BROADCOM

VMware Cloud Foundation®

Disaggregated vs. Monolithic LLM Serving on VMware Cloud Foundation with NVIDIA Run:ai and Dynamo

An Outcome-Centered Capacity Study of NVIDIA Dynamo (Disaggregated) versus a Monolithic vLLM Baseline for Nemotron-3-Super-120B-A12B, on VMware Cloud Foundation + vSphere® Kubernetes Service + NVIDIA Run:ai

White Paper

Table of Contents

Chapter 1: Abstract

Enterprises sizing GPU infrastructure for LLM inference are usually sold on tokens per second, a measure of how fast the engine spins, not whether the factory produces useful work. Building on Broadcom® SPOC and Envoy studies of realistic LLM workloads, this paper asks the question that actually governs enterprise AI economics:

> *How many useful workflows, per class of user, can a platform sustain within an acceptable user-experience service-level objective (SLO), and how does it fail once pushed past that point?*

We apply it to a concrete architectural choice: disaggregated LLM serving (separate prefill and decode pools joined by NVIDIA NIXL, on NVIDIA Dynamo) versus a monolithic vLLM baseline. The model under test is Nemotron-3-Super-120B-A12B, a ~120-billion-parameter Mamba/Attention-hybrid mixture-of-experts (MoE) model, running on VMware Cloud Foundation® (VCF) with VMware vSphere® Kubernetes Service (VKS), with NVIDIA Run:ai for GPU orchestration and NVIDIA Dynamo for serving deployed on top of that platform (8× NVIDIA H100), the on-premises, enterprise-governed AI posture the rest of this paper assumes. A traffic-realistic, four-profile workload driven from large real-world corpora drives each shape to its QoS-defined capacity and governing saturation point; we report the client-experienced, ingress-inclusive latency, not just the server's intrinsic latency.

Our most interesting finding is a failure-mode asymmetry. The monolith sustains far more raw concurrency before it breaks, but then breaks abruptly: an ~22× time-to-first-token jump in a single ramp step. The disaggregated shape, by contrast, is stopped while still healthy by a proactive compute-utilization gate at much lower concurrency, and it degrades via bounded admission-queueing rather than a cliff. On the primary SLO metric, disaggregation's violation is proportionally smaller and confined to one traffic profile rather than three. Two honesty caveats bound how far these results travel:

1. Each shape was measured once (single run, single model, single node).
2. The comparison is of each architecture's best realistic deployment rather than a flag-identical control, since the monolith uses prefix caching and speculative decoding that this Mamba-hybrid model cannot yet use across a disaggregation boundary. That limitation is itself a real cost of disaggregating this model class today.

We distill the result into an actionable operator decision framework and a reproducible setup recipe. That recipe includes the concrete mechanics of integrating NVIDIA Run:ai with NVIDIA Dynamo and the fixes required to disaggregate a Mamba-hybrid model at all, details not well documented elsewhere. We argue, with evidence, that tokens/sec is the wrong yardstick for this decision.

Chapter 2: Introduction and Motivation

2.1 The Token Is Not the Product

Enterprises evaluating GPU infrastructure for LLMs are repeatedly handed a single number: tokens per second. It is a popular metric because it is easy to measure and easy to compare. It is also a poor proxy for what an enterprise actually buys. Tokens per second tells you how fast the engine spins; it does not tell you whether the factory is producing useful work. The unit an enterprise pays for is not the token. It is the completed, trusted workflow.

The GPU matters, but in most enterprise workflows it is not the whole factory, and often not even the binding constraint. The real cost lives around the GPU: context assembly, retrieval, permissions, scheduling behavior, tail latency, and the service-level objectives that decide whether a chat feels interactive or an agent finishes its task before a human gives up. The right capacity-planning question is therefore not *How fast can this GPU generate tokens?* but:

> *How many useful enterprise workflows can this platform sustain, for each class of user, before latency, context growth, or orchestration break the platform's usability?*

This paper answers that question for a specific, increasingly common architectural choice: disaggregated LLM inference serving versus the monolithic serving that most deployments, including our own SPOC and Envoy studies, started with.

2.2 From SPOC and Envoy to Disaggregation

This is the third study in a series on traffic-realistic LLM capacity planning on VCF. SPOC modeled the stateful, context-accumulating behavior of developers using coding assistants. Envoy extended the model to a harder mix: human interactive traffic alongside autonomous agent actions. Both ran monolithic vLLM. Both moved the conversation away from generic GPU benchmarks and toward the shape of the actual work: usage profiles, accumulated context, prefix caching, scheduling, and service-level targets.

Here we keep that traffic-realistic methodology and turn it on an architectural question. Prefill (reading the prompt) is compute-bound; decode (emitting tokens) is memory-bandwidth-bound. Running both on one replica forces a single hardware profile to serve two very different workloads. Disaggregation splits them into independently scaled prefill and decode pools joined by a KV-cache transfer fabric (NVIDIA NIXL). This promises a better resource fit, but at the cost of transfer latency, coordination, and a larger failure surface. Recent analyses agree the benefit is conditional, not universal (see Chapter 10). Our contribution is to measure where the line is, in the currency that matters to operators.

2.3 What We Measure, and How

We compare two serving shapes for the same model (Nemotron-3-Super-120B-A12B) on identical hardware (8× H100) under an identical blended workload. One is a monolithic baseline, which we call Mono-vLLM, prefill and decode colocated. The other is a disaggregated shape which we call Dynamo, separate prefill and decode pools joined by NIXL. The workload blends four user profiles: an API orchestrator (bursty autonomous agents), a CI/terminal siege, a UI navigator, and an interactive knowledge worker. Each profile is driven by real-world prompt corpora and an arrival model grounded in traffic research rather than a convenient Poisson assumption.

Crucially, we evaluate each shape by user-experience SLO attainment per profile: the number of concurrent users/workflows it sustains while keeping that profile's latency within an acceptable interactive or agentic bound. We do not evaluate it by peak tokens/sec. Just as important, we characterize how each shape behaves once pushed past what it can sustain. The two fail in very different ways, and that difference turns out to be one of the study's central findings. We treat the mathematical saturation point as a mechanism to explain why a shape breaks, not as the headline. Because real users reach the service through an ingress, we report the client-experienced (ingress-inclusive) latency they actually see, not only the server-side intrinsic latency (see Chapter 5.5; the limits of cleanly decomposing the two for this dataset are discussed in Chapter 8.4).

2.4 Why VCF + VKS + NVIDIA Run:ai + Dynamo

The platform is part of the finding. We deploy NVIDIA Run:ai (GPU orchestration) and NVIDIA Dynamo (disaggregated serving + NIXL) on top of VKS atop VCF. That combination lets enterprises run capable models like NVIDIA Nemotron on their own infrastructure, under their own governance and data control, rather than sending prompts to a public endpoint. VCF/VKS supplies exactly the factory around the GPU: multitenant governance, life cycle and storage management, on-premises data residency, and an operationally mature, CNCF-conformant Kubernetes substrate. These are the parts of enterprise AI economics that a raw GPU benchmark ignores. We show this stack runs the full disaggregated campaign reproducibly, and we give the setup as an actionable recipe (see Chapter 6).

2.5 Contributions

1. An outcome-centered comparison of disaggregated vs monolithic serving, scored by per-profile SLO attainment (sustainable useful workflows) and by failure mode, with tokens per second deliberately demoted to supporting evidence.
2. A traffic-realistic, multiprofile workload built from large real-world corpora, with arrival models grounded in traffic-modeling research, engineered to avoid the prefix-cache masking that flatters synthetic benchmarks.
3. An empirical monolithic-versus-disaggregated comparison (Mono-vLLM vs. Dynamo, see Chapter 5.1) on production VCF + VKS + Run:ai + Dynamo, including a measured account of the ingress latency real users incur and the concrete mechanics of getting a Mamba/Attention-hybrid model disaggregated on Dynamo (see Chapter 6.9), an integration path NVIDIA's own recipes do not seem to document for this model.
4. A characterization of how each architecture fails past its sustainable point, bounded, proactive compute-gating for the disaggregated shape versus an abrupt collapse cliff for the monolith, which we argue is as decision-relevant to an operator as the capacity numbers themselves.
5. An actionable operator decision framework, which serving shape to choose for which profile and concurrency target, plus a reproducible setup and telemetry recipe, including the Run:ai-Dynamo integration mechanics and the specific fixes required to disaggregate a Mamba/Attention-hybrid model.

Chapter 3: Background: Prefill/Decode Disaggregation, NVIDIA Dynamo and the KV-Transfer Fabric

3.1 The Model Under Test: Nemotron-3-Super Is a Genuinely Capable Model, Not a Toy

Before getting into serving architecture, it's worth grounding what this paper is actually load-testing. Nemotron-3-Super-120B-A12B (~120B total / ~12B active parameters, Mamba/Attention-hybrid MoE, with full architectural rationale, see Chapter 5.1) is not a lightweight stand in; independent, published evaluations place it competitively against models in and above its compute bracket:

- Mathematical and scientific reasoning[1]: 90.21% on AIME 2025 (zero-shot, no tool use) and 79.23% on GPQA Diamond (graduate-level science), rising to 82.70% on GPQA with tool use enabled.
- Real-world software engineering: 60.47% on SWE-Bench Verified, a benchmark built from actual GitHub issues. A score in this range reflects genuine capability at "code generation, bug fixing, and architectural reasoning... tasks that require understanding complex interdependencies across files and systems[2]". This is directly relevant to this paper's own profile A and profile D traffic (agentic tool-calling and multi-turn coding sessions, see Chapter 5.2).
- Long-context retrieval: On the RULER benchmark, the model holds 96.30% accuracy at 256K tokens, 95.67% at 512K tokens, and 91.75% out at its full 1M-token context window. This is a useful external cross-check on our own corpus: profile D's p99 input context reaches ~115K tokens, see Chapter 5.2. That is comfortably inside the range where Nemotron's long-context retrieval is independently validated to still hold up, not at the edge of a claimed-but-untested limit.
- Reported throughput: Baseten's engineering write-up reports "over 50% faster token generation than comparable models[3]," attributed to the hybrid Mamba-Transformer architecture and multi-token prediction. We cite this as the vendor/ecosystem claim it is, not as something this paper independently verified. Our own throughput-adjacent findings, see Chapter 7.4, measure serving-architecture capacity under our specific load, not the model's raw per-request generation speed against other models; the two should not be conflated.

None of this is a benchmark this paper reproduces or verifies independently. It is cited to establish, plainly, that the model under test is a serious, capable one, so the mono-vs.-disaggregated comparison in Chapter 7 is read in that context rather than as *how does a toy model behave*.

1. “nemotron-3-super.” *Ollama*, https://ollama.com/library/nemotron-3-super.
2. “How does Nemotron 3 Super perform on software development benchmarks?” *Milvus*, https://milvus.io/ai-quick-reference/how-does-nemotron-3-super-perform-on-software-development-benchmarks.
3. Rachel Rapp, “NVIDIA Nemotron 3 Super for agentic AI in financial services.” *Baseten*, April 3, 2026, https://www.baseten.co/blog/introducing-nemotron-3-super/.

3.2 Two Phases, Two Hardware Profiles

LLM inference has two phases with fundamentally different resource behavior, decode and prefill:

- Prefill processes the entire input prompt in one (or a few chunked) forward passes to produce the first output token. It is compute-bound: work scales with input sequence length (ISL) and saturates GPU FLOPs. It sets time-to-first-token (TTFT), the latency a user feels before anything appears.
- Decode generates subsequent tokens one at a time, each step reading and appending to the growing KV cache, the per-layer key/value tensors that let the model avoid recomputing attention over the whole history. Decode is memory-bandwidth-bound and capacity-bound: throughput depends on KV-cache residency and how many sequences share the GPU. It sets inter-token latency (ITL) and time-per-output-token (TPOT). Either metric determines how smooth generation feels, and the pace at which an agent completes multi-step work.

Every chat completion and every agent turn is a prefill followed by a decode stream. The architectural question is only where those phases run.

Figure 1: Request Lifecycle, Every Completion

3.3 The Monolithic Baseline

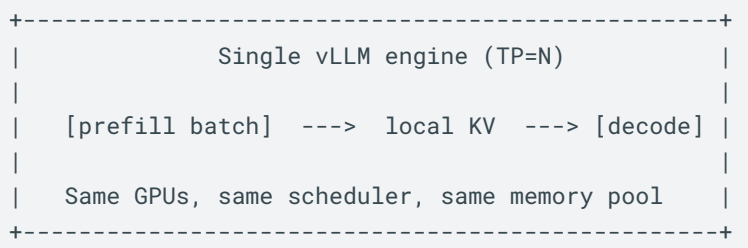
A monolithic replica runs both phases on the same GPUs. In this paper, Mono-vLLM is two colocated TP = 4 vLLM replicas load balanced by the front end, not one TP = 8 engine spanning all 8 GPUs (see Chapter 5.1). Prefill writes the KV into local GPU memory, and decode reads it immediately. There is no cross-worker transfer and no fabric to configure.

3.3.1 Two TP = 4 Replicas Instead of One TP = 8 Engine

We mirror NVIDIA's own official Dynamo recipe for this model verbatim, and that recipe specifies 2 × TP = 4, not TP = 8. The recipe does not state its reasoning, but the standard argument for large mixture-of-experts models applies directly. Tensor parallelism requires an all-reduce or all-to-all synchronization step on every layer across every participating GPU, and that cost grows with the number of GPUs in the group. Doubling the TP degree from 4 to 8 roughly doubles that per-layer coordination overhead without halving the useful compute per GPU, which hits decode's ITL especially hard because decode pays the synchronization cost once per generated token. Two independent TP = 4 replicas sidestep it: each batches and schedules its own requests, so the same 8 GPUs deliver more aggregate throughput through request-level (data) parallelism than one larger, more communication-bound engine would.

That simplicity is valuable and a compromise. One hardware and batching profile must serve two opposing workloads. A burst of long prompts can head-of-line-block token generation for everyone sharing the replica, inflating ITL when interactivity matters most. Operators also cannot add more prefill without adding more decode, because both phases live on the same GPUs.

Figure 2: Monolithic Replica, Generic; this Paper's Mono-vLLM is 2× Colocated TP = 4 Replicas



3.4 Disaggregation: Isolate Phases, Keep the Full Model

Disaggregated serving puts prefill and decode in separate, independently scaled worker pools. Prefill computes the prompt and exports the KV cache; decode imports it and streams tokens. That addresses the Chapter 3.3 compromises: a prefill burst no longer steals decode cycles, and operators can size each pool to its bottleneck (prompt-heavy versus generation-heavy traffic).

Two papers established the case for this split. DistServe (Zhong et al., OSDI 2024) and Splitwise (Patel et al.) both report large goodput and SLO gains over colocated baselines by giving prefill and decode independent hardware, see Chapter 10 for full details and figures. The costs are real: a KV-cache transfer, coordination between pools, and a larger failure surface. Our question is not *Is disaggregation faster?* but *For which traffic and user-experience targets does the added complexity earn its keep versus a well-tuned monolith?*

3.4.1 Critical Misconception: Full Weights Do Not Equal Duplicated Work

A first reading of Dynamo often produces the same objection: If both pools load the full model and neither disables layers, isn't that just duplicating work, and paying twice for VRAM, for no reason?

Separate two ideas:

Concept	What It Is	Duplicated Across Pools?
Model Weights	Static network parameters (the assembly line)	Yes, each worker loads a full TP-sharded copy
Inference Work	FLOPs / memory traffic for a given request	No, prefill runs once; the decode loop runs once

Dynamo duplicates weights, not work. Transformers have no front half of prefill layers and back half of decode layers. Every layer runs in both phases. Because every layer is needed for both jobs, each pool must hold the full network. What differs is job assignment, data shape, and the software loop, not which layers are powered on.

3.4.2 Same Layers, Two Jobs

Think of the weights as one assembly line (Layers 1-N). Prefill shoves the entire prompt down that line, in a single pass for a short prompt, or a handful of chunked passes for a long one, see Chapter 3.2. Each layer saves its notes (KV for that layer) and, at the end, the model emits the first new token. The engine then stops. It does not enter the autoregressive loop. Decode receives those notes via NIXL and runs the same line one token at a time, consulting and appending to the imported KV so it never re-prefills the prompt.

Nothing is amputated:

- Prefill is prevented from decoding by halting after one heavy pass and exporting KV.
- Decode is prevented from prefilling by never being handed a raw massive prompt, only imported KV and the next token.

Figure 3: Nothing is Amputated

```
Wrong mental model                          Correct mental model

Prefill worker: layers 1..K                 Prefill worker: FULL weights
Decode worker:  layers K+1..N               Decode worker:  FULL weights
      (layers amputated)                    Same layers; different data shape
                                            + different software loop
```

Parameter	Prefill Worker	Decode Worker
Model Weights	Full model (TP-sharded)	Full model (TP-sharded)
Layers Used	All (one bulk pass)	All (one-token loop)
Typical Input	Entire prompt	Imported KV + next token
Typical Output	KV blocks (+ first token)	Streamed tokens
Scheduler Focus	Large batches, FLOPs	Many sequences, KV residency

The VRAM cost of full weights on both pools is real. The bet is that phase isolation and independent sizing are worth it for some traffic shapes, which this paper measures.

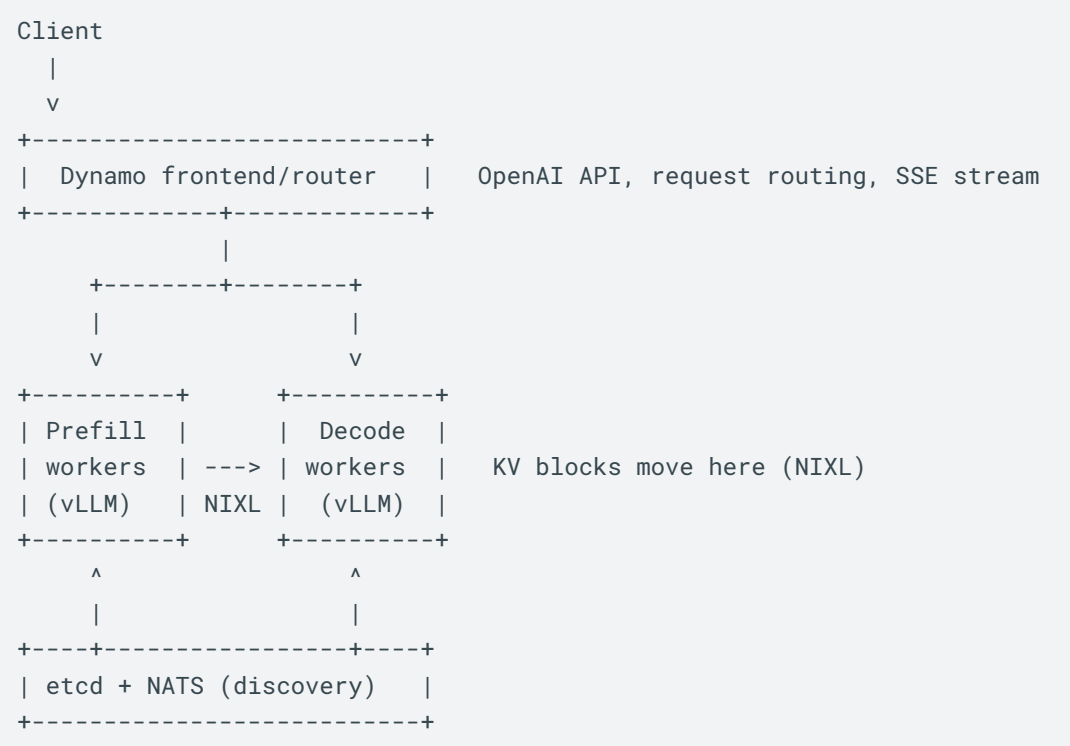
3.5 NVIDIA Dynamo: Operator Primer

NVIDIA Dynamo is the serving framework we use for disaggregated and monolithic inference behind an OpenAI-compatible API. There are four cooperating planes:

1. Front end/router: OpenAI HTTP (/v1/chat/completions, /v1/models). Admits the request, assigns prefill and decode workers, and streams tokens. Clients never talk to NIXL.
2. Prefill worker pool: vLLM engines that run the prompt phase and publish KV blocks.
3. Decode worker pool: vLLM engines that import those blocks and run the autoregressive loop.
4. Control/discovery plane: etcd and NATS (in our deployment) for worker registration.

Dynamo is not a different transformer. Engines are still vLLM on the same checkpoint. Dynamo adds orchestration and transfer glue: roles, routing, and the NIXL-backed KV handoff so the client sees one logical completion.

Figure 4: Dynamo, Disaggregated: Logical View



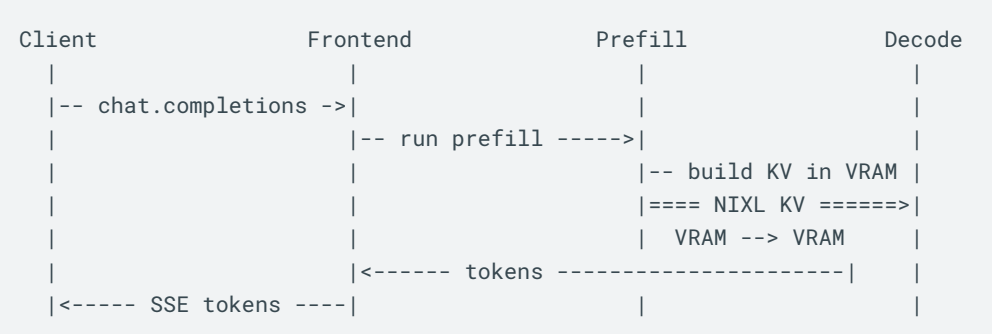
The Dynamo shape in this paper is one instance of that graph (1 prefill + 1 decode worker, each TP = 4; see Chapter 5.1). The Mono-vLLM baseline collapses the graph, running the model with no prefill/decode split on the same eight H100s.

3.6 The Disaggregated Request Path

Routing policy and metrics are detailed in Chapter 5 and Chapter 6. Here is what actually moves where.

The client POSTs to the front end, which queries etcd/NATS, assigns one prefill and one decode worker, and starts the client's TTFT clock. The prefill worker pushes the whole prompt through every layer, writes each layer's KV into local GPU VRAM, emits the first token, and stops without entering the autoregressive loop. NIXL then moves that KV from prefill VRAM to decode VRAM. What crosses the wire is computed tensor state plus the continuation, not the prompt string and not the model weights; on a single node NIXL prefers GPU-to-GPU paths (CUDA IPC over an NVLink-class fabric) rather than bouncing through host memory. The decode worker already holds full weights, so it never rereads the prompt as a bulk prefill: it feeds the continuation token into Layer 1, consults and appends to the imported KV, and loops until stop, growing KV locally on its own side. Tokens stream back through the front end, and both sides release their KV blocks on completion. The client saw one `/v1/chat/completions` call; the split was internal.

Figure 5: One Request, Disaggregated



If that handoff stalls, the user feels it as TTFT even when the prefill GPUs are already idle. That is why transfer cost and fabric health are first class in our methodology.

3.7 The KV-Transfer Fabric: NVIDIA NIXL

NVIDIA Inference Xfer Library (NIXL) is the fabric that moves KV between pools. After prefill, weights are already local on both sides; what must cross the wire is the per-request KV cache. Volume scales with layers × heads × hidden size × prompt tokens × precision, so long prompts enlarge the handoff; ISL-heavy profiles stress this differently.

NIXL is a four-layer abstraction over back ends including UCX (RDMA/GPU-Direct/CUDA IPC), Libfabric (AWS EFA), GDS (GPU-to-NVMe), and POSIX (file). Frameworks share one transfer API rather than each inventing RDMA plumbing. Adoption across Dynamo, SGLang, and vLLM (and AWS NIXL-with-EFA, 2026) marks KV transfer as a data center primitive.

Our 8 × H100 workers use `hostNetwork` / `hostIPC` so NIXL can prefer fast local GPU paths. We confirmed this directly rather than assuming it from configuration: worker logs show Backend UCX was instantiated (not an alternative NIXL back end), with the transport list including `cuda_ipc`. That is CUDA's mechanism for direct peer GPU memory access that does not stage through host RAM or route through the CPU, so a KV transfer between two GPUs on the same node's NVLink fabric moves GPU-to-GPU without the CPU acting as a middleman. Live Prometheus counters (`vllm:nixl_bytes_transferred`) confirmed real transfer activity under load, evidence that the fabric is actively moving data, not merely configured. The counter's average rate, bytes moved divided by wall-clock, is low because transfer is bursty and duty-cycled per request. It is not a measure of the fabric's peak bandwidth, which on an intra-node cuda_ipc/NVLink path is orders of magnitude higher and, as our results confirm, never the binding constraint, see Chapter 7.2 and Chapter 8.5. Multi-node RDMA is the same API with a different back end. That is out of scope here, but relevant in Chapter 10.

NIXL is not a model server, not a weight-delivery path, and not free: transfer time sits on the path to first token and can fail under stress.

Figure 6: NIXL Memory

3.7.1 Priming

First-contact registration between a specific prefill and decode engine can cost seconds per new pair. A cold mesh would tax early requests and unfairly bias Dynamo against the monolith. Our valid-run sequence therefore primes the prefill × decode mesh until coverage telemetry is satisfied before recording capacity results, matching the steady state of an always-on service. Priming must succeed for the Dynamo shape or the run is invalid. Transfer latency remains a real cost to account for even when bandwidth is often not the bottleneck in the large.

3.7.2 A Hybrid Architecture Changes What Crosses the Wire

Everything above assumes a pure-attention transformer, where the only per-request state is attention KV blocks. Nemotron-3-Super is a Mamba/Attention hybrid. Alongside attention KV, each Mamba layer carries two small, fixed-size pieces of recurrent state: a short causal-convolution window state and the state-space model (SSM) recurrent state. Neither is per-token history in the KV sense, but both must still cross from prefill to decode intact. NIXL's block-transfer abstraction was built for attention KV. Carrying Mamba state across the same boundary needed configuration that NVIDIA's own recipes do not seem to cover this model yet—the larger Nemotron-3-Ultra ships a disaggregated recipe; Nemotron-3-Super does not. One of the Section 6.9 fixes, the conv-state layout, targets the convolution state specifically, which is why the distinction matters.

3.8 Putting It Together

Aspect	Monolithic (Mono-vLLM)	Disaggregated (Dynamo)
Prefill/Decode Placement	Same engine	Separate pools
Weights/Layers	Full copy per replica (this paper: 2 × TP = 4)	Full weights on each worker; all layers both sides
What is Duplicated	None	Weights (VRAM), not per-request work
KV Path	GPU HBM	NIXL: prefill VRAM → decode VRAM
Scaling/Failure Surface	Scale whole replica; simpler	Independent P/D ratios: front end + two pools + NIXL mesh + priming

Disaggregation is a systems optimization (isolate phases, size pools, pay a transfer tax), not model compression and not simply half the network on each GPU. This perspective is important to keep in mind as we describe the specific shapes we test and the results.

3.9 What Better Means Here

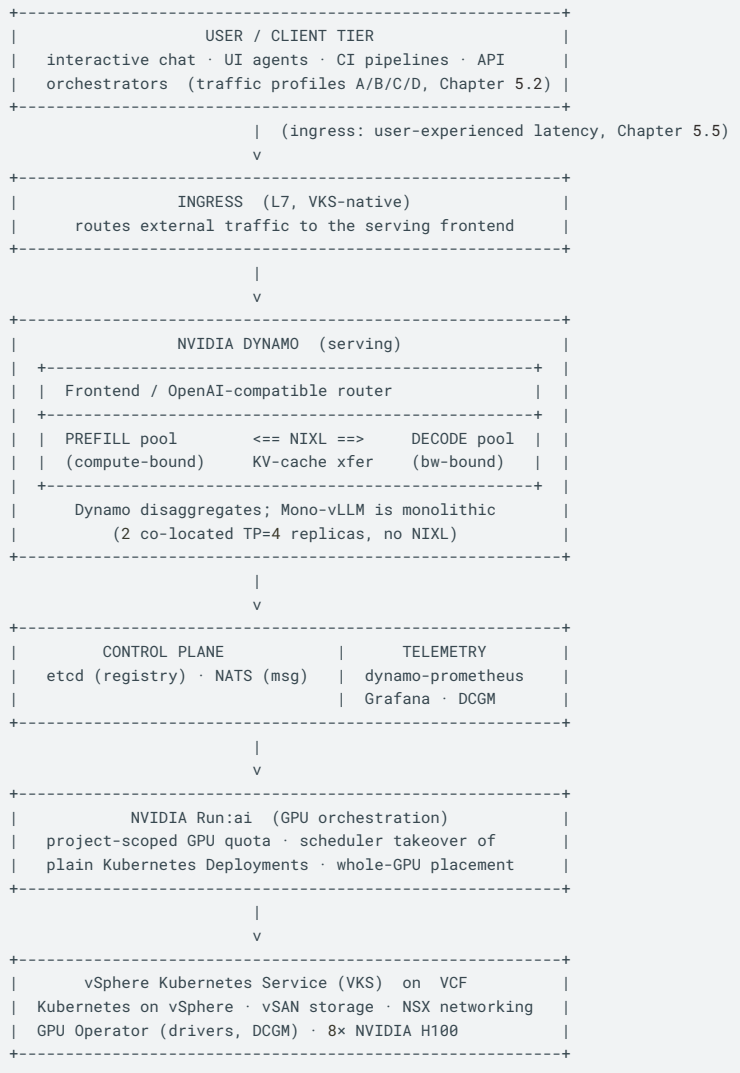
Because our lens is outcome-centered, we do not judge these architectures by peak tokens per seconds. We judge them by how many concurrent users/workflows of each profile a topology sustains within that profile's user-experience SLO (interactive latency for chat/UI users, agentic-completion bounds for API-orchestrator and CI workloads), including ingress latency.

Chapter 4: Platform: VCF + VKS + NVIDIA Run:ai + Dynamo

4.1 Why the Platform Is Part of the Finding

As Section 2 argues, enterprise AI economics live around the GPU: governance, multi-tenancy, storage, lifecycle, and operational maturity. A disaggregated serving stack has more moving parts than a monolith (separate pools, a control plane, a transfer fabric, per-shape telemetry), so the substrate that schedules, isolates, and observes those parts is not incidental to the result. It is what makes the comparison reproducible and the architecture operable. We run the entire study on VCF with VKS as the Kubernetes layer, with NVIDIA Run:ai for GPU orchestration and NVIDIA Dynamo for disaggregated serving over NIXL deployed on top of that platform. Together, this lets an enterprise run capable models on its own infrastructure, under its own governance and data control, rather than against a public API. That on-premises posture is the reason the substrate has to be enterprise-grade, and the reason we treat it as part of the finding rather than as plumbing.

4.2 The Stack



4.3 What Each Layer Contributes

Taken together, these layers make the study scriptable end to end: every topology is declared once, rendered to manifests, drift-checked against the declaration, and deployed or switched with a single command. That is a direct benefit of running on VKS rather than hand-managed hosts, and it is what lets a reader reproduce the campaign.

4.3.1 VCF (vSphere + vSAN + NSX)

The enterprise substrate: compute virtualization, software-defined storage for model/dataset volumes (PVCs), and software-defined networking. Across this study's multi-day redeploy cycle, we tore down and re-stood GPU workers many times, each time reattaching the same model PVC, reestablishing NSX-routed pod networking, and running 8 × H100 at full passthrough utilization for hours at a stretch. Through all of that, VMware vSphere® and vSAN did not lose a volume or drop a network path, so the substrate never became a variable we had to account for in the results. It supplies the governance, isolation, and lifecycle management that a bare GPU box lacks: the factory around the GPU.

4.3.2 VKS

Managed Kubernetes on VCF, and CNCF-conformant: every object in this study (deployments, services, PVCs, ingress, the GPU device-plugin's nvidia.com/gpu extended resource) is the standard Kubernetes API surface, not a VMware-specific abstraction. In practice this meant every tool the study depends on (Run:ai's scheduler, the NVIDIA GPU Operator, Dynamo's own Kubernetes deployment path, kubectl) worked against VKS as documented for upstream Kubernetes, with no VKS-specific or Kubernetes-platform-specific workarounds in the harness. We got declarative deployment of both serving shapes, the ingress path users actually traverse, and the standard operational surface the study is built on.

NOTE: To be precise about scope: the five fixes in Chapter 6.9 were serving-runtime/model-enablement issues in Dynamo/vLLM for a Mamba-hybrid model, not platform workarounds; VKS ran them unmodified.

4.3.3 NVIDIA GPU Operator

Drivers plus DCGM GPU telemetry, the per-GPU utilization/VRAM/NVLink series our measurement layer consumes.

4.3.4 NVIDIA Run:ai

GPU orchestration and, concretely, what made repeated redeploys reliable: project-scoped GPU quota enforcement meant that scaling a superseded shape's deployments to zero reclaimed its GPUs before the next shape's pods were admitted, every time, across many switches. As a result, we did not hit a GPU-deadlock (two shapes' pods holding overlapping devices), a failure mode a hand-rolled scheduler integration would have been prone to at this redeploy cadence. See Chapter 6.3 for the exact integration mechanics: how Run:ai is wired to plain Kubernetes Deployments rather than its own runai submit CLI. That wiring is not something most Dynamo-on-Kubernetes guides document, and it is one of the paper's more portable operational takeaways.

4.3.5 NVIDIA Dynamo + NIXL

The serving layer under test: an OpenAI-compatible front end/router, prefill and decode worker pools, and the NIXL KV-transfer fabric. This disaggregated shape is what we call Dynamo in this paper's comparison. The Mono-vLLM baseline collapses this to colocated monolithic vLLM replicas with no NIXL transfer in play.

4.3.6 Control Plane and Telemetry

etcd and NATS (as StatefulSets) coordinate workers; a dedicated `dynamo-prometheus` instance (not a cluster-wide monitoring stack) scrapes the serving-level and GPU-level series that all SLO evaluation and scoring depend on.

Chapter 5: Methodology

5.1 Two Serving Shapes

This paper compares two deployment shapes: Mono-vLLM, the monolithic baseline, prefill and decode colocated, no split; and Dynamo, the disaggregated shape, separate prefill and decode pools joined by NIXL, the capability that distinguishes NVIDIA Dynamo from plain vLLM. Both are deployed from the same underlying Dynamo runtime container; Mono-vLLM names the architectural pattern it exercises (no disaggregation), not a different image.

Nemotron-3-Super-120B-A12B-FP8 is a ~120-billion-parameter Mamba/Attention-hybrid MoE. Its per-GPU memory footprint at FP8 makes tensor-parallel size TP = 4 a hard floor rather than a design choice (TP = 2 leaves too little headroom for KV/Mamba-state alongside ~60 GB of weights per GPU on an 80 GB H100), so with 8 GPUs on this node there is exactly one disaggregated shape that physically fits: 1 prefill (TP = 4) + 1 decode (TP = 4), which is Dynamo. That single-shape constraint is itself a finding for operators on similar hardware. The monolithic baseline, Mono-vLLM, plays the analogous role (no prefill/decode split, full node) but is not one engine: NVIDIA's own recipe for this model runs it as 2 colocated TP = 4 replicas (with expert-parallel enabled) load-balanced by the front end, not one TP = 8 process. Every shape is declared once in a single source-of-truth YAML and rendered to Kubernetes manifests. Dynamo is NIXL-primed to full mesh coverage before any measurement; Mono-vLLM skips priming (no cross-pool transfer).

This is a best-realistic-deployment comparison, not a flag-identical control, and the asymmetries favor the monolith. We deliberately run each shape in the strongest configuration actually available to it, which is what an operator would deploy, rather than crippling one to match the other. That choice has a consequence: Mono-vLLM runs with prefix caching and multi-token prediction (MTP) speculative decoding enabled, and Dynamo does not, not by choice, but because both features are currently incompatible with this Mamba/Attention-hybrid model across a NIXL disaggregation boundary. Both features accelerate decode, which is exactly where Dynamo turns out to be bottlenecked. A third asymmetry runs the same direction: Mono-vLLM is graded against a looser profile-D latency ceiling than Dynamo. All three favor the monolith on the axis that decides the result. We do not neutralize them, because losing prefix caching and speculative decoding is a genuine, current cost of disaggregating this model class, not an artifact to be tuned away. Still, the reader should hold them in mind wherever we compare the two shapes' raw numbers.

5.1.1 Model Roster and Provenance

The quantitative comparison in this paper covers Nemotron-3-Super-120B-A12B only. The harness itself was built and iterated earlier on gpt-oss-20b/120b, which shaped the topology-rendering pipeline and the SLO-calibration approach described below. That exploration is not a data point here, because it ran at a shorter, less realistic generation cap:1024 tokens versus the 8192 used here, and it predates the harness-hardening pass. The two are simply not comparable numbers, and we prefer to say so than to argue the gap probably doesn't matter. We also attempted the Gemma-4-31B-IT model, which we excluded entirely after it proved incompatible with our Dynamo/vLLM version across two disaggregated shapes and repeated instability. This single-model, single-node scope is a real limitation; within it, the value is depth and honesty rather than breadth.

5.2 Traffic Profiles and Workload Realism

The load blends four user profiles, each modeling a class of real LLM consumer with a distinct arrival process chosen from traffic-modeling research rather than a convenient Poisson assumption. Profile weights are hot-reloadable per phase.

Profile	Represents	Arrival Model, as Implemented
Profile A: API Orchestrator	Autonomous agent swarms issuing parallel tool calls	2-state ON/OFF renewal burst generator: lognormal idle dwell > a burst of concurrent requests (discrete fan-out), then an exponential gap back to idle
Profile B: Terminal Siege (CI/CD)	Pipelines dumping large log blocks	Gamma, shape k≈0.444 > coefficient of variation CV = 1.5 (bursty, > Poisson)
Profile C: UI Navigator	Multimodal UI-driving agents	Exponential inter-arrival (independent actors)
Profile D: Knowledge Worker	Human interactive chat	Exponential inter-arrival (read-pause-respond pacing)

5.2.1 Grounding

Paxson & Floyd's foundational result established that wide-area traffic is not Poisson: bursts are self-similar and arise from aggregating heavy-tailed ON/OFF sources. That result justifies the bursty, correlated models for profile A and profile B, while exponential inter-arrival remains defensible for the independent human/agent actors in profile C and profile D. Profile A is a renewal ON/OFF burst process, and we describe it as such rather than as an MMPP.

5.2.2 Prompt Realism / Anti-prefix-cache Masking

Prompts are replayed from large real-world corpora (ToolBench, Terminal-Bench, OmniACT, SWE-rebench, CoderForge), sharded across load-generator workers by a hash of the session id. Using the full, high-cardinality corpus (not a small repeated subset) is deliberate: low-cardinality prompts drive vLLM's prefix cache to artificially high hit rates, masking true prefill cost and hiding KV-cache pressure, the very phenomena the disaggregation comparison must measure honestly. This is a methodological through-line from SPOC and Envoy.

5.2.3 Deterministic, Fair Sharding across Topologies

Every worker's corpus shard is selected by `crc32(session_id) mod worker_count`, keyed on a stable identifier embedded in the corpus itself, with a fixed worker count across every run. This means every topology in every comparison draws the identical session set from the corpus by construction, not by convention. We verified that directly against the sharding code rather than assuming it, since it underwrites every cross-topology comparison in Chapter 7.

5.2.4 Prompt vs. Completion Length Are Different Axes

Per-profile corpus statistics (input context: mean 2.3k tokens for profile A, 7.4k for profile B, 107 for profile C, 32.7k for profile D, up to 115k for profile D at p99, reflecting profile D's long multi-turn coding sessions) describe prefill burden, not decode load. A short prompt, such as `summarize X in one line`, and a long one can drive comparably long completions, and vice versa. We measure the prompt:completion ratio empirically per profile from live server-observed completion lengths rather than inferring decode-heaviness from prompt size alone.

5.2.5 Injected Traffic

The histograms below (6000-request random-offset samples per profile, drawn from the corpora themselves) show each profile's input-context distribution directly, rather than leaving mean 2.3k / 7.4k / 107 / 32.7k as bare numbers. Profile C is overwhelmingly short single-turn UI actions. Profile A clusters tightly in the low thousands (agentic tool-call context). Profile B is bimodal, split between short and 5-20k-token log-heavy turns. Profile D is the outlier, with over 40% of its context in the 32-64k range and a real tail past 100k; those are the long multi-turn coding sessions driving the p99 figure above.

Figure 7: Traffic Histograms

```
Profile A (API orchestrator), n=6000
      <=500 |                              0 ( 0.0%)
     0.5-1k | ##                         246 ( 4.1%)
       1-2k | ###############            2211 (36.9%)
       2-5k | ######################     3495 (58.2%)
      5-10k |                             48 ( 0.8%)
     10-20k |                              0 ( 0.0%)
     20-32k |                              0 ( 0.0%)
    32-131k |                              0 ( 0.0%)

Profile B (Terminal siege), n=5999
      <=500 |                              0 ( 0.0%)
     0.5-1k |                              0 ( 0.0%)
       1-2k | ####                         599 (10.0%)
       2-5k | ######                       852 (14.2%)
      5-10k | ################           2607 (43.5%)
     10-20k | #############              1941 (32.4%)
     20-32k |                              0 ( 0.0%)
    32-131k |                              0 ( 0.0%)

Profile C (UI navigator), n=6000
      <=500 | ##################   5603 (93.4%)
     0.5-1k |                             0 ( 0.0%)
       1-2k | ##                         397 ( 6.6%)
       2-5k |                             0 ( 0.0%)
      5-10k |                             0 ( 0.0%)
     10-20k |                             0 ( 0.0%)
     20-32k |                             0 ( 0.0%)
    32-131k |                             0 ( 0.0%)

Profile D (Knowledge worker/coder), n=12000
      <=500 |                              0 ( 0.0%)
     0.5-1k |                              0 ( 0.0%)
       1-2k |                              0 ( 0.0%)
       2-5k | #                          265 ( 2.2%)
      5-10k | #                          333 ( 2.8%)
     10-20k | ######                     1923 (16.0%)
     20-32k | #############              3994 (33.3%)
     32-64k | ################           4986 (41.5%)
    64-131k | ##                          499 ( 4.2%)
```

5.2.6 Generation Length

Each turn's `max_tokens` is fixed at 8192, chosen to accommodate the long-form outputs these workload types (agentic coding fixes, knowledge-worker reports) plausibly require without letting a single generation run unbounded; requests that would naturally continue past that point are truncated (`finish_reason=length`), a bound we disclose because it caps the decode compute a single request can demand in our measurements.

5.3 SLO and QoS Calibration

Because the paper is outcome-centered, the primary result for each shape is how many concurrent users/workflows it sustains within each profile's user-experience SLO (its QoS capacity), not its peak throughput. SLOs are per-profile because the definition of acceptable differs by work type: an interactive chat user notices token smoothness (inter-token latency, ITL), while an API orchestrator cares about task completion latency (time-to-first-token, TTFT, plus total time).

5.3.1 Per-Profile SLO Ceilings, as Configured

These values encode does it still feel interactive/does the agent finish in time, not an engine spec.

- Profile A: TTFT p99 ≤ 1000 ms
- Profile B: ITL p99 ≤ 40 ms
- Profile C: ITL p99 ≤ 30 ms
- Profile D: ITL p99 ≤ 50 ms

5.3.2 One Deliberate, Disclosed Asymmetry

Profile D's ceiling for Mono-vLLM is notable. Mono-vLLM is graded against a looser profile-D ITL ceiling of 115 ms, not the 50 ms applied to Dynamo. The rationale, which is carried over from earlier tuning, is that a colocated engine's prefill/decode co-batching imposes a measured practical ITL floor around 95 ms for profile D, so 50 ms is not physically reachable for the monolith and 115 ms adds approximately 20% headroom over that floor. This is a real relaxation of the bar for the monolith, and it runs the same direction as the prefix-caching/MTP asymmetry, so we flag it rather than bury it. Crucially, it does not change the ranking: Mono-vLLM violates even the looser 115-ms ceiling (profile D hits 220 ms at its lock point), so the finding survives the concession. Against the shared 50-ms ceiling, the monolith's profile D margin would only look worse.

5.3.3 Calibration, Not Fixed Thresholds

Gates are evaluated relative to a live baseline captured at each phase's low-concurrency Step 0 (per-domain: prefill and decode baselines tracked separately, never blended). A safe expression evaluator compares live metrics to baseline (e.g., `metric > max(1.0, 2.0×baseline)`); if a baseline reads exactly 0.0, empirically observed idle minimums are substituted rather than dividing by zero. Gates trip only after sustained breach (N consecutive polls, hysteresis) to reject transient noise, and gate state is reset at each phase boundary.

5.3.4 Capacity Determination

As concurrency ramps, a shape's QoS capacity for a profile is the highest sustained user count before that profile's SLO is violated for the required consecutive steps (capacity locking). This per-profile sustainable workflows within SLO number is the headline metric.

5.3.5 Saturation Gates

Alongside the SLO ceilings, the harness runs a set of architectural saturation gates that stop a ramp when a resource is exhausted, independent of whether an SLO has yet been breached. Two are latency-shaped: a compute boundary (rolling p99 TTFT degrading super-linearly, > 3× baseline) and a memory boundary (KV-cache preemption rate > 5%, a hard-stop safety trip). Two more are utilization-shaped and transfer-shaped and are the ones that actually govern our results: a GPU-utilization gate (tripped when a worker pool sustains ≥ 95% GPU utilization across the hysteresis window) and, for the disaggregated shape only, a handoff-latency gate (NIXL transfer p99 exceeding `max(1.0 s, 2× baseline)`). A utilization gate is deliberately proactive: it can stop a ramp while client-observed latency is still nominal, marking the point of no remaining headroom rather than the point of user-visible failure, a distinction that matters a great deal when interpreting the two shapes' governing numbers in Chapter 7.2.

5.4 Composite Roll-Up

Our harness can also reduce a run to a single topology performance score (TP-Score), a roll-up of SLO-bounded capacity rather than throughput. We do not report one for this comparison, and the reason is worth stating because it recurs in any mono-versus-disaggregated study. The formula needs an independently measured prefill/decode capacity split. That split exists for Dynamo but not for a monolith, whose single unified saturation point conflates both roles by construction, and the formula's handoff-latency penalty structurally favors whichever shape has no transfer fabric. Forcing a number through it would manufacture false precision, so we report the two shapes' governing points and failure mechanisms directly instead.

5.5 Measurement: Server Side, Client Side, and the Ingress Tax

We measure latency at two vantage points and report both, because they answer different questions:

- Server side (intrinsic): Prometheus histograms at the Dynamo front end / vLLM workers (`dynamo_front end_*` for Dynamo; `vllm:*` for Mono-vLLM) give the serving system's intrinsic TTFT/ITL/TPOT, independent of where the client sits. This is the fair basis for comparing topologies.
- Client side (experienced): The load generator measures TTFT/ITL from the streamed response through the ingress, the latency a real user actually experiences.

The ingress tax is the difference (client minus server). We report it as a first-class pipeline component, because users pay it regardless of how fast the engine spins. Empirically the tax is small but real on ITL (order tens of ms) and is exactly the kind of cost around the GPU value a tokens/sec number ignores; it can push a tight interactive SLO from pass to fail, so per-profile SLO attainment is reported on the experienced (client-side) latency, with the intrinsic (server-side) number and the tax shown beside it.

5.5.1 NIXL Handoff Latency

The runtime exports the real KV-cache transfer time (`vllm:nixl_xfer_time_seconds`, plus `nixl_post_time_seconds` and failed-transfer counters) on the disaggregated workers, so we report the aggregate p99 transfer time as the handoff latency for Dynamo (peak 0.328 seconds during soak, as shown in Chapter 7.4) and 0 for Mono-vLLM (no NIXL, no handoff; a structural, not missing, value). Read that 0.328 seconds as a two-sided, modest-concurrency aggregate rather than a clean one-way transfer time. Even so, it lets us test, rather than assume, the recent claim that KV-transfer bandwidth is generally not the bottleneck. It is not. But Dynamo's prefill limit is nonetheless gated on handoff latency: transfer latency binds prefill at high concurrency even though bandwidth is ample, and decode's compute gate binds first.

Chapter 6: Experiment Setup and Reproducibility

6.1 Hardware, Model, and Software Versions

Reproducibility starts with disclosing exactly what we ran. A reader who wants the same results should be able to match this bill of materials.

- GPU node: 8× NVIDIA H100 (80 GB), single node, NVLink-connected, presented to the VM via VMware® ESX® GPU passthrough (PCI passthrough / DirectPath I/O), whole-GPU (no vGPU/MIG partitioning).
- Model: nvidia/NVIDIA-Nemotron-3-Super-120B-A12B-FP8 (FP8 weights on a shared PVC, mounted read-only at /model).

Table 1: Test Parameters

Layer	Component	Version
Foundation	VMware Cloud Foundation (vSphere + vSAN + NSX)	9.1
Kubernetes	vSphere Kubernetes Service (VKS; formerly Tanzu Kubernetes Grid Service)	v1.35.5+vmware.1
Node OS/Runtime	Ubuntu	24.04.4 LTS, containered 2.2.3
Storage	vSphere CSI (vSAN)	provisioner csi.vsphere.vmware.com (default StorageClass)
GPU Enablement	NVIDIA GPU Operator	v26.3.3 (driver 580.126.20, container-toolkit v1.19.1, device-plugin v0.19.3)
GPU Telemetry	NVIDIA DCGM exporter	4.5.3-4.8.2 (scraped on :9400)
GPU Orchestration	NVIDIA Run:ai	control plane 2.24.66, scheduler/engine v3.37.27
Serving Runtime (all roles)	NVIDIA Dynamo runtime (vLLM backend)	nvcr.io/nvidia/ai-dynamo/vllm-runtime:1.3.0
KV-Transfer Fabric	NVIDIA NIXL (bundled with the Dynamo vllm-runtime)	as shipped in vllm-runtime:1.3.0
Control Plane	etcd / NATS	etcd v3.5.30, NATS 2.10
Observability	Prometheus / Grafana / NATS exporter	prom/prometheus:v2.45.0, grafana/grafana:10.2.0, natsio/prometheus-nats-exporter:0.15.0
Ingress	HAProxy Kubernetes Ingress Controller	haproxytech/kubernetes-ingress:3.2.9

Nemotron requires vllm-runtime:1.3.0 because of the Mamba/Attention-hybrid support, NixlConnector, and the MoE expert-parallel path used here. With 1.3.0, NVIDIA's own official recipe runs both the monolithic and disaggregated shapes from the same Dynamo runtime image (2 colocated TP = 4 replicas for the monolith, no separate stock vLLM image needed). That removed a whole axis of version drift between the two topologies being compared.

6.2 Design Decisions and Rationale

A few choices are non-obvious and deserve further explanation:

- Why vLLM, Not TensorRT-LLM or a NIM? The disaggregated runtime we deployed (`ai-dynamo/vllm-runtime:1.3.0`) is built on the vLLM inference back end, and that was the disaggregation path available to us at this Dynamo version. A TensorRT-LLM/NIM-based disaggregated runtime was not an option for us here, so to keep the comparison controlled we use vLLM for both the disaggregated workers and the monolithic

baseline. This surprises people who assume any NVIDIA serving stack implies TensorRT-LLM or a NIM container; it does not, and using one back end on both sides removes a major confounding variable from the disaggregated-vs-monolithic result.

- Why a dedicated `dynamo-prometheus`, not the cluster monitoring stack? A cluster-wide monitoring Prometheus does not scrape Dynamo's per-worker and front end jobs, so evaluating SLOs against it silently yields empty or irrelevant series. We deploy a dedicated instance whose scrape config is re-rendered per topology; this is a real, observed failure mode, not a hypothetical.
- Why drive load through an ingress? Real users reach the service through an ingress, so the client-side latency we report includes that path (the ingress tax referenced in Chapter 5.5). It is also the only concurrency-capable path in our topology: node internal IPs are not reachable from the client, and a `kubectl port-forward` drops connections under concurrent streaming load.
- Why the full multi-corpus dataset (never a subset)? High session cardinality is what keeps vLLM's prefix cache from being artificially warm; a truncated subset would inflate cache hit rates and flatter every topology equally, hiding the prefill cost the comparison depends on.

6.3 Building the Platform: From VCF to a Serving-Ready GPU Cluster

This is the layer-by-layer path we followed. Values in brackets are environment-specific; the versions above are what we validated against. For the VCF, Supervisor, and VMware NSX® provisioning steps, follow the VCF and VKS product documentation; we call out the decisions that matter for this workload rather than restating those guides.

1. VCF foundation: A VMware Cloud Foundation 9.1 domain providing vSphere compute, vSAN for software-defined storage, and NSX for software-defined networking. The 8× H100 are presented to the GPU worker in GPU passthrough mode (PCI passthrough/DirectPath I/O) on the ESX hosts, so each GPU is assigned whole to the VM. We use passthrough rather than vGPU/MIG partitioning because the study needs full, undivided GPUs: Mono-vLLM runs two colocated TP = 4 replicas spanning all 8 GPUs, and Dynamo pins whole GPUs per prefill/decode worker, so partitioning would confound the comparison.

2. Enable the Supervisor and create a VKS cluster. Turn on vSphere Workload Management (the Supervisor, reachable at `[supervisor VIP]`), then provision a VKS cluster (`runai-wld01-cluster`, Kubernetes v1.35.5+vmware.1, Ubuntu 24.04 node image).
   Node plan: a control-plane node plus at least one GPU worker sized to hold all 8 GPUs. Log in non-interactively with `kubectl vsphere login`. The session token is short-lived; automate refresh for long runs.

3. Networking. NSX supplies the pod and service networks. Two facts shape the load-test path: node InternalIPs sit on a cluster-internal subnet not reachable from an external client, and a plain port-forward is single-connection. Expose the serving front end through the HAProxy ingress controller on the ingress LB subnet the client can reach [e.g. 10.192.134.x], with streaming-friendly timeouts for SSE.

4. Storage. The vSphere CSI vSAN StorageClass backs every PVC: the model volume (Nemotron-3-Super-120B-A12B-FP8 weights at the PVC root), etcd and NATS state, and the load-generator dataset volume. Use `allowVolumeExpansion` so volumes can grow.

5. GPU enablement. Install the NVIDIA GPU Operator (v26.3.3), which lays down the driver (`580.126.20`), the container toolkit, the Kubernetes device plugin, and the DCGM exporter (the per-GPU utilization/VRAM/NVLink telemetry the study consumes on `:9400`). Confirm the device plugin advertises 8 allocatable GPUs on the worker.

6. GPU orchestration, the Run:ai way. Install the NVIDIA Run:ai (2.24.66) cluster components and bind the serving namespace (`dynamo`) to a Run:ai project with a GPU quota covering all 8 devices. From that point Run:ai's scheduler, not the stock Kubernetes scheduler, arbitrates every GPU pod: it places each worker against the project quota onto the passthrough GPUs the GPU Operator's device plugin advertises. What matters for reproducibility is how we hand workers to Run:ai:

- Declarative manifests, not `runai submit`. A topology is not one job; it is a coupled set (the front end/router, N prefill workers, M decode workers, plus the etcd/NATS control plane) that must come up together, be drift-checked against the source of truth, and be switched atomically between shapes. So instead of issuing per-workload `runai submit` CLI calls, we render each worker as a Kubernetes Deployment and apply the whole set with `kubectl apply -f workers.yaml` into the Run:ai project namespace. This is a documented, first-class Run:ai path, not a workaround: Run:ai lists deployment among the standard Kubernetes workload types it supports via YAML, and its scheduler is designed to take over third-party workloads submitted directly to Kubernetes. Run:ai schedules our deployments exactly as it would a CLI-submitted workload, because they are ordinary GPU-requesting pods in a Run:ai-managed namespace; we simply keep the shape in version-controlled YAML rather than in shell history. We confirmed this on the running cluster: every serving pod is placed by `schedulerName: runai-scheduler` (injected for the project namespace), not the default kube-scheduler.
- Whole-GPU requests + container-local pinning. Each worker Deployment requests whole GPUs with an integer `nvidia.com/gpu` request/limit (equal to its `tensor_parallel_size`: 4 for each of Dynamo's prefill and decode workers, 4 for each of Mono-vLLM's two colocated replicas). Note a subtlety that bites reproducers (see Chapter 6.9, fix 1). The physical gpus: list in the YAML is a logical/quota declaration of how many GPUs a worker gets, but `CUDA_VISIBLE_DEVICES` inside the pod must be set to container-local indices `0..TP-1`, not the node-physical indices, because Run:ai's device plugin presents each pod only the GPUs it was allocated, re-indexed from 0. The renderer emits `0,1,2,3` for every TP = 4 worker regardless of which physical devices the scheduler picked; setting node-physical indices (`4,5,6,7`) fails with `NVMLError_InvalidArgument`. There is no fractional-GPU or MIG sharing, consistent with the whole-GPU passthrough decision in Step 1 above.
- Readiness is gated, not assumed. A single topology-start command renders the manifests, asserts the running shape matches the declaration, brings up the control plane (etcd/NATS as StatefulSets), applies the workers, and then blocks on `kubectl rollout status` for every deployment before the topology is considered serving-ready. Switching topologies scales the superseded topology's deployments to zero before applying the next one's, so transitions are clean and the control plane is never disturbed, and the GPUs are actually reclaimed before the next topology's pods are admitted. This is Run:ai's quota accounting doing real work, not a formality.

The net effect is the Run:ai operational model (project-scoped GPU quota, Run:ai scheduler placement, device-plugin-advertised passthrough GPUs) driven by declarative, drift-checked YAML rather than imperative CLI submits, which is what makes an unattended multi-shape sweep reproducible.

7. NVIDIA Dynamo control plane and serving. Deploy the control plane (etcd and NATS as StatefulSets), then the Dynamo front end/router and the prefill/decode worker pools, all from the single `ai-dynamo/vllm-runtime:1.3.0` image. Mono-vLLM's two replicas run from the same image, just without `NIXL/kv-transfer-config` wired in. Every shape is rendered from the single source-of-truth YAML. The component layout (front end/router in front of separately scaled prefill and decode pools joined by the NIXL KV-cache transfer) follows the NVIDIA Dynamo Kubernetes deployment and disaggregated serving guides; we render those same components from our YAML rather than hand-editing per topology. Because all workers are coscheduled on the single GPU node (a `podAffinity` hostname rule), the NIXL transfer stays intra-node over `cuda_ipc/shared-memory` transports rather than requiring cross-node RDMA.
8. Observability. Deploy the dedicated `dynamo-prometheus`, Grafana, and the NATS exporter, and reuse the GPU operator's DCGM. The scrape config is re-rendered per topology so worker ports always match the deployed shape (Section 6.2).

At that point the cluster is serving-ready, and the valid-run sequence produces a comparable measurement.

6.4 Single Source of Truth Yields Rendered Manifests

Every shape's GPU pinning, TP size, ports, and NIXL priming counts live in one YAML. Deterministic renderers turn that into Kubernetes manifests and per-shape Prometheus scrape configs; a runtime assertion refuses to proceed if the rendered/running shape (GPU/TP/port/priming layout) drifts from the declaration. That keeps the two deployments from silently diverging in their hardware placement, but it does not, and cannot, make them identical in their serving-feature flags. This is a controlled comparison of deployment shape, deliberately run at each shape's best-available configuration, not a flag-identical A/B test.

NOTE: For this model the Mono-vLLM manifest is hand-authored to mirror NVIDIA's official 2 × TP = 4 recipe, and the disaggregated worker manifests carry the five Chapter 6.9 fixes by hand, so the render-and-drift-check machinery is a framework rather than a fully hands-off generator here.

Concretely, this is the declaration of the disaggregated shape (Dynamo): a shared header (served model name, image, router/Prometheus endpoints) followed by the topology block that pins each role to whole GPUs and fixes its ports and NIXL priming counts. Mono-vLLM is a sibling block under the same file with `kind: monolithic` and no NIXL priming section.

Figure 8: Excerpt of the Topology Declaration with Shared Defaults and the Disaggregated Dynamo Shape Defaults Follows:

```
defaults:
  router_url: "http://10.192.134.77"              # haproxy Ingress, not port-forward (Section 6.3)
  prometheus_url: "http://127.0.0.1:9090"
  served_model_name: "nvidia/NVIDIA-Nemotron-3-Super-120B-A12B-FP8"
  model_path: "/model"
  dynamo_image: "nvcr.io/nvidia/ai-dynamo/vllm-runtime:1.3.0"

dynamo_disagg:                                     # the disaggregated shape
  kind: disaggregated
  description: "1 prefill TP=4 + 1 decode TP=4 -- TP=4/role is a memory floor, not a preference,
    for this ~120B-param FP8 model on 80GB H100s (Section 5.1)."
  max_num_batched_tokens: 16384
  max_num_seqs: 128
  gpu_memory_utilization: 0.90
  priming: { n_prefill: 1, n_decode: 1, n_decode_actual: 1, n_decode_tp: 4 }
  prefill_groups:
    - { tp: 4, gpus: [0, 1, 2, 3], port: 8081, nixl_side_channel_port: 5601, nixl_exporter_port: 9501 }
  decode_groups:
    - { tp: 4, gpus: [4, 5, 6, 7], port: 8082, nixl_side_channel_port: 5602, nixl_exporter_port: 9502 }
```

Each worker gets a distinct system_port (health/metrics), a nixl_side_channel_port (KV-transfer coordination), and a nixl_exporter_port (the NIXL metrics the study consumes); the renderer derives the Kubernetes Deployment, Service, and the Prometheus scrape targets straight from these, so the running shape cannot silently diverge from the declaration.

6.5 The Valid Run Sequence

A data point counts only if produced by this exact sequence:

1. Declare: One topology YAML is the sole source of truth: GPUs, TP, ports, priming
2. Render: Generate Kubernetes manifests and Prometheus scrapes the config from it; Mono-vLLM's 2 × TP = 4 recipe is hand-authored; see Chapter 6.4
3. Assert: Running shape == declaration; refuse to proceed on drift
4. Control: etcd + NATS are healthy (StatefulSets) before any worker is ready

5. Deploy: Apply the whole topology as a set; block on rollout of every deployment
6. Prime: NIXL prefill x decode mesh to full coverage; Dynamo only, Mono-vLLM skips this step
7. Telemetry: Repoint the per-shape scrape config at the dedicated Prometheus
8. Verify: The front end is serving and all Prometheus targets are up, per shape
9. Load: Ramp concurrency > soak, evaluating per-profile SLO gates throughout

Deviations invalidate the run. Measuring before NIXL priming reports full coverage, or scraping a cluster-wide monitoring Prometheus instead of the dedicated one, are both treated as invalid; both are real failure modes we hit rather than hypotheticals.

6.6 Load Generation and Ingress

The load generator (Locust + a ramp controller) drives the four-profile blend, ramps concurrency, holds a soak, and evaluates per-profile SLO gates against `dynamo-prometheus`. Traffic reaches the serving front end through the VKS-native L7 ingress, the same path a real user takes, so client-side latency includes the ingress tax. We validated that the ingress sustains concurrent streaming load cleanly. A naive `kubectl port-forward` does not, and must not, be used for load, because it drops connections under concurrency and would inject spurious errors.

6.7 Endpoint and Credential Robustness

Two operational lessons are worth passing on to anyone running long unattended sweeps on this stack:

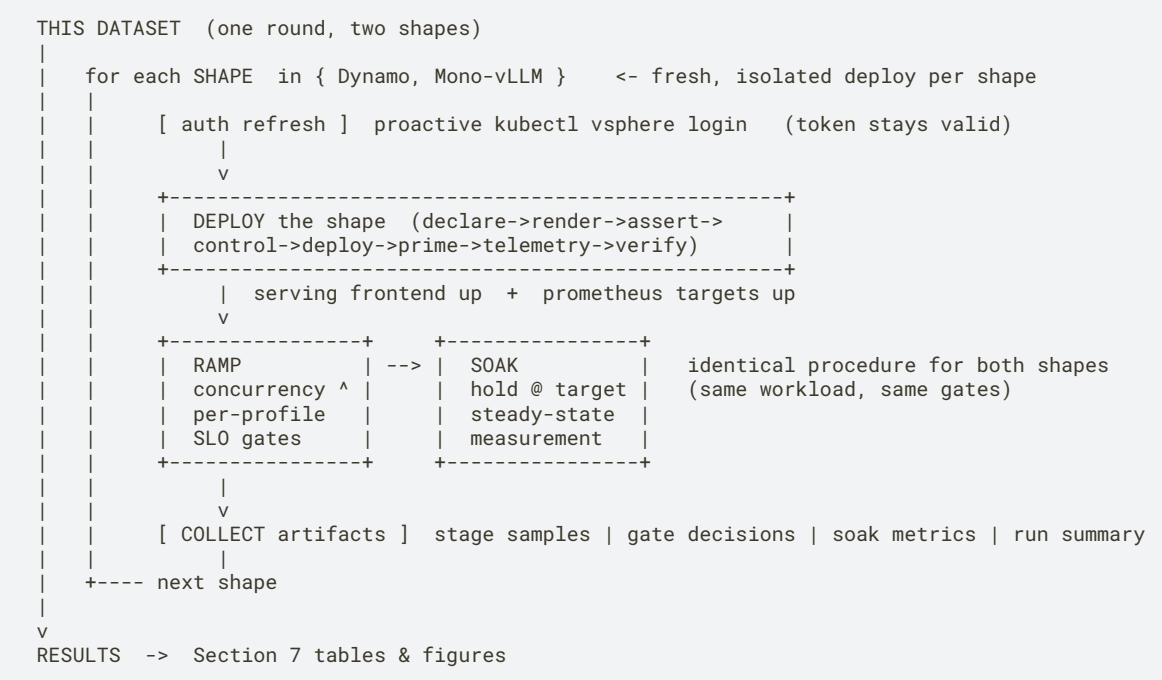
- GPU reclaim on redeploy is the caller's job. Scale the superseded shape's deployments to zero and confirm the pods are actually gone before applying the next shape. Do not rely on `kubectl apply` alone to sort out overlapping `nvidia.com/gpu` claims. Run:ai's project quota accounting reclaims correctly once the old pods are gone, but the sequencing (scale down, confirm, then apply) is not done for you, and skipping it risks two shapes' pods holding the same physical GPUs.
- Session credentials expire mid-run. A VKS admin kubeconfig carries a short-lived vSphere session token, so a multi-hour campaign must refresh it non-interactively at each shape switch or a deploy will fail partway through the sweep on an expired token rather than on anything to do with the workload.

6.8 Campaign Structure

Each shape is deployed fresh via the sequence above, then driven through an identical ramp (a prefill/decode ramp for Dynamo, a unified saturation ramp for the monolith, Section 5.1) followed by a soak, with per-run artifacts collected for Section 7. The whole flow is scriptable end to end (deploy, drift-check, ramp, soak, teardown), which is the reproducibility contribution of running on VKS rather than hand-managed hosts.

The harness supports unattended multi-round sweeps across many shapes. For this paper's dataset, however, we executed one round of two shapes, each measured once (Section 11.7 gives the reason; Section 12.2 treats a repeat campaign as future work):

Figure 9: Campaign Structure



Reading Figure 9 top to bottom: each shape walks the same deploy > ramp > soak > collect path, so the deployment procedure is identical between the two data points; the shapes differ in architecture and, as Chapter 5.1 discloses, in the decode-side serving features each can use.

6.9 Case Study: Disaggregating a Mamba-Hybrid Model (Dynamo)

This chapter describes a pure-attention transformer's disaggregated deploy. Nemotron-3-Super's Mamba/Attention hybrid architecture needed five additional, previously undocumented fixes to run as Dynamo. These fixes are worth recording in full because we could not find any NVIDIA disaggregated recipe for this model to check against, unlike the larger Nemotron-3-Ultra.

1. `CUDA_VISIBLE_DEVICES` must use container-local indices, not physical ones. Run:ai's device plugin presents a pod's allocated GPUs as local indices `0..N-1` regardless of which physical GPUs the scheduler picked. Setting the decode worker's `CUDA_VISIBLE_DEVICES` to the node-level physical range (e.g., `4,5,6,7`) fails with `NVMLError_InvalidArgument`, because that container only ever sees 4 GPUs, indexed `0..3`. Both prefill and decode workers use `0,1,2,3`, a general lesson for any multi-role GPU split on Run:ai, not specific to this model.
2. `PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True` is incompatible with NixlConnector. vLLM's own engine-config validator rejects this combination: expandable segments can remap KV-cache virtual addresses, which would invalidate NIXL's pinned/registered memory regions. Harmless to omit, since the monolithic baseline has no NIXL transfer and does not need this guard.
3. Mamba `conv-state` transfer requires an explicit layout. vLLM's own error was specific and actionable: 3-read Mamba conv transfer requires DS conv state layout. Set `VLLM_SSM_CONV_STATE_LAYOUT=DS`. Set on both roles.
4. Prefix caching is unsupported for Mamba-hybrid models across a NIXL boundary. `--enable-prefix-caching` (fine on the monolithic baseline, which has no cross-worker transfer) crashes decode with `AssertionError`: SSM can only have one local block, inside vLLM's own NIXL connector code, whose comment states the limitation outright: Mamba hybrid: no prefix caching support so far. Removed from both roles.

5. Speculative decoding (MTP) conflicts with cross-worker state continuation. A request whose first decode step arrives via NIXL transfer rather than local continuation crashes with an assertion inside vLLM's Mamba mixer forward pass. That assertion tracks previous step scheduling metadata that speculative decoding needs, but a freshly-transferred request never populates it. MTP was a monolithic-baseline performance optimization we mirrored by default. Removing it for Dynamo resolved the crash.

Each fix was root-caused from the actual error message or source comment before being applied. None were guessed. With all five in place, Dynamo runs stably: zero request failures observed across every ramp and soak run at this shape.

Concretely, the disaggregated decode worker's argument/environment block carries fixes 1 to 3 and the absence of the flags removed by fixes 4 and 5; prefill is identical except `--disaggregation-mode=prefill` and its own KV-events endpoint:

Figure 10: An Excerpt of the Dynamo Decode Worker, with the Five Fixes in Place:

```
env:
  - { name: CUDA_VISIBLE_DEVICES,        value: "0,1,2,3" }   # fix 1: container-local, not 4,5,6,7
  - { name: VLLM_SSM_CONV_STATE_LAYOUT,  value: "DS" }        # fix 3: Mamba conv-state transfer layout
  - { name: UCX_TLS,                     value: "tcp,cuda_ipc,cuda_copy,sm,self" }
  # fix 2: PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True is deliberately ABSENT (breaks NixlConnector)
args:
  - "--disaggregation-mode=decode"
  - '--kv-transfer-config={"kv_connector":"NixlConnector","kv_role":"kv_both"}'
  - "--tensor-parallel-size=4"
  - "--enable-expert-parallel"          # MoE expert-parallel: enabled on BOTH shapes
  - "--max-model-len=131072"            # capped from the model's 262K/1M ceiling to bound KV reservation
  - "--kv-cache-dtype=fp8_e4m3"
  - "--mamba-ssm-cache-dtype=float16"
  # fix 4: --enable-prefix-caching  is deliberately ABSENT (crashes Mamba-hybrid decode over NIXL)
  # fix 5: --speculative-config MTP is deliberately ABSENT (conflicts with cross-worker state handoff)
```

The connection to the comparison's fairness is exactly those last two ABSENT lines: Mono-vLLM keeps `--enable-prefix-caching` and the MTP `--speculative-config` (it has no NIXL boundary to break them), so the monolith enjoys two decode accelerators the disaggregated shape cannot use for this model. Expert-parallel and the FP8 KV-cache are enabled on both shapes; the reproducibility note in Chapter 6.4 applies; the Mono-vLLM manifest is hand-authored from NVIDIA's recipe.

Chapter 7: Results

7.1 QoS-Defined Capacity: The Joint Mixed-Traffic Lock Point

Because both shapes run all four profiles concurrently in a weighted mix (not one profile at a time), QoS capacity is a joint number: the highest sustained concurrency before the aggregate SLO check fails for three consecutive steps. This is the paper's primary metric.

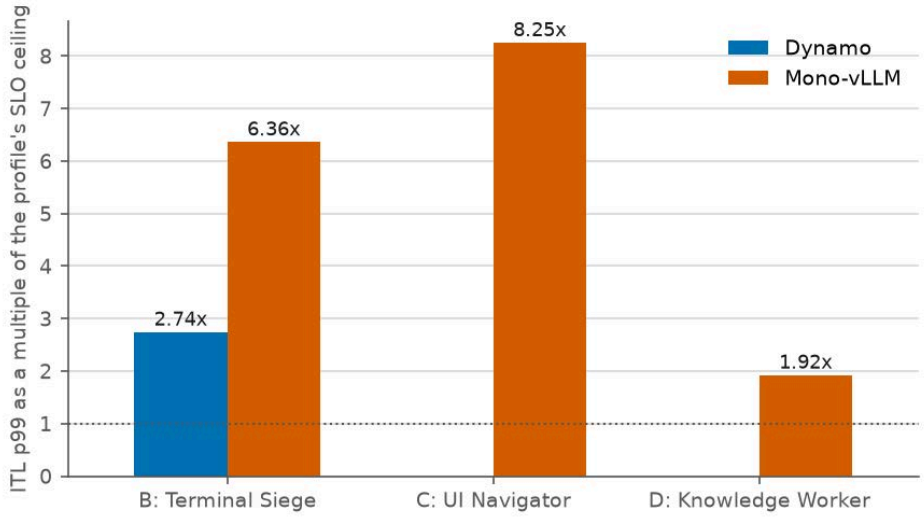
Shape	QoS-Locked Capacity	Which Profiles Failed, and by How Much
Dynamo (disaggregation)	40 users	Profile B: ITL p99 = 109.6 ms vs. 40 ms ceiling (2.74× over)
Mono-vLLM	80 users	Profile B: 254.4 ms vs. 40 ms (6.36× over) Profile C: 247.6 ms vs. 30 ms (8.25× over) Profile D: 220.3 ms vs. its looser 115 ms ceiling (1.92× over)

Neither shape clears the per-profile SLO ceilings at any concurrency tested. The two lock points are not equally bad, and the comparison actually favors disaggregation here despite the confounds working against it: Dynamo's violation is proportionally far smaller (2.74× vs. 6.36× over, on the same profile B metric), and it fails on one profile rather than three, as shown in Figure 11. That is the more striking because all three configuration asymmetries run the monolith's way here, and it still posts the wider, broader violation. One caveat on the one profile versus three point: the two shapes lock at different concurrencies (40 vs. 80 users), so this compares failure breadth at each shape's own lock point, not at matched load. We did not hold Dynamo at 80 users to see how many profiles it would then fail.

NOTE: Dynamo's QoS lock (40 users) and its decode saturation gate (40 users, Section 7.2) coincide. That is not a coincidence of rounding: the same ramp step that pushed profile B's ITL past its ceiling also drove the decode pool to its utilization gate, so for Dynamo the SLO limit and the architectural limit arrive together.

For Mono-vLLM, the two are far apart (QoS lock at 80, collapse at 805), which is itself informative: the monolith spends a very wide concurrency band in SLO violation while still serving requests, whereas Dynamo has essentially no such band on this hardware.

Figure 11: QoS-Ceiling Violation Margin at Each Shape's Own Lock Point



NOTE:

- The dotted line at 1.0 is the SLO ceiling.
- For profile B only, Dynamo locks at 40 users.
- For profiles B, C, and D, Mono-vLLM locks at 80 users.
- Blank bars indicate that profile did not fail for that shape.

7.2 Governing Saturation Point, and Two Very Different Failure Mechanisms

Beyond the SLO ceiling, each shape has an architectural capacity limit: the concurrency at which a saturation gate stops the ramp. The two limits are reached by different kinds of gate, so the numbers are not a like-for-like capacity ratio. Reading them correctly is the point of this section. Abbreviations in the table are as follows: decode limit point (DLP), prefill limit point (PLP), overall saturation point (OSP) for the monolith, which has no separate prefill/decode split.

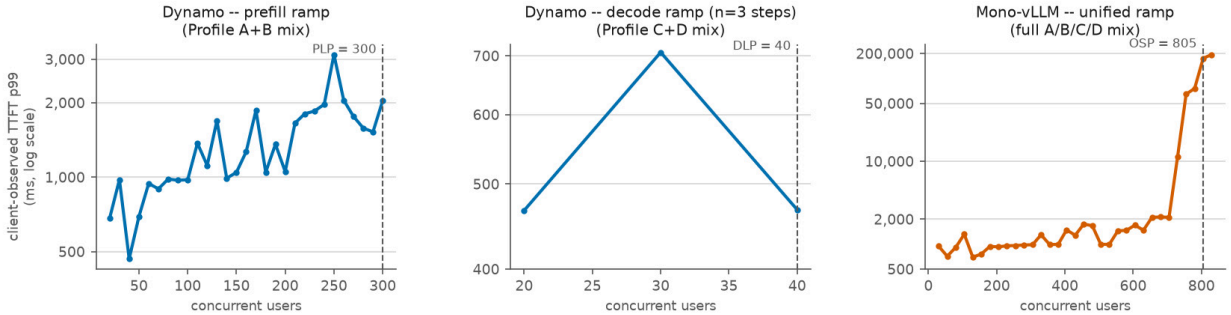
Shape	Governing Point	Gate that Stopped It	What Is Actually Happening
Dynamo (disaggregation)	40 users (DLP)	Decode GPU-utilization gate (decode pool ≥ 95% util)	A proactive headroom cutoff, not a failure. The harness stops the ramp because decode's 4 GPUs are saturated, but users are still well served: blended client ITL is ~24 ms and no SLO has collapsed. 40 marks where Dynamo runs out of decode headroom, *not* where it stops serving acceptably.
Mono-vLLM	805 users (OSP)	Actual collapse: Profile D exception rate 51.9% (138 exceptions/266 completions) at the 830-user step, GPU util pinned at 100%	A genuine, abrupt collapse. Client TTFT p99 jumps from ~2.0 seconds (705 users) to 43.6 seconds in a single step (730 users) and stays pinned in the tens-of-seconds range; by 830 users requests start failing outright, so the harness locks the last clean step (805) as OSP.

7.2.1 How to Read the Gap

Dynamo's 40 is a proactive utilization stop while it is still healthy; Mono-vLLM's 805 is an actual collapse point. They are different kinds of limits, so 805 vs. 40 is not a statement that the monolith serves 20× more usefully. It is failure-onset concurrency measured two different ways.

Figure 12 plots both ramps against client TTFT, where the monolith's collapse appears as a near-vertical step while Dynamo's ramps simply end at their gates well before any comparable region.

Figure 12: Concurrency vs. Client-Observed TTFT: Two Bounded Curves, One Cliff

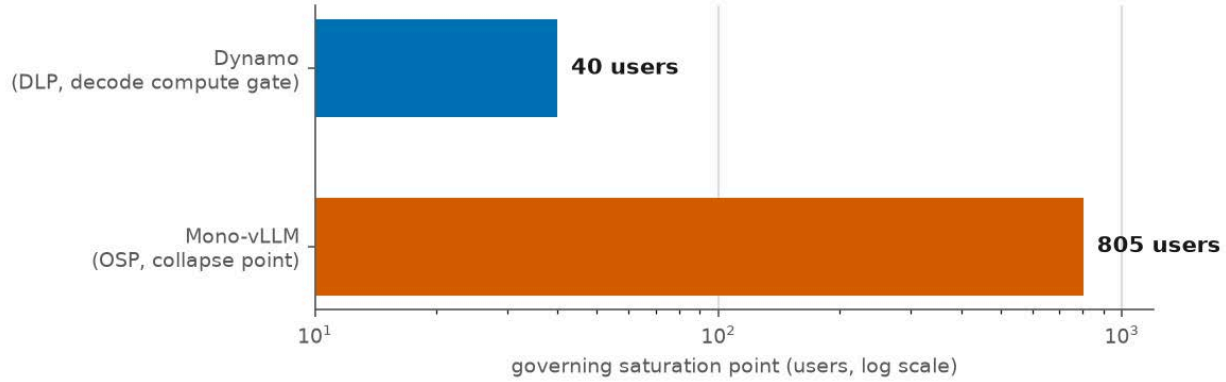


NOTE:

- Mono-vLLM's TTFT jumps ~20× in a dingle ramp step (705 > 730 users)
- Dynamo's two curves stay bounded; note the decode panel's narrow y-range: ~460 ms to 980 ms throughout because each gate stops the ramp proactively — on BGPU utilization, not on any visible latency degradation — before anything comparable happens.

Figure 13 places the two onset points side by side, and should be read with that difference in kind firmly in mind. We did not push Dynamo to its collapse at the 8192-token cap; the ramp stops it, healthy, at the utilization gate. The only evidence for what Dynamo does past that gate is a provisional probe run at the older 1024-token cap, in which decode did not collapse even at 500 users: client TTFT rose to a bounded ~21 second to 22 second plateau via admission queueing while per-token speed stayed roughly flat. That number predates the 8192-cap hardening and should be treated as indicative, not confirmed, but it is the basis for describing Dynamo's degradation as bounded queueing rather than a cliff. Note also that ~21 second TTFT is itself far outside any interactive SLO: the value of bounded degradation is predictability, not usable extra capacity.

Figure 13: Governing Capacity: A 20× Gap between the Two Shapes

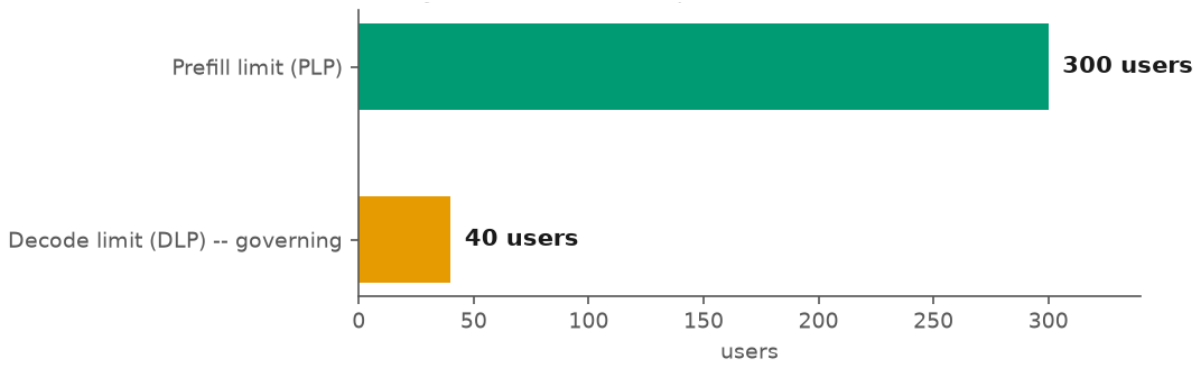


NOTE: Not the same kind of limit: Dynamo's 40 is a proactive compute gate; Mono-vLLM's 805 is the point of a genuine, abrupt collapse. Bigger is not simply better here.

7.2.2 Why the Two Failure Modes Differ

Both shapes run the same vLLM engine, so the asymmetry comes from how each pools its GPUs. In Mono-vLLM, prefill and decode share one scheduler and one KV-cache pool across all 8 GPUs. As concurrency climbs, a rising tide of concurrent long-context prefills contends with in-flight decode for the same memory. Once KV pressure tips the pool into heavy preemption, the whole replica cascades: the classic queueing cliff. In Dynamo, decode has its own dedicated 4-GPU pool that admits work independently of prefill. When it is full, it simply queues admissions rather than letting prefill steal its cycles, and that is what bounds the degradation. The flip side, and the structural reason Dynamo's decode gate lands so low, is that its fixed 1-prefill/1-decode split gives decode only 4 of the 8 GPUs. Mono-vLLM's colocated replicas, by contrast, can bring the full 8-GPU budget to bear on decode when prefill is momentarily light. Dynamo's prefill limit (PLP = 300 users, gated on NIXL handoff latency exceeding `max(1 s, 2× baseline)`; measured on this same hardened 8192-cap run) sits 7.5× above its decode limit, as shown in Figure 14, so this split strands prefill silicon while decode starves. Decode is the true bottleneck for this model under disaggregation, and on this hardware the shape cannot rebalance to fix it. Mono-vLLM has no separate prefill/decode capacity to report: its unified OSP conflates both roles on the shared pool.

Figure 14: Within Dynamo, Decode, Not Prefill, Is the Real Bottleneck (7.5× Lower)



7.3 Soak: Sustained Mixed-Traffic Stability, and Why the Targets Differ

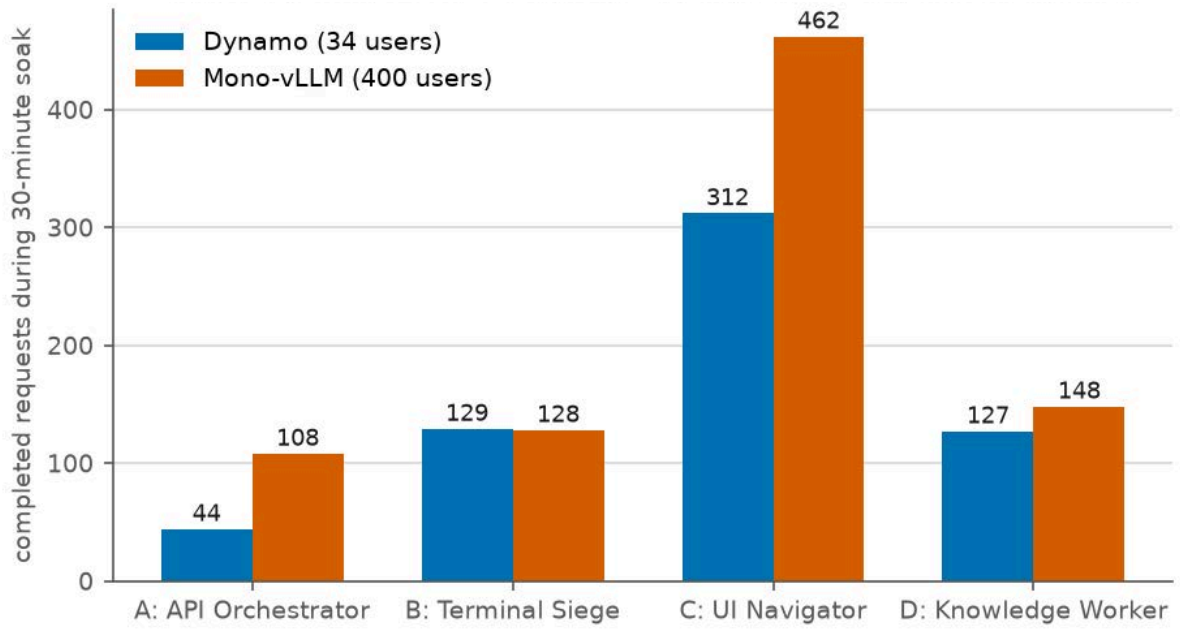
Shape	Soak Target	Fraction of Governing Point	Duration	Requests (A / B / C / D)	Exception Rate
Dynamo (disaggregation)	34 users	85% of 40 (standard 0.85 × min(PLP, DLP))	1800s, full duration	44 / 129 / 312 / 127	0.0%
Mono-vLLM	400 users	~50% of 805 (revised down; see below)	1800s, full duration	108 / 128 / 462 / 148	0.0%

Both shapes sustained their full 30-minute soak with zero exceptions, serving the complete four-profile mix throughout rather than starving any one class of user, as shown in Figure 15. The soak targets are legitimately different fractions of each shape's governing point, for a documented harness-behavior reason, not a threshold moved for convenience: at 85% of OSP (684 users), Mono-vLLM's soak showed profile D's heaviest requests (32.7k-token contexts + near-8192-token generation) taking so long under a synchronized cold-start burst that the harness's own zero-completions check misfired even after we widened it. That burst meant all soak users spawned within ~40 seconds, unlike the ramp's gradual step-by-step buildup. 400 users is the concurrency the ramp itself showed to be clear of that effect: full, healthy 8192-token completions, no synchronized-burst pathology. We disclose this explicitly rather than normalizing it away, because it bounds what the soak demonstrates.

One thing to read carefully in that table: Mono-vLLM carries roughly 12× Dynamo's user count but completes only about 1.4× as many requests in the same 30 minutes (846 vs. 612). That is expected rather than anomalous. Completions in a fixed window are governed by per-request service time, and at 400 concurrent users each request (long profile D contexts against an 8192-token generation cap) takes far longer to finish than it does at 34. Concurrency and completed-request throughput are simply not proportional at these context and generation lengths, which is the same point made in Chapter 2.1 about tokens per second: a bigger user count is not by itself more delivered work.

One honest limit on what the soak demonstrates: both targets sit well below each shape's governing point (Dynamo's 34 is 85% of a proactive 40-user gate; Mono-vLLM's 400 is ~half its 805 collapse point), so 0.0% exceptions is evidence of steady-state stability at a sustainable operating load, not of resilience right at the edge of capacity. We were unable to soak Mono-vLLM near its governing point because of the synchronized-cold-start harness artifact above, a limitation we record in Chapter 11.7, not a property of the architecture.

Figure 15: Soak: Both Shapes Serve the Full Mixed-Profile Load, with 0% Exceptions



7.4 NIXL Transfer-Fabric Cost

Shape	Peak Handoff Latency (Soak)	TP-Spread (Decode)
Dynamo (disaggregation)	0.328 seconds	0.0% single decode worker; no cross-worker imbalance possible
Mono-vLLM	N/A: no NIXL transfer to measure structural zero, not a missing value	0.57% small residual imbalance between the two colocated replicas

Read the 0.328 seconds as an order-of-magnitude cost rather than a precise one-way transfer time. We did verify it is not a clamped artifact: the raw Prometheus histogram's le=5.0 bucket count exactly equals the `+Inf` count for Dynamo's handoff-latency metric: zero transfers exceeded the 5 second ceiling, so this is genuine transfer-fabric cost, not a masked tail; the measurement artifact this check guards against is described in Chapter 11.4.

Chapter 8: Discussion

8.1 Reading the Result the Way an Operator Should

The question we set in Chapter 2 was not *Which shape emits the most tokens?* but *Which sustains the most concurrency within the experience SLO, and how does it fail past that point?* Chapter 7.1 and Chapter 7.2 answer both questions, and the answer is not the clean *monolith wins low concurrency, disaggregation wins high concurrency* story a reader might expect. Two things are true at once. On the primary SLO metric, disaggregation is the better citizen: its ceiling violation is proportionally smaller (2.74× vs. 6.36× over) and confined to one profile rather than three, and it wins that despite three configuration asymmetries that favor the monolith, as shown in Chapter 5.1. On raw failure-onset concurrency, the monolith goes far further before it breaks: its collapse is up near 705 to 805 users, while disaggregation's ramp is stopped, still healthy, by a proactive utilization gate at 40. But as Chapter 7.2 stresses, those two numbers are different kinds of limit (see the caution there against reading them as a capacity ratio). An operator should therefore weigh two things, not one: how much concurrency they actually need, and how they want the system to behave once they exceed it. The choice is between a bounded, predictable admission-queueing degradation (disaggregation's decode role) and a simpler stack that runs fine across a wide range and then hits a genuine, abrupt cliff (the monolith). All of this rests on a single measured run of each shape (see Chapter 11.7), so we frame it as a well-characterized data point, not a law.

8.2 The Cost of Disaggregation Is Not Just Latency

Disaggregation's price is operational as much as numeric: NIXL priming before every measurement, a control plane and transfer fabric to keep healthy, per-topology telemetry, and a larger failure surface. This study itself hit and had to fix five real disaggregation-specific defects to get Nemotron running at all (see Chapter 6.9), none of which the monolith needed. An honest decision framework weighs these against the QoS margin and failure-mode headroom disaggregation buys, not just a latency delta. This is the enterprise-economics point: the cost lives around the GPU.

8.3 Tokens Per Second Is a Weak Verdict; QoS Capacity and Failure Mode Are the Useful Ones

We report throughput-adjacent metrics (NIXL handoff cost, TP-spread, exception rate), but we decline to crown a winner with a single composite number for this pairing, for the reasons given in Chapter 5.4. The per-profile QoS-lock margin and the governing-point failure mode are what an operator is actually buying, and we report those directly instead. The discipline of not reducing a two-different-limits comparison to one number is itself part of the methodological contribution: a scalar score would have hidden precisely the asymmetry that makes this result useful.

8.4 On the Ingress Tax and What We Didn't Measure Cleanly This Time

Earlier work in this harness (see Chapter 5.5) established the client/server latency split (the ingress tax) as a first-class metric, and found it real but small on ITL for other topologies. For this dataset we did not isolate a clean, defensible ingress-tax split: the two candidate numbers we have (client-observed `fe_ttft_p99` versus server-side `prefill_vllm_p99`) differ by a real margin at moderate concurrency, but for Dynamo that gap is entangled with NIXL handoff time sitting in the same path, so attributing it cleanly to ingress overhead specifically would overstate our certainty. We report the client-observed figures throughout Chapter 7 (the latency a real user actually experiences, which is the methodologically correct default per Chapter 5.5) and flag the clean server-versus-client decomposition as unfinished for this dataset rather than publish a number we can't fully stand behind.

8.5 Relation to Prior Findings

Our data speaks directly to three claims in the recent literature, confirming one, complicating a second, and giving a production instance of a third.

8.5.1 KV-Transfer Bandwidth Is Not the Bottleneck

This claim is confirmed. The Beyond the Buzz[4] paper argues from simulation that provisioned data center bandwidth is generally ample and KV transfer is not the binding constraint. We find the same empirically: decode's governing gate is GPU-utilization saturation (100% util; we label it utilization-bound rather than asserting compute-vs-memory-bandwidth, since 100% util alone does not distinguish the two; this is consistent with the primer's point that decode is fundamentally memory-bandwidth-bound, see Chapter 3.2), and prefill's own limit sits 7.5× higher (PLP = 300 versus DLP = 40). The transfer fabric is a real, measured cost (0.328 second peak handoff, see Chapter 7.4) and it is even what gates prefill, but it binds only at a concurrency decode never reaches, so it is never the overall bottleneck.

8.5.2 Disaggregation Wins for Prefill-Heavy Traffic (Input >> Output Length, ISL >> OSL)

This claim is complicated by our result. Our workload is prefill-heavy by that measure (profile D averages ~32.7k input tokens against an 8192-token output cap; see Chapter 5.2), the regime the Beyond the Buzz paper says should favor disaggregation. Yet decode, not prefill, saturates first, and disaggregation's governing gate lands lower than the monolith's, not higher. The reconciliation is exactly that paper's own caveat: disaggregation's payoff "depends on dynamic rate-matching[4]" of prefill to decode capacity, and our single hardware-forced 1P + 1D split is the opposite of rate-matched, stranding prefill silicon while decode starves (see Chapter 7.2). So we do not contradict the ISL >> OSL claim; we show what happens when the rate-matching precondition cannot be met, which on a single 8-GPU node at this model's TP = 4 floor, it cannot.

8.5.3 The Monolithic Collapse is a Production Instance of a Simulated Knee

The Price-of-Anarchy analysis locates a saturation knee around concurrency ~128 on Dynamo/B200 in simulation, beyond which coordination degrades. Our production monolith's abrupt collapse (705 to 730 users) sits well above that figure, and disaggregation's proactive gate (40) sits below it, so on this stack we never drive the disaggregated shape into the routing-degradation regime that analysis characterizes, while the monolith's cliff is a concrete, measured cousin of the knee it predicts.

Where we differ from all of this in kind is method and currency: an empirical, traffic-realistic, production-platform comparison scored on QoS-margin and failure mode rather than simulated throughput.

4. Tiyasa Mitra, et al., “Beyond the Buzz: A Pragmatic Take on Inference Disaggregation.” arXiv, https://arxiv.org/abs/2506.05508.

Chapter 9: Decision Framework

9.1 The One-Page Operator Guide

If your situation is…	Choose	Because
Need the smallest SLO-violation margin on a tight per-profile ITL/TTFT ceiling, at low concurrency	Dynamo	Smaller QoS-violation margin (2.74× vs. 6.36× over ceiling, Section 7.1), isolated to one profile instead of three, even with the monolith's configuration advantages (see Chapter 5.1).
Need to sustain many hundreds of concurrent users, can tolerate a broader (not tighter) SLO miss, and want the simpler stack (no NIXL, no transfer fabric, no priming)	Mono-vLLM	Far higher failure-onset concurrency (collapse near 705 to 805 users vs. Dynamo's 40-user gate), *provided you stay clear of the cliff*; its failure is abrupt, so leave margin (see Chapter 7.2).
Traffic is decode-heavy (long generations, profile-D-like) and you would otherwise run the monolith near its cliff	Dynamo, or keep Mono-vLLM well clear of ~700 users	Mono-vLLM's collapse is abrupt, not gradual. Dynamo instead degrades *predictably* (bounded admission-queueing) rather than cliffing. See the caveat on predictable ≠ usable in rule 2.
Cost-sensitive, low-to-moderate concurrency, no tight interactive SLO	Mono-vLLM	Fewer moving parts, no transfer-fabric operational cost (see Chapter 8.2), ample headroom below the cliff.

9.2 Decision Rules

1. Know your real concurrency need first, but read the two capacity numbers correctly. Dynamo's governing point (40 users) is where a proactive utilization gate stops it while it is still serving well; the monolith's (805) is where it actually collapses. They are not a like-for-like 20× capacity ratio (see Chapter 7.2). What you can say cleanly: if your steady-state need is comfortably in the tens of users, either shape can serve it and the choice is about SLO margin and operational cost; if it is in the many hundreds, only the monolith was measured to reach there (below its cliff).
2. Predictable degradation is not extra usable capacity. Past its gate, Dynamo queues admissions to a bounded latency rather than collapsing. But that bounded latency (a provisional ~21 second TTFT in our probe, see Chapter 7.2) is itself far outside any interactive SLO. The value of the bounded mode is predictability and no cascade, not headroom you can actually serve users in. Do not size a deployment into it.
3. Budget SLOs on the client-observed (ingress-inclusive) latency, not intrinsic server-side numbers; that's the number your users actually experience (see Chapter 5.5). We did not cleanly isolate the ingress-tax component for this dataset (see Chapter 8.4); use the client-observed figures in Chapter 7 directly rather than trying to back one out.
4. On this hardware, a decode-bottlenecked workload strands prefill silicon. Dynamo's fixed 1P + 1D split gives decode only 4 of the 8 GPUs while prefill (limit 300) is massively over-provisioned relative to decode (limit 40), so if your workload is decode-heavy and you cannot add GPUs or rebalance the split, the monolith's flexible 8-GPU pool may simply have more decode silicon to give (see Chapter 7.2). Weigh Dynamo's smaller SLO-margin against that.
5. This rule set is Nemotron-on-one-8-GPU-node, single-run specific. A different model, more GPUs, or a multi-node deployment could open a real prefill:decode rebalancing choice this hardware forecloses (see Chapter 5.1 and Chapter 8.5), and every number here is one measurement per shape (see Chapter 11.7). Re-derive this table for your own model and hardware rather than porting it.

9.3 What This Does NOT Say

It does not say disaggregation is faster, rank by tokens per second, or claim either shape clears the strict per-profile SLO ceilings outright; neither does, at any concurrency we tested (see Chapter 7.1). It says: for your concurrency need and tolerance for how a system fails once pushed past what it can sustain, here is which of these two shapes fits, and why. That is the enterprise-economics question, answered in the currency of sustainable, boundedly-failing capacity rather than a spec-sheet number.

Chapter 10: Related Work

Disaggregated LLM inference, separating the compute-bound prefill phase from the memory-bandwidth-bound decode phase onto independently provisioned worker pools, has moved quickly from proposal to production substrate. We group prior work into four strands and position our contribution against each.

10.1 Foundational Disaggregation

DistServe (Zhong et al., OSDI 2024) established the architectural case, reporting up to 7.4× more requests (or 12.6 × tighter SLO) than colocated baselines by assigning prefill and decode to separate GPUs and tuning each phase independently. Splitwise (Patel et al.) framed the same split around the compute-bound versus memory-bound distinction and phase-specific hardware provisioning, reporting substantial throughput gains at lower cost. These works motivate why to disaggregate. They do not answer, for a given operator and a realistic traffic mix, whether it pays off versus a well-tuned monolith, which is the question we take up.

10.2 Systematic Design-Space and Boundary Analyses

The most direct antecedent to our thesis is NVIDIA Research's Beyond the Buzz: A Pragmatic Take on Inference Disaggregation, which simulates hundreds of thousands of design points and concludes disaggregation is conditional. It wins for prefill-heavy traffic (ISL >> OSL) and models larger than ~10B parameters, and depends on dynamic rate-matching of prefill:decode capacity. Notably for our methodology, it derives a closed-form KV-cache-transfer bandwidth model and argues that provisioned data center bandwidth is generally sufficient, KV transfer is not the bottleneck.

The Price of Anarchy in Disaggregated Inference formalizes disaggregation as three coupled games using NVIDIA Dynamo as its case study (Dynamo v0.9.0 on B200), and empirically locates a saturation knee (~concurrency 128) beyond which selfish, cache-affinity routing degrades. This is a useful lens for the point where coordination overhead erodes the disaggregation benefit.

How Far Can Disaggregation Go? pushes to operator-level attention-FFN disaggregation and finds that finer granularity helps latency/interactivity but not peak throughput, a caution against assuming more disaggregation is monotonically better. Our work is complementary and distinct: rather than simulation or synthetic microbenchmarks, we run an empirical, traffic-realistic monolithic versus disaggregated comparison on production infrastructure (VCF + VKS + Run:ai + Dynamo), for a capable ~120B hybrid-architecture model, and reduce it to an operator decision framework.

Our own data speaks directly to two of these prior claims. First, consistent with the Beyond the Buzz paper, we find KV-transfer bandwidth is not the binding constraint (Dynamo's prefill/NIXL limit sits ~7.5× higher than its decode-compute limit; see Chapter 7.2). Second, the abrupt monolithic collapse we observe past ~705 users (see Chapter 7.2) is a concrete, production-stack instance of the kind of saturation knee the Price-of-Anarchy analysis locates in simulation.

10.3 Adaptive Scheduling and P/D Ratio Control

DOPD treats prefill/decode as a producer-consumer imbalance and dynamically retunes the instance ratio, reporting P90 TTFT –67.5% and P90 TPOT –22.8% with goodput up to 1.5× versus vLLM and DistServe. Arrow elastically reassigns stateless instances between prefill and decode roles from real-time metrics, reporting 5.62× (versus vLLM) and 7.78× (versus vLLM-disaggregated) higher sustainable request rate on bursty Azure Code traffic. These reinforce our decision-framework point that the right prefill:decode ratio is workload-dependent. On a single 8× H100 node at this model's

TP = 4 memory floor, though, only one disaggregated ratio (1P + 1D) physically fits, which bounds what our own experiment can sample and is itself a finding operators on similar hardware should note (see Chapter 5.1). DriftSched is adjacent rather than P/D-specific (multi-tenant GPU QoS scheduling, published as a preprint), but its EMA-based runtime drift calibration parallels our per-phase baseline recalibration, and we cite it as prior art for adaptive, drift-aware QoS gating.

10.4 Traffic Realism and the Coordination Fabric

Our arrival models rest on Paxson & Floyd's foundational result that wide-area traffic is not Poisson (IEEE/ACM ToN 3(3), 1995): session arrivals may be Poisson, but bursts are self-similar ($0.5 < H < 1.0$), and aggregating heavy-tailed ON/OFF sources is the mechanism that produces this burstiness. This directly grounds our profile A (2-state ON/OFF renewal burst generator) and profile B (high-CV Gamma, CV = 1.5) rather than exponential inter-arrivals. On the transfer fabric, NIXL (ai-dynamo) provides a four-layer abstraction over UCX (RDMA/GPU-Direct), Libfabric (EFA), GDS, and POSIX back ends. AWS's NIXL-with-EFA integration (2026) confirms it as a cross-framework standard (Dynamo, SGLang, vLLM) for "high-throughput, low-latency KV-cache transfer between prefill and decode nodes," establishing external validity for the NIXL-dependent topologies we test.

10.5 Lineage and Positioning

This paper is the third in a series on realistic LLM workload simulation, following SPOC (developers using coding assistants) and Envoy (a mixed human + autonomous-agent workload). Both prior papers used monolithic vLLM serving; here we reuse and evolve the same user-profile methodology to ask the disaggregation question directly. To our knowledge, this is the first study to combine (i) traffic-realistic, multi-profile load derived from large real-world corpora, (ii) an empirical monolithic-vs-disaggregated comparison for a capable ~120B Mamba/Attention-hybrid MoE on (iii) production VCF + VKS + Run:ai + Dynamo infrastructure, distilled into (iv) an actionable operator decision framework, including an explicit account of the ingress/measurement overhead real users incur and the concrete disaggregation-enablement fixes this model required (see Chapter 6.9).

Chapter 11: Limitations and Threats to Validity

11.1 NIXL Handoff Latency: Measured, with a Minor Caveat

The runtime does export the NIXL KV-transfer time (`vllm:nixl_xfer_time_seconds`) and post time on the disaggregated workers, so we report the aggregate p99 transfer time as the handoff latency for Dynamo (0 for Mono-vLLM). Caveats are as follows:

1. The metric is emitted by both the sending (prefill) and receiving (decode) workers, so our aggregate p99 reflects the transfer from both perspectives rather than a single one-way number
2. The dedicated NIXL exporter ports (9501+) remain unbound on this runtime, so we source the transfer time from the vllm: series rather than a NIXL-native endpoint.

Neither affects the comparison, and our headline metric, per-profile SLO attainment (see Chapter 5.3), captures any transfer cost end-to-end regardless.

11.2 The Ingress Tax is Testbed Influenced

Client-side latency includes the ingress hop, which we report as a real UX factor and use as the default throughout Chapter 7, because it is what users actually experience. The magnitude of that tax, however, partly reflects our specific testbed network position, so a differently located client would see a different absolute tax. Both shapes were measured from the same client through the same ingress, so the comparison between them is not distorted by client placement even though the absolute latencies are testbed-specific. As Chapter 8.4 records, we did not achieve a clean server-versus-client decomposition for this dataset.

11.3 Arrival-Model Grounding

Profile A is implemented as a two-state ON/OFF renewal burst generator (lognormal idle plus discrete fan-out), not a Markov-Modulated Poisson Process, and we describe it accurately as such rather than reaching for the more familiar label. The bursty, correlated arrival models for profile A and profile B are grounded in Paxson & Floyd's established heavy-tailed ON/OFF result (see Chapter 5.2 and Chapter 10). Profile C and profile D use exponential inter-arrivals, which remains defensible for independent human and agent actors but is a simplification: real users are not perfectly independent.

11.4 Model Roster and Single Node

The quantitative comparison (see Chapter 7) covers one model, Nemotron-3-Super-120B-A12B-FP8, on a single 8× H100 node. Earlier tool development on gpt-oss-20b/120b is not carried forward as a data point, for the reasons in Chapter 5.1.1. A single model on a single node means we cannot yet say how disaggregation's benefit scales with model size or across architectures beyond this one Mamba/Attention-hybrid MoE. Recent work suggests the benefit grows with model size (>10B active-relevant params) and prefill-heavy traffic (see Chapter 10), which motivates testing additional models as future work rather than treating this result as universal. Multi-node disaggregation is also future work. TP shapes are fixed per topology (no dynamic prefill:decode rebalancing), which brackets, but does not reproduce, the elastic schedulers surveyed in Chapter 10.

We attempted a fourth model (Gemma-4-31B-IT) and excluded it from the quantitative comparison entirely after it proved incompatible with our Dynamo/vLLM version. Across two disaggregated shapes and repeated NIXL re-tuning, it showed a confirmed KV-transfer tail-latency defect: true average transfer time substantially exceeded what an initial, clamped p99 histogram reading had suggested. We root-caused the clamp itself as a measurement artifact, not the underlying transfer behavior, so the defect is real. It also showed repeated crash instability that did not resolve with configuration changes. We report this as a compatibility finding, not a performance data point: the numbers we observed were produced mid-failure and are not a fair characterization of the architecture. No further numbers from this model appear anywhere in Chapter 7.

11.5 Dataset and Session Cardinality

We use the full multi-corpus prompt set to keep session cardinality high and avoid prefix-cache masking, but the load generator caps sessions held in memory per worker; effective runtime diversity is bounded by that cap × worker count. We use the full corpus (never a truncated subset) specifically so the sampled sessions are drawn from across the whole corpus rather than a clustered prefix, but we do not claim to exercise every unique session simultaneously.

11.6 The Comparison is Best-Realistic-Config, Not Flag-Identical, and the Asymmetries Favor the monolith

This is the most important threat to validity to hold in mind, as first stated in Chapter 5.1. We deployed each shape in the strongest configuration actually available to it, not a lowest-common-denominator matched control. Three specific asymmetries all run in the monolith's favor, and all sit on the decode axis that decides the result. Mono-vLLM runs prefix caching and MTP speculative decoding, both of which accelerate decode; Dynamo must disable both because they are currently incompatible with this Mamba/Attention-hybrid model across a NIXL boundary (Chapter 6.9, fix 4 and fix 5). Mono-vLLM is also graded against a looser Profile-D ITL ceiling (115 ms versus 50 ms; see Chapter 5.3). We chose not to neutralize these, on the view that losing prefix caching and speculative decoding is a genuine present-day cost of disaggregating this model class, not an artifact to tune away. A monolith operator really does get those features, and a Dynamo operator really does not, today. But it means two things for interpretation:

1. The head-to-head is between realistic deployments, not identical engines
2. Because every asymmetry favors the monolith, disaggregation's win on SLO-violation margin (see Chapter 7.1) is if anything understated, while the monolith's raw-concurrency headroom (see Chapter 7.2) is somewhat flattered.

A future flag-matched run (monolith with prefix-caching/MTP disabled) would isolate the architectural effect from the feature effect; we did not perform one.

11.7 Single Run Per Shape, and Soaks below the Governing Point

Every number in Chapter 7 is a single measurement of each shape (one ramp and one soak, no repeated rounds), so we report point values without variance bounds. Reaching a clean run required several harness fixes discovered along the way (see Chapter 6.9); a fresh multi-round campaign on the finalized harness is the right way to add error bars and is listed as future work (see Chapter 12.2). Relatedly, neither soak ran near its shape's governing point: Dynamo soaked at 85% of a proactive 40-user gate and Mono-vLLM at ~50% of its 805-user collapse point, the latter because a synchronized cold-start spawn in the load generator tripped a false zero-completions abort when we attempted a higher soak target (see Chapter 7.3). So the 0.0% soak exception rates demonstrate steady-state stability at a sustainable load, not resilience at the edge of capacity. In particular, we did not observe Mono-vLLM's sustained behavior just below its cliff.

Chapter 12: Conclusion and Future Work

12.1 Conclusion

We compared disaggregated and monolithic LLM serving for Nemotron-3-Super-120B-A12B-FP8 not by tokens per second, but by the currency that governs enterprise AI economics: sustained concurrency within an acceptable user-experience SLO. Because neither topology cleared the strict per-profile ceilings at this scale (see Chapter 7.1), we also compare by how each one actually fails when pushed past what it can sustain, since that failure mode is itself an operator-relevant property.

The honest result is conditional, not a clean win for either side. On the primary SLO metric, disaggregation (Dynamo) is the better citizen: its ceiling violation is proportionally much smaller (2.74× vs. 6.36× over, on the same metric) and confined to one profile rather than three, and it earns that despite three configuration asymmetries that all favor the monolith (see Chapter 5.1 and Chapter 11.6). Mono-vLLM reaches far higher raw concurrency before it fails, on a simpler and lower-overhead stack, but its failure is an abrupt collapse cliff (an ~22× time-to-first-token jump in a single ramp step) with no graceful buffer. The two governing concurrencies, 40 users and 805 users, are different kinds of limit and emphatically not a 20× better capacity ratio (see Chapter 7.2). Which property matters more depends on whether an operator faces a tight interactive ceiling they must never cross, or a broad mixed load they would rather run hot and simple. Section 9 gives the rule set for making that call. On a single measured run of each shape (see Chapter 11.7), we would rather report this asymmetry plainly than round it into a single winner.

12.1.1 The Platform underneath the Result

This comparison ran with NVIDIA Run:ai and NVIDIA Dynamo deployed on top of VCF and VKS, the enterprise, on-premises posture the whole study assumes: a capable model served under an organization's own governance and data control. Three platform observations shaped whether the experiment was runnable at all, and Chapter 4.3 gives them in full: across a multi-day teardown-and-redeploy cycle on eight passthrough H100s, vSphere and vSAN never lost a volume or dropped a network path; the VKS CNCF conformance meant every tool ran as upstream-documented, with no platform-specific workarounds in the harness; and Run:ai's project-scoped GPU quota is the specific reason repeated redeploys never deadlocked on GPUs. The most portable of these, independent of the mono-versus-disaggregated result, is the integration mechanic in Chapter 6.3: wiring Run:ai to plain Kubernetes deployments rather than its own `runai submit` CLI is a documented, supported path that few Dynamo-on-Kubernetes guides walk through.

The broader message continues SPOC and Envoy: capacity planning for enterprise AI must rise above the spec sheet and above the serving engine alone. The result above is a property of Nemotron on this stack, not a universal law; Chapter 12.2 outlines what it would take to know how far it generalizes.

12.2 Future Work

- Other model families and sizes: This paper's quantitative result rests on one model. Re-running the earlier gpt-oss-20b/120b exploration (see Chapter 5.1.1) on the hardened harness, at the same 8192-token generation cap, would give a genuine second model in this comparison. That is the most direct way to learn whether Nemotron's specific asymmetry (large margin difference, mono's abrupt cliff) generalizes, or whether it is specific to this Mamba/Attention-hybrid, MoE architecture.
- Round-to-round variance: Each topology here was measured once, as shown in Chapter 11.7 A second round on the same hardware would put real error bars on the numbers in Chapter 7 rather than single-run point estimates.
- Multi-node disaggregation and larger models: This node's 8 GPUs and Nemotron's TP = 4 memory floor left exactly one disaggregated shape to test (see Chapter 5.1). More GPUs or a multi-node deployment would open up the prefill:decode ratio as a real design variable instead of a hardware-forced constant.

- Elastic prefill:decode rebalancing (see Chapter 10): Re-run the framework to see how much of the static-topology gap dynamic role-switching recovers.
- Broaden profiles and corpora as agentic workloads evolve, and revisit the 8192-token generation cap (see Chapter 5.2) if real workloads are shown to need longer completions than that bound accommodates.

BROADCOM®