [Xu et al., 2025]: Performs bottom-up roofline composition from silicon-measured GEMM, attention, MoE, and NCCL kernel benchmarks to auto-configure TP/EP degrees for disaggregated clusters. `vllm-sr-sim` embeds three key results: (i) memory bandwidth scaling factor $\alpha_B = 0.80$; (ii) MoE kernel latency tables measured on H100 SXM for DeepSeek-V3 and Qwen3; (iii) disaggregated-serving degradation constants $\alpha_{\text{pre}} = 0.90$, $\alpha_{\text{dec}} = 0.92$, $\beta_{\text{TTFT}} = 1.80$. All data are embedded directly in the `fleet_sim` packages with no runtime dependency on the AIConfigurator codebase.
- **Mélange** [Griggs et al., 2024]: Formulates heterogeneous GPU allocation as cost-aware bin packing, showing that request size, rate, and SLO jointly determine the optimal GPU mix. The hardware catalog in `fleet_sim.hardware` enables the same cross-GPU comparisons with a simulator-in-the-loop cost model.
- **SageServe** [Jia et al., 2025]: Combines ILP-based routing with forecast-aware VM auto-scaling, saving 25% GPU-hours on Microsoft Office 365 at 10M+ daily requests. Complements this simulator's static sizing analysis with a production auto-scaling policy layer.

Disaggregated prefill/decode serving.

- **DistServe** [Zhong et al., 2024]: The foundational disaggregated serving paper, achieving 7.4× more requests or 12.6× tighter SLO by routing prefill and decode to separate GPU pools with independent parallelism. `DisaggFleetOptimizer` models the same separation at fleet scale.
- **Splitwise** [Patel et al., 2024]: Co-designs hardware topology and scheduling policy for PD-split clusters, including ML-based decode-length prediction for optimal worker assignment.

- **TokenScale** [Dong et al., 2024]: Introduces Token Velocity as a leading-indicator metric for proactive autoscaling of disaggregated fleets, improving SLO attainment from 50–88% to 80–96%.

LLM inference simulation.

- **Vidur** [Agrawal et al., 2024]: A high-fidelity single-instance DES covering KV-cache block allocation, chunked prefill scheduling, and preemption. `vllm-sr-sim` is complementary: Vidur tunes a single engine; this simulator sizes the fleet and evaluates routing policies across many engines.

The simulator serves as an independent ground-truth check for analytical models. A key finding from DES validation: the long-pool service rate must be recalibrated for the post-compression request distribution when estimating C&R's provisioning benefit — without this step, the model overestimates savings. This underscores the value of a runnable simulator as a complement to any analytical framework.

12 Quick Reference Card

Question	Command / Section
How many GPUs do I need?	`optimize -cdf ... -lam N -slo M`
Will my current fleet meet SLO?	`simulate -n-s N -n-l M ...`
How do I scale with traffic growth?	`whatif -lam-range 50 100 200 ...`
Which router is best?	`compare-routers -n-s N -n-l M ...`
A100 vs H100 for short pool?	Run `optimize` twice with different `-gpu-short`
Worth splitting into two pools?	Compare `optimize` vs `simulate` with `-b-short 65536`
Validate my analytical estimate	Cross-check `simulate` P99 against analytical from `optimize`
Size a semantic-router fleet	`optimize` per pool at its traffic fraction; see Section 6.5
Replay production router log	`TraceWorkload(fmt="semantic_router")`; see Section 8.4
Test a new routing policy offline	`SemanticRouter(classify_fn=...)`; see Section 8.5
Build a physics-informed profile	`ProfileBuilder().build(hw, model, cfg)`; see Section 4.5
Size a disaggregated fleet	`DisaggFleetOptimizer(...).optimize(...)`; see Section 4.6
Safe power curtailment for grid DR	`grid-flex -n-gpus N -lam L -slo S`; see Section 5.5 and Case Study 6.6

Physics-informed profile quick reference.

Task	Code / Call
Look up a GPU spec	`from fleet_sim import get_hardware; get_hardware("h100")`
Look up a model spec	`from fleet_sim import get_model; get_model("llama-3.1-70b")`
Build a computed profile	`ProfileBuilder().build(hw, model, cfg)`
Inspect profile	`profile.summary()`
List available GPUs	`fleet_sim.hardware.list_hardware()`
List available models	`fleet_sim.models.list_models()`
Use FP8 quantization	`ServingConfig(dtype_bytes=1, ...)`
Disaggregated sizing	`DisaggFleetOptimizer(...).optimize(...)`

Key parameters cheat sheet.

Parameter	Typical range	Effect
`-b-short`	2K–16K tokens	CDF knee; higher = more long-pool
`-slo`	100–2000 ms	Tighter = more GPUs needed
`-gamma-max`	1.1–2.0	C&R compression range to explore
$\rho_{\max}$ (internal)	0.85	Utilization cap for stability
C_{chunk} (internal)	512	Prefill chunk size
`mean_ctx_tokens`	512–8192	Representative context for H calc
α_{pre} (disagg)	0.90	Prefill throughput degradation
α_{dec} (disagg)	0.92	Decode throughput degradation
β_{TTFT} (disagg)	1.80	TTFT correction for KV transfer

Comparison with related tools.

Aspect	Vidur [Agrawal et al., 2024]	AIConf. [Xu et al., 2025] / Mélange [Griggs et al., 2024]	vLLM Semantic Router Fleet Sim
Scope	One engine	One cluster / GPU-type selection	Whole fleet (N pools)
Service model	High-fidelity scheduling	Disaggregated only	Aggregated + disaggregated
Routing	—	—	Length, semantic, C&R, model
Performance model	Kernel-level trace	Roofline + kernel DB	Roofline + MoE kernel table
Optimization	—	TP×EP / bin packing	Min-cost fleet (Erlang-C + DES)
Validation	Trace replay	Kernel benchmarks	Discrete-event M/G/c
Dependencies	vLLM trace	Dynamo / custom solver	Self-contained Python

References

Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale simulation framework for LLM inference, 2024. URL https://arxiv.org/abs/2405.05465. Single-instance high-fidelity DES covering KV-cache, scheduling, and preemption; complementary scope to this fleet-level simulator.

Huamin Chen, Xunzhuo Liu, Junchen Jiang, Bowei He, and Xue Liu. Compress-and-route: Routing-layer prompt compression against the long-context cost cliff in LLM inference fleets. *arXiv preprint*, 2026a. Manuscript under review.

Huamin Chen, Xunzhuo Liu, Junchen Jiang, Bowei He, and Xue Liu. FleetOpt: Workload-CDF-aware LLM GPU fleet provisioning via co-design of pool boundaries and compression policy. *arXiv*, 2026b.

Huamin Chen, Xunzhuo Liu, Junchen Jiang, Bowei He, and Xue Liu. Token-budget-aware pool routing for cost-efficient LLM inference. *arXiv preprint*, 2026c. Manuscript under review.

Jae-Won Chung, Woosuk Kwon, and Ion Stoica. LLM-Pilot: Characterize and optimize inference of LLMs on dedicated GPU servers. In *Proc. NeurIPS Datasets and Benchmarks Track*, 2025. URL https://ml.energy/leaderboard. ML.ENERGY Benchmark v3.0: measured H100-SXM5 GPU power vs. batch size (1–256) running vLLM on Llama-3.1 variants. Idle (batch≈1) ≈300 W; saturated (batch=128) ≈600 W for 70B-class model.

Chenhe Dong et al. TokenScale: Timely and accurate autoscaling for disaggregated LLM serving with token velocity. *arXiv:2512.03416*, 2024. URL https://arxiv.org/abs/2512.03416. Token Velocity as a leading indicator for proactive PD scaling; improves SLO attainment from 50–88% to 80–96%.

Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. Mélange: Cost efficient large language model serving by exploiting GPU heterogeneity. *arXiv:2404.14527*, 2024. URL https://arxiv.org/abs/2404.14527. Formulates heterogeneous GPU allocation as cost-aware bin packing; up to 77% cost reduction vs. single GPU-type deployments.

M. Hassan, J. Lin, D. Kim, R. Bhatt, Y. Wang, N. Li, and S. Grijalva. GPU-to-Grid: Coupling LLM inference with power system control. *arXiv:2602.05116*, 2025. URL https://arxiv.org/abs/2602.05116. Shows vLLM max_num_seqs is the primary GPU power knob; power follows a logistic curve vs. log2(batch size); data from ML.ENERGY Benchmark on H100-SXM5.

Ningxin Jia et al. SageServe: Optimizing LLM serving on cloud data centers with forecast aware auto-scaling. *arXiv:2502.14617*, 2025. URL https://arxiv.org/abs/2502.14617. ILP-based routing + VM auto-scaling for 10M+ daily requests; 25% GPU-hour savings at Microsoft Office 365.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Brijesh Warrier, Nithish Mahalingam, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *Proc. ISCA*, 2024.

Tianhao Xu, Yiming Liu, Xianglong Lu, Yijia Zhao, Xuting Zhou, Aichen Feng, Yiyi Chen, et al. AIConfigurator: Lightning-fast configuration optimization for multi-framework LLM serving. *arXiv:2601.06288*, 2025. URL https://arxiv.org/abs/2601.06288. Kernel-level performance DB (GEMM/attn/MoE/NCCL) across Ampere/Hopper/Blackwell; source of calibration constants in `fleet_sim.hardware`.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proc. OSDI*, 2024. URL https://arxiv.org/abs/2401.09670. Founding paper of PD disaggregation; 7.4× more requests or 12.6× tighter SLO vs. colocated serving.