Game-Theoretic Analysis of NVIDIA Dynamo on 8×B200: Research & Execution Plan

Author: Athrael **Hardware**: 8× NVIDIA B200 SXM (192 GB HBM3e each) **Goals**:

1. Produce a vLLM cookbook for NVIDIA Dynamo
2. Contribute B200 benchmarks to vLLM (tracking issue #28883)
3. Release an extendable paper with game-theoretic analysis (Nash equilibrium + Pareto optimality)

Working title: *"Are Inference Serving Systems Playing Games? A Game-Theoretic Analysis of Multi-Component LLM Serving on NVIDIA Dynamo"*

Table of Contents

---

1. Literature Gap & Prior Work

The gap is confirmed and clean. No published work applies game theory to model internal component interactions of an LLM inference serving system. The game theory + LLM intersection exists but in entirely different contexts: LLMs playing strategic games (IJCAI-25 survey), game-theoretic alignment (GTAlign), and attention mechanism aggregation (MAHA). None touch serving infrastructure.

All current inference serving systems (vLLM, SGLang, Orca, Mooncake, DistServe, Dynamo) use centralized schedulers. The literature frames component interactions as joint optimization problems (ILP, MDP, heuristic search), not as equilibrium problems between agents with conflicting objectives. Yet the tensions are well-

documented: scheduler wants large batches but memory constrains them; router wants cache affinity but load balancer wants even distribution; autoscaler wants cost efficiency but scheduler wants headroom. These are implicit games that nobody has formalized.

5 Most Directly Usable Papers

1. Congestion game for microservices — Luo et al., CollaborateCom 2018

- Models microservices competing for VMs using M/M/1 queuing with revenue functions based on response time
- Congestion game structure guarantees Nash equilibrium existence via Rosenthal's potential argument
- Directly applicable to modeling inference request routing as a congestion game
- **Limitation**: M/M/1 assumption breaks down for GPU batch inference where latency has step-function behavior at batch boundaries

2. Three-tier congestion game for cloud economics — Anselmi et al., TOMPECS 2017

- Models users → SaaS → IaaS as a multi-tier game with Price of Anarchy = 5/2 for linear congestion costs
- Three-tier structure maps naturally to requests → inference service → GPU cluster
- Key finding: adding infrastructure providers can *increase* PoA — directly relevant to disaggregated serving where adding prefill workers can increase overall latency through coordination overhead

3. Generalized Nash Equilibrium for cloud service provisioning — Ardagna et al., IEEE TSC 2013

- Handles shared resource constraints via variational inequality reformulation
- Exactly the scenario where router, planner, and KV manager share a fixed GPU memory budget
- M/G/1 processor-sharing model can be adapted for continuous batching

4. Mean-field game for service function chaining — Abouaomar et al., IEEE Systems Journal 2022

- Scales to many VNFs using mean-field approximation with two-phase placement+scheduling decomposition
- Service function chain model maps to multi-stage inference pipelines
- MFG approach avoids exponential blowup of N-player formulations

5. Atomic congestion game PoA bounds — Christodoulou & Koutsoupias, STOC 2005

- Provides foundational PoA ≤ 5/2 result for linear latency functions
- Any model of inference components as congestion game players inherits this bound
- A 2024 arXiv paper on edge-cloud task allocation shows PoA is typically close to 1 except under overload — theory is pessimistic for normal operating conditions

Additional Context

The Shamshirband et al. survey (Mathematical Biosciences and Engineering, 2021) of 110 papers on game theory for cloud resource allocation explicitly identifies the inter-component interaction gap: most work models user-provider interactions, very little models interactions between control loops *within* a system. This is the novel contribution space.

Practical warning: Most papers assume complete information, steady-state equilibrium, and rational agents. Real autoscalers use threshold-based heuristics — they are "boundedly rational" at best.

---

2. NVIDIA Dynamo Architecture

Dynamo is an open-source distributed inference serving framework (`github.com/ai-dynamo/dynamo`) that wraps vLLM (or TensorRT-LLM) as a backend while adding three orchestration subsystems:

Smart Router (KV Router)

- Dispatches each incoming request to a worker GPU by evaluating a cost function over the cluster's RadixTree (tracking which KV blocks are cached where)
- Balances KV cache overlap (prefix reuse) against load balancing
- Key parameters: `kv_overlap_score_weight` (range 0.1–5.0), `router_temperature` (range 0.0–2.0)
- Modes: `random`, `round-robin`, `kv` (KV-aware)
- Uses softmax over worker scores: `score = kv_overlap_score * weight + load_balance_score`
- Ref: https://docs.nvidia.com/dynamo/latest/router/README.html

Resource Planner (SLA-based Planner)

- Dynamically scales prefill and decode GPU worker pools based on SLA targets
- Uses ARIMA-predicted load with correction factors calibrated against actual TTFT/ITL
- Key parameters: `adjustment_interval` (default 30s), `grace_period` (3 intervals before scale-down)
- Maintains separate prefill and decode worker pools in disaggregated mode
- Ref: https://docs.nvidia.com/dynamo/latest/architecture/sla_planner.html

KV Block Manager (KVBM)

- Manages memory tier hierarchy: HBM3e (G1) → system DRAM (G2) → SSD (G3)
- B200 provides 192 GB HBM3e per GPU at ~8 TB/s bandwidth
- Controls eviction/promotion policies between tiers
- Key parameters: KV cache allocation percentage (50–95% of HBM), eviction aggressiveness, promotion threshold
- Ref: https://docs.nvidia.com/dynamo/latest/design-docs/kvbm-design

Observability Stack

- Prometheus metrics endpoint at `/metrics` on each component
- ~20+ built-in metrics: frontend requests, TTFT/ITL histograms, inflight/queued requests, KV stats (active blocks, total blocks, GPU cache usage percent, prefix cache hit rate), worker counts
- `MetricsRegistry` API supports custom counters, gauges, histograms with dynamic labels
- Grafana dashboards available: https://docs.nvidia.com/dynamo/latest/observability/prometheus-grafana.html
- Metrics reference: https://docs.nvidia.com/dynamo/latest/observability/metrics.html

Complementary Benchmarking Tools

- **AIPerf** (`github.com/ai-dynamo/aiperf`): Dynamo-native benchmarking tool measuring TTFT, ITL, request latency, throughput with time-sliced analysis
- **AI Configurator**: Supports `b200_sxm` for rapid 20-30 second simulation-based profiling
- **KV Router A/B Testing Guide**: https://docs.nvidia.com/dynamo/latest/benchmarks/kv-router-ab-testing.html
- **Planner Benchmark**: https://docs.nvidia.com/dynamo/latest/guides/planner_benchmark/benchmark_planner.html
- **Profiler Guide**: https://docs.nvidia.com/dynamo/v-0-9-0/components/profiler/profiler-guide

---

3. Three-Player Game Formulation

Player Definitions and Strategy Spaces

Player 1 — Smart Router

- Strategy space: routing weight vector across workers, parameterized by `kv_overlap_score_weight` (0.1–5.0) and `router_temperature` (0.0–2.0)
- Discretized strategies: {cache-greedy, balanced, load-greedy} (or finer: 5 evenly-spaced weight/temperature combos)
- Utility function:

$$U_R = -\alpha \cdot \text{TTFT}_{p95} - \beta \cdot \text{TBT}_{p95} + \gamma \cdot \text{cache\_hit\_rate}$$

where α, β, γ are weights measurable from telemetry.

Player 2 — Resource Planner

- Strategy space: allocation vector (n_prefill, n_decode) where n_prefill + n_decode ≤ 8

- Discretized strategies: {aggressive-scale, moderate, conservative} or all valid integer splits
- Utility function:

$$U_P = \text{throughput} - \lambda \cdot \text{GPU\_count} - \mu \cdot \text{SLO\_violation\_rate}$$

Player 3 — KV Block Manager

- Strategy space: eviction/promotion policy parameterized by KV cache allocation % (50–95%), eviction aggressiveness, promotion threshold
- Discretized strategies: {aggressive-evict, moderate, retain-heavy}
- Utility function:

$$U_M = -\delta \cdot \text{eviction\_rate} + \epsilon \cdot \text{memory\_headroom} - \zeta \cdot \text{recomputation\_rate}$$

Where Conflicts Emerge

Conflict	Mechanism
Router vs Planner	Router benefits from more workers (more routing options, better cache distribution). Planner wants to minimize GPU count. When planner scales down, router's cache hit rate drops as blocks are lost on terminated workers.
Router vs KVBM	Router routes to workers with cached prefixes (maximizing hits), concentrating load. KVBM on those workers faces memory pressure, triggering evictions that destroy the cache the router was exploiting. Creates measurable feedback loop: alternating spikes in cache hit rate and eviction rate.
Planner vs KVBM	When planner adds a new decode worker, KVBM must populate cache from scratch (cold start). Planner wants rapid scaling; KVBM needs time to build useful cache. The grace period (3 adjustment intervals before scale-down) explicitly acknowledges this tension.

Cooperative vs Non-Cooperative Formulations

Non-cooperative: Each player maximizes own utility given others' strategies. Nash equilibrium = configuration where no single subsystem can improve its utility by unilaterally changing strategy. Natural formulation since Dynamo's components operate independently.

Cooperative (Shapley value): Model grand coalition utility as weighted sum of all three utilities. Shapley value attributes marginal contributions: how much does the router's KV-aware intelligence contribute beyond random routing? Directly answers "which component matters most?"

Stackelberg structure (best fit for Dynamo): Planner as leader (sets worker allocation first), Router as follower (adapts routing given allocation), KVBM as another follower (adapts caching given routing patterns). Maps to actual control flow.

Simplest Implementable Version

Discretize each player's strategy space into 3–5 heuristic strategies. Run all **27–125 strategy profiles** as benchmark configurations. Build the payoff tensor from measured metrics. Solve for Nash equilibria using Gambit or Nashpy.

---

4. Empirical Equilibrium Detection

Tier 1: Stationarity Tests (when metrics stop drifting)

Augmented Dickey-Fuller test (`statsmodels.tsa.stattools.adfuller`) on sliding windows of each key metric — TTFT p95, throughput, queue depth, replica count, KV cache utilization. p-value < 0.05 = stationary. Combine with **KPSS test** for robustness: both stationary = confirmed equilibrium; both non-stationary = still converging.

```python
from statsmodels.tsa.stattools import adfuller, kpss

def check_stationarity(series, window_size=300):
    adf_p = adfuller(series[-window_size:])[1]
    kpss_p = kpss(series[-window_size:], regression='c')[1]
    return adf_p < 0.05 and kpss_p > 0.05  # stationary by both tests
```

Tier 2: Change-Point Detection (moment of convergence)

PELT algorithm (via `ruptures` library) identifies the last structural break. Everything after = candidate steady-state window. For online detection, **CUSUM** fires when metric mean shifts — if CUSUM stops firing, system has stabilized.

```python
import ruptures as rpt

def find_last_changepoint(signal):
    algo = rpt.Pelt(model="rbf").fit(signal)
    changepoints = algo.predict(pen=10)
    return changepoints[-2] if len(changepoints) > 1 else 0
```

Tier 3: Oscillation Detection (true equilibrium vs limit cycles)

Critical insight: a system can be stationary (ADF passes) but trapped in a stable oscillation. The **autocovariance function zero-crossing method** (Thornhill, Shah & Huang, 2003) detects this:

```python
```

```python
import numpy as np

def detect_oscillation(signal, cv_threshold=0.5):
    acf = np.correlate(signal - np.mean(signal), signal - np.mean(signal), mode='full')
    acf = acf[len(acf)//2:] / acf[len(acf)//2]  # normalize
    zero_crossings = np.where(np.diff(np.sign(acf)))[0]
    if len(zero_crossings) < 4:
        return False, None
    intervals = np.diff(zero_crossings)
    cv = np.std(intervals) / np.mean(intervals)
    period = 2 * np.mean(intervals)
    return cv < cv_threshold, period
```

What instability looks like in Dynamo:

- Queue depth oscillation: router overloads a worker → KVBM evicts → router redirects → original recovers → cycle repeats
- Replica count ping-ponging: planner scales up → utilization drops → scales down → latency spikes → scales up
- Cache hit rate sawtooth: router achieves high affinity → memory pressure builds → mass eviction → cache hit rate crashes

Tier 4: Game-Theoretic Regret Measurement (ε-Nash confirmation)

For each player, compute **regret** = (payoff from best alternative strategy) − (payoff from current strategy). If max regret across all players < ε, system is at ε-Nash equilibrium.

```python
```

```
import nashpy as nash
import numpy as np

def compute_regret(payoff_tensor, current_profile):
    """
    payoff_tensor: shape (n_strategies_p1, n_strategies_p2, n_strategies_p3, 3)
    current_profile: tuple (s1, s2, s3) - current strategy indices
    """
    regrets = []
    for player in range(3):
        current_payoff = payoff_tensor[current_profile][player]
        # Find best response for this player given others' strategies
        best_payoff = -np.inf
        for alt_strategy in range(payoff_tensor.shape[player]):
            alt_profile = list(current_profile)
            alt_profile[player] = alt_strategy
            alt_payoff = payoff_tensor[tuple(alt_profile)][player]
            best_payoff = max(best_payoff, alt_payoff)
        regrets.append(best_payoff - current_payoff)
    return max(regrets)
```

Recommended Detection Pipeline

For each benchmark run:

1. Collect metrics at 1-second intervals
2. Apply ADF+KPSS to identify when stationarity begins
3. Run PELT to pinpoint the last changepoint
4. Apply ACF oscillation detection to the post-changepoint window
5. If stationary and non-oscillatory, compute per-player regret
6. Composite verdict: all four tests must pass

Required Python Libraries

```
pip install statsmodels ruptures nashpy gambit scipy numpy pandas
```

---

5. Pareto Frontier Methodology

Experiment Design for 8×B200 in 3 Days

Configuration space has **5 primary axes**: batch size (1–512, powers of 2), request concurrency (1–128), prefill/decode GPU split (1:7 through 4:4), KV cache allocation (50–95% of HBM), routing temperature (0.0–

2.0).

With 5 parameters, Latin Hypercube Sampling requires minimum **51 configurations** (10D + 1 rule). Target **80–100 configurations** for a well-populated frontier.

```python
from scipy.stats.qmc import LatinHypercube
import numpy as np

sampler = LatinHypercube(d=5)
sample = sampler.random(n=80)

# Scale to parameter ranges
batch_sizes = [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
configs = []
for row in sample:
    config = {
        'batch_size': batch_sizes[int(row[0] * len(batch_sizes))],
        'concurrency': int(1 + row[1] * 511),
        'prefill_gpus': max(1, int(1 + row[2] * 3)),  # 1-4
        'kv_cache_pct': 0.50 + row[3] * 0.45,  # 50-95%
        'router_temp': row[4] * 2.0,  # 0.0-2.0
    }
    configs.append(config)
```

3-Day Schedule

Day 1 — Broad exploration: Run 80 LHS configs, 2 warmup + 5 measurement iterations each. ~15 min/config including restarts = ~20 hours.

Day 2 — Adaptive refinement: Construct initial Pareto front. Import Day 1 results into Ax/BoTorch. Switch to **qNEHVI** acquisition to suggest 30-40 configs targeting frontier improvement.

Day 3 — Validation: For all Pareto-optimal configs, run 15-20 additional iterations. Bootstrap Pareto confidence analysis. Compute hypervolume indicator.

Tools

pymoo — Post-hoc Pareto analysis of existing data:

```python
```

```python
from pymoo.util.nds.non_dominated_sorting import NonDominatedSorting
from pymoo.indicators.hv import HV

# objectives: minimize latency, maximize throughput, minimize cost
# (negate throughput for minimization)
F = np.column_stack([ttft_p95, -throughput, cost_per_token])
nds = NonDominatedSorting()
fronts = nds.do(F)
pareto_mask = fronts[0]

# Hypervolume indicator
ref_point = np.array([500, -100, 0.01])  # worst acceptable values
hv = HV(ref_point=ref_point)
hv_value = hv(F[pareto_mask])
```

Ax/BoTorch — Adaptive benchmark campaign driver:

```python
from ax.service.ax_client import AxClient
from ax.service.utils.instantiation import ObjectiveProperties

ax_client = AxClient()
ax_client.create_experiment(
    parameters=[
        {"name": "concurrency", "type": "range", "bounds": [1, 512]},
        {"name": "prefill_gpus", "type": "range", "bounds": [1, 4]},
        {"name": "kv_cache_pct", "type": "range", "bounds": [0.5, 0.95]},
        {"name": "router_temp", "type": "range", "bounds": [0.0, 2.0]},
    ],
    objectives={
        "ttft_p95": ObjectiveProperties(minimize=True, threshold=500.0),
        "throughput": ObjectiveProperties(minimize=False, threshold=100.0),
        "cost_per_token": ObjectiveProperties(minimize=True, threshold=0.01),
    },
)
```

Statistical Robustness

ε-Dominance filtering: Set ε to ~2× standard error per objective. Point is ε-Pareto-optimal only if no other point dominates it by more than ε on all objectives.

Bootstrap Pareto confidence: Resample 1000 times, count how often each config appears on frontier. Keep points appearing in ≥95% of resamples. Use BCa bootstrap (`scipy.stats.bootstrap`) since latency distributions are skewed.

Hypervolume indicator: Only Pareto-compliant unary metric. Reference point must be set once from domain knowledge and never changed. For 2-3 objectives, exact computation is fast; 4+ objectives use Monte Carlo.

Gotchas

- Insert 30-second cooling periods between runs (thermal throttling)
- Restart serving process between config changes (memory fragmentation)
- Discard first 2-3 inference requests (JIT warmup)
- If all objectives are correlated, the "Pareto front" will be thin — objectives aren't truly conflicting in your parameter range

---

6. vLLM Benchmark Suite Catalog

Core Trio

Script	CLI Alias	Measures	Key Parameters
`benchmark_serving.py`	`vllm bench serve`	TTFT, TPOT, ITL, E2EL, throughput, goodput	`--backend`, `--base-url`, `--dataset-name`, `--request-rate`, `--max-concurrency`, `--goodput`
`benchmark_throughput.py`	`vllm bench throughput`	Offline batch: req/s, tok/s	`--backend`, `--tensor-parallel-size`, `--pipeline-parallel-size`, `--enable-prefix-caching`, `--kv-cache-dtype`
`benchmark_latency.py`	`vllm bench latency`	Per-iteration latency	`--batch-size`, `--input-len`, `--output-len`, `--tensor-parallel-size`, `--num-iters`

KV Cache & Prefix Caching

Script	Purpose
`benchmark_prefix_caching.py`	Throughput with/without prefix caching; `--repeat-count`, `--input-length-range`
`benchmark_long_document_qa_throughput.py`	Long-document prefix reuse; `--repeat-mode {random,tile,interleave}`, `--num-documents`
`benchmark_hash.py`	Micro-benchmark for prefix-cache hash function latency
`benchmark_prefix_block_hash.py`	Block-hashing throughput across hash algorithms

Specialized

Script	Purpose
`benchmark_serving_structured_output.py`	JSON/grammar/regex structured output; `--structured-output-ratio`
`benchmark_prioritization.py`	Priority scheduling; `--scheduling-policy priority`

Subdirectories

- **`disagg_benchmarks/`**: `disagg_prefill_proxy_server.py` — proxy routing prefill/decode to specialized instances
- **`multi_turn/`**: `benchmark_serving_multi_turn.py` — multi-turn conversation with `--num-clients`, `--max-active-conversations`
- **`sweep/`**: `serve.py` (Cartesian product sweep), `serve_sla.py` (SLA-aware tuning). **Critical for systematic exploration.**

Output Format

JSON via `--save-result` / `--result-dir`. Fields: successful_requests, benchmark_duration, total_tokens, throughput (req/s, tok/s), mean/median/P99 for TTFT, TPOT, ITL, E2EL.

---

7. Benchmark-to-Goal Mapping

Goal 1: vLLM Cookbook for Dynamo

What to benchmark	Script	Configuration
TP scaling (TP1×DP8 through TP8)	`benchmark_serving.py` via `sweep/serve.py`	ShareGPT + random, concurrency sweep
Aggregated vs disaggregated	`benchmark_serving.py` against different Dynamo deploy configs	Same workloads, different P:D ratios
Routing mode comparison	`benchmark_serving.py`	Same workload, routing = {random, round-robin, kv}
KV cache dtype impact	`benchmark_throughput.py`	`--kv-cache-dtype {auto,fp8}`
Prefix caching benefit	`benchmark_prefix_caching.py`	Various repeat counts and input lengths
Multi-turn performance	`multi_turn/benchmark_serving_multi_turn.py`	Various conversation counts

Goal 2: B200 Contribution to vLLM

Deliverable	Format	Target
Standard benchmark results	JSON from `--save-result`	PR against `.buildkite/performance-benchmarks/` configs
B200-specific model configs	Python config files	PR similar to #19455 (fused MOE configs)
Coordination point	GitHub issue	Track under #28883

Goal 3: Game-Theoretic Paper

Data needed	Source	Gap?
TTFT, ITL distributions per config	`benchmark_serving.py`	No — direct output
Throughput per config	`benchmark_serving.py`	No — direct output
Cache hit rate	Dynamo Prometheus metrics	No — `dynamo_prefix_cache_hit_rate`
GPU cache utilization	Dynamo Prometheus metrics	No — `dynamo_gpu_cache_usage_perc`
Queue depth over time	Dynamo Prometheus metrics	No — `dynamo_num_requests_waiting`
Worker count over time	Dynamo Prometheus metrics	No — available
Per-worker router scores	**Not exposed**	**Yes — requires instrumentation**
Per-tier eviction rates	**Not exposed**	**Yes — requires instrumentation**
Planner decision timestamps	**Logs only, not metrics**	**Yes — requires conversion**
Block lifecycle timing	**Not exposed**	**Yes — requires instrumentation**
Recomputation rate	**Not exposed**	**Yes — requires instrumentation**

8. Dynamo + vLLM Integration Details

Deployment

```bash
```

```
# Pull container (includes B200-compatible PyTorch)
docker pull nvcr.io/nvidia/ai-dynamo/vllm-runtime:latest

# Or install via pip
uv pip install ai-dynamo[vllm]

# Launch aggregated mode (TP8, single model)
python3 -m dynamo.vllm \
   --model meta-llama/Llama-3.1-70B-Instruct \
   --tensor-parallel-size 8 \
   --gpu-memory-utilization 0.90 \
   --max-model-len 8192 \
   --port 8000

# Launch disaggregated mode (example: 2P:6D)
# Use Dynamo deployment config (disagg_2p6d.yml or equivalent)
dynamo serve --config disagg_2p6d.yml
```

B200-Specific Flags

```
bash

# Required for optimal B200 performance
--cuda-graph-capture-size 2048
--api-server-count 20
--stream-interval 20

# Environment variables
export VLLM_USE_FLASHINFER_MOE_MXFP4_MXFP8=1
```

Requirements

- NVIDIA driver 575+ / CUDA 13+
- NGC container-based PyTorch (standard PyPI PyTorch does not support B200/sm_100)
- FlashInfer MOE backends for MoE models

Known Issues

- GitHub ai-dynamo/dynamo#2552: Dynamo+vLLM disaggregated P/D can be slower than vanilla vLLM in some configs
- GitHub vllm-project/vllm#18725: B200 performing similarly to H200 without tuned configs
- Fix: requires specific flags, newer CUDA/driver versions, and FlashInfer MOE backends

Connecting vLLM Benchmarks to Dynamo

```bash
# benchmark_serving.py works directly against Dynamo's OpenAI endpoint
python benchmarks/benchmark_serving.py \
  --backend openai-chat \
  --base-url http://localhost:8000 \
  --model meta-llama/Llama-3.1-70B-Instruct \
  --dataset-name sharegpt \
  --request-rate 10 \
  --max-concurrency 64 \
  --save-result \
  --result-dir ./results/
```

---

9. Contributing B200 Results to vLLM

Process

1. B200 is already in vLLM's Buildkite CI alongside A100, H100, Intel Xeon, Gaudi 3, Arm Neoverse
2. CI config lives in `.buildkite/performance-benchmarks/` with JSON configs (`latency-tests.json`, `throughput-tests.json`, `serving-tests.json`)
3. Submit PR modifying or adding to config files with `perf-benchmarks` label
4. `compare-json-results.py` compares before/after for output throughput, median TTFT, median TPOT ratios
5. Results appear on continuous dashboard at **perf.vllm.ai**

Existing B200/Blackwell Work to Build On

- **Tracking issue #28883**: Central coordination for (G)B200/300 performance improvements, maintained by NVIDIA's @pavanimajety
- **vLLM blog (Feb 2026)**: Up to 4.3× throughput on gpt-oss-120b (1K/1K ISL/OSL) on Blackwell vs. Hopper
- **SemiAnalysis InferenceX** (`github.com/SemiAnalysisAI/InferenceX`): B200-specific benchmark scripts, continuous results at inferencex.com
- **PR #19455**: Pattern for hardware-specific contributions (fused MOE configs for B200 with before/after data)

Contribution Checklist

- [ ] Run standard benchmark suite on Llama 3.1 70B (most benchmarked model for comparability)
- [ ] Save JSON results via `--save-result`
- [ ] Document B200-specific flags and container requirements

- [ ] Compare against H100/H200 baselines if available
- [ ] Submit PR with B200 config additions to `.buildkite/performance-benchmarks/`
- [ ] Reference tracking issue #28883

---

10. Instrumentation Gap Analysis

What Dynamo Prometheus Already Provides (~40% of game-theoretic needs)

```
# Frontend / Router
dynamo_request_success_total
dynamo_request_failure_total
dynamo_time_to_first_token_seconds (histogram)
dynamo_inter_token_latency_seconds (histogram)
dynamo_e2e_request_latency_seconds (histogram)
dynamo_num_requests_running
dynamo_num_requests_waiting

# KV Cache
dynamo_gpu_cache_usage_perc
dynamo_num_active_kv_blocks
dynamo_num_total_kv_blocks
dynamo_prefix_cache_hit_rate

# Workers
dynamo_num_prefill_workers
dynamo_num_decode_workers
```

What Needs Custom Instrumentation (~60%)

Using Dynamo's `MetricsRegistry` API:

```python
```

```python
# Example: Add per-worker router score distribution
from dynamo.metrics import MetricsRegistry

registry = MetricsRegistry()

# Router instrumentation
router_worker_score = registry.histogram(
    "dynamo_router_worker_score",
    "Per-worker routing score",
    labels=["worker_id", "score_component"],  # kv_overlap, load_balance
    buckets=[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
)

router_decision_counter = registry.counter(
    "dynamo_router_decisions_total",
    "Routing decisions by type",
    labels=["selected_worker", "reason"]  # cache_hit, load_balance, random
)

# KVBM instrumentation
kvbm_eviction_counter = registry.counter(
    "dynamo_kvbm_evictions_total",
    "KV block evictions by tier transition",
    labels=["from_tier", "to_tier"]  # g1_to_g2, g2_to_g3, g3_to_evict
)

kvbm_tier_occupancy = registry.gauge(
    "dynamo_kvbm_tier_occupancy_bytes",
    "KV cache occupancy per memory tier",
    labels=["tier"]  # g1_hbm, g2_dram, g3_ssd
)

# Planner instrumentation (convert logs to metrics)
planner_scaling_events = registry.counter(
    "dynamo_planner_scaling_events_total",
    "Scaling decisions",
    labels=["direction", "worker_type"]  # scale_up/scale_down, prefill/decode
)

planner_predicted_load = registry.gauge(
    "dynamo_planner_predicted_load",
    "ARIMA-predicted load vs actual",
    labels=["metric_type"]  # predicted, actual
)
```

Estimated effort: 2-3 weeks for someone familiar with the Dynamo codebase.

11. Publishability Assessment

Minimum Viable Contribution

The game-theoretic framing alone is **necessary but not sufficient**. The contribution becomes publishable when you demonstrate something traditional single-metric benchmarking cannot reveal:

1. **Empirical equilibrium detection**: Show Dynamo converges to measurable ε-Nash equilibrium under steady load, characterize how ε varies with load intensity
2. **Oscillation characterization**: Identify and quantify Router↔KVBM feedback oscillation (period, amplitude, onset conditions)
3. **Pareto frontier with Shapley attribution**: "The KV-aware router contributes 48% of the latency improvement at the Pareto frontier"
4. **Price of Anarchy measurement**: Compare decentralized equilibrium vs social optimum. PoA close to 1 = well-designed; significantly above 1 = room for coordination

Venue Targeting

Effort	Venue	Requirements	Probability
3-day sprint, 1 model	Workshop (MLSys, HotOS, EuroMLSys)	Formulation + equilibrium detection + Pareto/PoA. 4-6 pages.	**High**
1-2 weeks, 2-3 models	**EuroMLSys or ISPASS main track**	Cross-model comparison. Show game dynamics change with architecture. 8-10 pages.	**Solid shot**
1-2 months	**MLSys main track**	Everything above + game-theory-informed optimizer that beats Dynamo defaults.	**Competitive**

OSDI, SOSP, ISCA are unrealistic for single-author effort without a new system artifact.

Strongest Parts

- Confirmed novel gap (game theory × LLM serving infrastructure)
- Practical detection methodology (ADF → PELT → ACF → regret)
- Timely: Dynamo is new, B200 is current-gen
- Multi-objective analysis goes beyond standard peak-throughput reporting

Weakest Parts & Mitigations

- **Post-hoc rationalization risk**: Mitigate by showing game formulation predicts specific instabilities
- **Strategy space discretization is arbitrary**: Run sensitivity analysis on 3, 5, 7 strategy granularities

- **Single system/hardware**: Frame as case study + methodology contribution
- **Benchmark time is tight**: Use Ax/BoTorch adaptive sampling; be transparent about exploratory scope

---

12. 4-Week Execution Plan

Week 1: Infrastructure & Validation

Priority: Get Dynamo + vLLM running. Everything else is blocked without this.

```bash
# Step 1: Pull container
docker pull nvcr.io/nvidia/ai-dynamo/vllm-runtime:latest

# Step 2: Deploy aggregated mode (TP8, Llama 3.1 70B)
python3 -m dynamo.vllm \
    --model meta-llama/Llama-3.1-70B-Instruct \
    --tensor-parallel-size 8 \
    --gpu-memory-utilization 0.90 \
    --max-model-len 8192 \
    --cuda-graph-capture-size 2048 \
    --port 8000

# Step 3: Smoke test
python benchmarks/benchmark_serving.py \
    --backend openai-chat \
    --base-url http://localhost:8000 \
    --model meta-llama/Llama-3.1-70B-Instruct \
    --dataset-name sharegpt \
    --request-rate 10 \
    --num-prompts 100 \
    --save-result \
    --result-dir ./results/week1_smoke/

# Step 4: Set up Prometheus + Grafana
# docker-compose with prometheus scraping Dynamo /metrics at 1s intervals
# Import Dynamo's Grafana dashboard templates
```

Deliverables: Confirmed working deployment, baseline latency/throughput numbers, Grafana dashboards live.

Week 2: Systematic Benchmarking (Cookbook + Contribution Data)

```bash
```

```
# Sweep: aggregated mode, vary concurrency and TP
python benchmarks/sweep/serve.py \
    --serve-params '{"tensor_parallel_size": [1,2,4,8]}' \
    --bench-params '{"max_concurrency": [1,8,32,64,128,256,512], "dataset_name": ["sharegpt","random"]}' \
    --base-url http://localhost:8000 \
    --model meta-llama/Llama-3.1-70B-Instruct \
    --num-runs 3 \
    --result-dir ./results/week2_sweep_agg/

# Sweep: disaggregated mode, vary P:D ratios
for ratio in "1:7" "2:6" "3:5" "4:4"; do
    # Redeploy with each P:D config
    # Re-run same concurrency sweep
done

# Second model (MoE)
# Repeat above with Mixtral-8x22B-Instruct or DeepSeek-V3-Lite
```

Deliverables: Complete benchmark matrix for cookbook recipes, JSON results ready for vLLM PR.

Week 3: Game-Theoretic Instrumentation & Targeted Runs

bash

```
# Fork Dynamo, add custom metrics
git clone https://github.com/ai-dynamo/dynamo.git
cd dynamo
# Add MetricsRegistry instrumentation per Section 10

# Re-run subset with instrumentation
# Focus on: routing mode comparison
for mode in "random" "round_robin" "kv"; do
    python benchmarks/benchmark_serving.py \
        --backend openai-chat \
        --base-url http://localhost:8000 \
        --dataset-name prefix_repetition \
        --prefix-repetition-num-prefixes 1,4,16,64 \
        --max-concurrency 64 \
        --num-prompts 1000 \
        --save-result \
        --result-dir ./results/week3_routing_${mode}/
done

# KV pressure testing
python benchmarks/benchmark_long_document_qa_throughput.py \
    --repeat-mode interleave \
    --num-documents 10 \
    --kv-cache-dtype auto

# Scrape Prometheus at 1s during all runs
# Save to CSV for post-hoc analysis
```

Deliverables: Per-component telemetry data, routing strategy comparison data, KV cache behavior under pressure.

Week 4: Analysis & Writing

```
python

# Build Pareto frontiers from Week 2 data
# Apply equilibrium detection pipeline to Week 3 time series
# Compute Shapley values
# Measure Price of Anarchy
# Generate all figures

# Write paper (6-8 pages)
# Submit vLLM contribution PR
# Start cookbook draft
```

Deliverables:

- [ ] Paper draft (6-8 pages)
- [ ] vLLM PR with B200 benchmark configs + results
- [ ] Cookbook outline with key recipes
- [ ] Instrumented Dynamo fork on GitHub

---

13. Key References

Game Theory & Cloud/Distributed Systems

1. Luo et al., "Runtime Resource Management for Microservices-Based Applications", CollaborateCom 2018
2. Anselmi et al., "Three-tier congestion game for cloud economics", TOMPECS 2017
3. Ardagna et al., "A Game Theoretic Formulation of the Service Provisioning Problem in Cloud Systems", IEEE TSC 2013
4. Abouaomar et al., "Mean-field game for service function chaining", IEEE Systems Journal 2022
5. Christodoulou & Koutsoupias, "The Price of Anarchy of Finite Congestion Games", STOC 2005
6. Shamshirband et al., "Game theory in cloud resource allocation: survey of 110 papers", Mathematical Biosciences and Engineering 2021
7. Johari & Tsitsiklis, "Efficiency Loss in a Network Resource Allocation Game", Mathematics of Operations Research 2004

LLM Inference Systems

8. vLLM — PagedAttention, continuous batching (https://github.com/vllm-project/vllm)
9. NVIDIA Dynamo — distributed inference framework (https://github.com/ai-dynamo/dynamo)
10. vLLM Blog: "GPT-OSS Performance Optimizations on NVIDIA Blackwell" (Feb 2026)
11. vLLM Blog: "SemiAnalysis InferenceMAX: vLLM and NVIDIA Accelerate Blackwell Inference" (Oct 2025)
12. SemiAnalysis InferenceX benchmarks (https://github.com/SemiAnalysisAI/InferenceX)

Statistical Methods

13. Thornhill, Shah & Huang, "Detection of multiple oscillations in control loops", Journal of Process Control 2003
14. PELT algorithm — `ruptures` library for change-point detection
15. Daulton et al., "Parallel Bayesian Optimization of Multiple Noisy Objectives with Expected Hypervolume Improvement", NeurIPS 2021

Multi-Objective Optimization Tools

16. pymoo: Multi-objective Optimization in Python (https://pymoo.org)
17. Ax/BoTorch: Bayesian optimization platform (https://ax.dev, https://botorch.org)
18. Hypervolume indicator — Zitzler et al., "The Hypervolume Indicator Revisited", EMO 2007

Dynamo Documentation

19. Smart Router: https://docs.nvidia.com/dynamo/latest/router/README.html
20. SLA Planner: https://docs.nvidia.com/dynamo/latest/architecture/sla_planner.html
21. KVBM Design: https://docs.nvidia.com/dynamo/latest/design-docs/kvbm-design
22. Metrics: https://docs.nvidia.com/dynamo/latest/observability/metrics.html
23. KV Router A/B Benchmarking: https://docs.nvidia.com/dynamo/latest/benchmarks/kv-router-ab-testing.html
24. AIPerf: https://github.com/ai-dynamo/aiperf

Last updated: February 27, 2026