Building Open Source Inference Serving Systems

by

Xiangxi Mo

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph E. Gonzalez, Co-Chair
Professor Ion Stoica, Co-Chair
Professor Koushik Sen
Professor Alexey Tumanov

Spring 2026

Building Open Source Inference Serving Systems

Abstract

Building Open Source Inference Serving Systems

by

Xiangxi Mo

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph E. Gonzalez, Co-Chair

Professor Ion Stoica, Co-Chair

Inference serving deserves recognition as its own area of systems research. Just as operating systems abstract machine resources, file systems and databases abstract storage—inference frameworks abstract accelerated computing devices for production machine learning workloads. The dissertation makes three primary contributions spanning six years of open-source research in inference systems.

First, **model composition** through Clipper, InferLine, and Ray Serve established principles of SLO-aware pipeline serving—treating serving pipelines as compositional, schedulable units. InferLine achieved up to 7.6× cost reduction through optimized SLO decomposition, and Ray Serve translated these insights into production infrastructure adopted by companies and financial institutions running significant GPU clusters.

Second, **vLLM**—the dominant open-source LLM serving system—introduced PagedAttention, a virtual memory abstraction for KV caches that achieves near-zero memory waste ($<$4%) and 2–4× throughput improvements over prior systems. Under the author's role as one of the project leads (2023–2026), vLLM grew to 75K+ GitHub stars and 2,000+ contributors. The author's subsequent work extended these ideas through explorations in expanding the scope of PagedAttention to hybrid architectures and NVLink-aware memory management.

Third, **GPU kernel multiplexing** through the OoO JIT compiler and GreenContext pushed resource sharing below the model boundary toward fine-grained hardware utilization via kernel combining and SM partitioning.

Together, these contributions demonstrate that purpose-built abstractions are necessary and beneficial at every layer of the inference serving stack—from model composition to memory management to hardware resource multiplexing.

To Lili, for your unwavering support.

Contents

List of Figures

List of Tables

Acknowledgments

I have been fortunate to build this dissertation inside a community that made the work deeper, more useful, and more joyful than it could have been alone. My research and open-source work began during my undergraduate studies, when I first encountered RISELab's breadth and depth in systems research in 2017. After almost ten years at the intersection of systems and machine learning, I have many people to thank.

First and foremost, I thank Lili. We met in the first month of freshman year, and she has been my biggest supporter through every challenge, deadline, and leap of faith. Her belief made the hard parts possible, and her presence made the good parts brighter.

Lili and I raised our two cats, Momo and Juju, together. They spent countless nights beside me through experiments, papers, and overnight open-source releases, and they made the daily rhythm of intense work gentler.

My parents, Sophy and Jeff, inspired the path I could take and nurtured the brave, critical, and open mindset that made this dissertation possible. My friends Rick, Viki, Henry, Helen, and Jiahua supported me, grounded me, and kept life larger than the dissertation.

My co-chair and advisor, Professor Joseph E. Gonzalez, shaped this PhD through guidance, trust, and an intellectual home for the work. Thank you for teaching me project taste and product taste, for encouraging research at the boundary between systems and practice, and for making room for systems that had to be built, shipped, and maintained in the world before their lessons were fully understood. My co-chair, Professor Ion Stoica, offered advice on topics big and small; our weekly walk-and-talks to Asha Tea House were formative in how I approached scaling the vLLM community and ecosystem. Professor Koushik Sen guided me throughout my course of study with generous time, feedback, and perspective. Professor Alexey Tumanov and I share a history of optimizing inference systems since Clipper and InferLine, and I appreciate the full-circle support throughout the years. My advisors and committee sharpened both the technical claims and the broader argument that inference serving is now a systems discipline.

The Berkeley research community, especially RISELab and the Sky Computing Lab, created the environment in which this work grew. Its culture of building ambitious open-source systems, arguing from measurements, and caring about users shaped every chapter of this dissertation. The broader Berkeley staff and administrative community made the day-to-day machinery of graduate school work.

This dissertation was built through collaborations across many projects and many years. Thank you to Dan Crankshaw, Eyal Sela, Corey Zumar, Paras Jain, Ajay Jain, Hari Subbaraj, Rehan Durrani, Ryan Cheng, Ian Fang, Devin Petersohn, William Ma, Doris Lee, Stephen Macke, Doris Xin, Ionel Gog, Justin Wong, Kumar Agrawal, Sukrit Kalra, Peter Schafhalter, Eric Leong, Xin Wang, Bharathan Balaji, Sarah Wooders, Amit Narang, Kevin Lin, Shu Liu, Vincent Liu, Asim Biswal, Audrey Cheng, Shiyi Cao, Lily Liu, Cade Daniel, Langxiang Hu, Woosuk Kwon, Zhuohan Li, Zhijie Deng, Jongseok Park, Chen Zhang, Kuntai Du, Kaichao You, Yi Xu, Ziming Mao, Moshik Hershcovitch, Henric Zhang, Guy Girmonsky, Gil Vernik, Michael Factor, Tiemo Bang, Soujanya Pon-

napalli, Danny Harnik, Amog Kamsetty, Luis Gaspar Schroeder, Liana Patel, Dacheng Li, Tyler Griggs, Eric Tang, Sumanth Hegde, Kourosh Hakhamaneshi, Shishir Patil, and Roger Wang. There are surely others I have missed, and I apologize in advance. Your ideas, code, experiments, debates, deadlines, and trust turned sketches into systems.

The vLLM team and the open-source community around it taught me that open source is not only a way to distribute software; it is a way to discover whether an abstraction actually helps people. Every issue, pull request, benchmark, integration, deployment report, and late-night release carried the project forward. The contributors, users, hardware partners, and maintainers who treated vLLM as shared infrastructure helped it grow beyond any one lab or company.

Anyscale, Character AI, and Inferact showed me what production systems demand after the research prototype works. I was fortunate that my research career overlapped with teams that cared about translating ideas into dependable systems for real users.

Finally, thank you to everyone who helped turn this dissertation from a collection of projects into a coherent argument. The central lesson of the work is that good systems expose the right structure. The same was true of the PhD itself: the structure that mattered most was the community around it.

Chapter 1

Introduction

1.1 The Rise of Inference as a Systems Problem

Systems Abstractions and the Hardware They Tame

The history of systems research is a history of abstraction. Each major advance in computing hardware has demanded new systems layers that hide complexity, manage resources, and expose useful interfaces to higher-level software. Operating systems abstract the raw machine—CPU cycles, physical memory, peripheral devices—presenting processes and virtual memory to applications. Dijkstra's layered approach to the THE multiprogramming system [1] and the UNIX philosophy of uniform interfaces [2] demonstrated that well-chosen abstractions could tame the complexity of concurrent execution and diverse I/O devices. Virtual memory [3] freed programmers from physical address management, establishing that software indirection could provide the illusion of resources that didn't physically exist.

File systems and databases abstract storage media—spinning disks, flash drives, networked storage—presenting files, tables, and queries. Codd's relational model [4] separated logical data access from physical storage structures, while System R [5] demonstrated that declarative query languages could hide index structures, buffer management, and concurrency control from application developers. Cost-based query optimization [6] enabled systems to automatically select efficient execution strategies, and transaction semantics [7] provided guarantees that freed applications from managing locks or recovery.

Network stacks abstract physical communication channels—copper wires, fiber optics, wireless signals—presenting sockets and reliable byte streams. The TCP/IP protocol suite [8, 9] hid network heterogeneity behind reliable byte streams, and the end-to-end argument [10] provided principled guidance for abstraction placement. The BSD socket interface [11] unified file I/O, device I/O, and network communication under a common abstraction that became standard across all operating systems. File systems like FFS [12] and LFS [13] showed how storage abstractions could exploit device characteristics (locality, sequential I/O) transparently, achieving $10\times$ throughput improvements without

changing the programming interface.

Each of these abstractions arose in response to hardware that was too complex for applications to manage directly. And each became the foundation for decades of systems research, spawning its own vocabulary, its own optimization techniques, and its own research community.

We are now witnessing the emergence of another such moment. The proliferation of accelerated computing devices—GPUs, TPUs, custom AI accelerators—has introduced hardware complexity that demands new systems abstractions. Training frameworks and inference frameworks have emerged as the systems layer that abstracts these accelerators, presenting high-level operations (matrix multiplications, attention computations, activation functions) to machine learning practitioners who need not understand the intricacies of CUDA kernels, memory hierarchies, or tensor core scheduling.

This parallel is not accidental. Just as database systems required purpose-built abstractions that generic operating systems could not provide—buffer pools, query optimizers, transaction managers [14]—inference serving requires purpose-built abstractions that generic frameworks cannot provide. The structure of inference workloads is exploitable in ways that demand specialized treatment.

What makes inference systems special, and deserving of recognition as their own area of systems research, is the unique combination of challenges they face: the structure of ML workloads (predictable computation graphs, statistical arrival patterns, cacheable intermediate state), the characteristics of accelerated hardware (massive parallelism, high fixed costs, memory bandwidth constraints), and the production requirements of serving (latency SLOs, throughput targets, cost efficiency). No single prior systems discipline adequately addresses this combination of challenges and requirements.

The Economic Argument for Inference

The economics of machine learning tell a simple but profound story: models are trained once but served millions—sometimes billions—of times. A language model that costs $100 million to train may serve 100 million queries per day for years. The cumulative inference cost dwarfs the training investment within months.

This asymmetry is well-documented. Patterson et al. [15] found that inference accounted for roughly 60% of Google's ML energy use, with the split consistently at 3/5 inference and 2/5 training across 2019–2021. Industry analyses suggest inference consumes 80–90% of AI computing power in production-scale systems [16], because every prompt generates tokens that incur computational costs. The Stanford AI Index [17] reports that inference costs dropped 280-fold between November 2022 and October 2024—yet this dramatic cost reduction has only increased demand, making efficient serving more critical than ever.

This asymmetry has reshaped how organizations think about ML infrastructure. At frontier-model scale, reported OpenAI Azure inference spending reached billions of dollars annually [18], making serving infrastructure a first-order engineering and business

concern. Facebook's ML infrastructure serves "tens of millions of inferences per second" across their data centers [19], with personalized feed/ranking, visual content understanding, and natural language processing workloads characterized by embedding layers with model sizes exceeding 10GB and low arithmetic intensities [20].

Yet until recently, the systems research community had not caught up to this economic reality. Training systems received attention; serving was an afterthought. Sculley et al. [21] observed that "machine learning offers a fantastically powerful toolkit for building useful complex prediction systems quickly," but warned that "these quick wins are not free"—real-world ML systems incur "massive ongoing maintenance costs" from boundary erosion, entanglement, hidden feedback loops, and configuration complexity. The gap between model development and production deployment remains significant: Gartner [22] found that only 47% of AI projects reach production, with scaling operations and proving ROI as the primary blockers.

For decades, the research community focused almost exclusively on training—developing new architectures, scaling to larger datasets, and pushing the boundaries of model capabilities. Deployment was a task relegated to production engineers who would "figure it out" once the model was ready. This dissertation argues that this perspective is no longer tenable.

Three Eras of Inference Systems

The evolution of inference serving can be understood through three distinct eras, each defined by the dominant model architectures and the systems challenges they introduced. Chapter 2 surveys this evolution in detail (Section 2.2); here we summarize the trajectory.

In the **classical ML era** (pre-2015), inference was not a systems problem. Models were small, CPU-efficient, and deployable as stateless microservices—a Flask endpoint wrapping a scikit-learn model sufficed:

Listing 1.1: A minimal Flask-based prediction server representative of Era 1 deployments.

```
@app.route('/predict')
def predict():
    features = extract_features(request.json)
    return model.predict(features)
```

The **deep learning era** (2015–2022) changed this. GPU acceleration introduced high fixed costs per invocation, making batching critical [23]. Purpose-built model servers emerged—TensorFlow Serving [24], Clipper [25], Nexus [26], Clockwork [27]—each exploiting the predictable structure of DNN inference to achieve order-of-magnitude improvements over generic approaches. Model composition added orchestration complexity: multi-stage pipelines with end-to-end SLO constraints demanded systems like InferLine [28] that could reason across pipeline stages.

The **LLM era** (2022–present) represents a discontinuity. LLMs are memory-bound rather than compute-bound; their KV caches grow unpredictably and dominate GPU memory; autoregressive generation creates sequential dependencies that break prior batching assumptions. Orca [29] introduced continuous batching for 10–20× throughput gains, and vLLM [30] addressed the remaining memory management bottleneck through PagedAttention. Complementary advances—FlashAttention [31], speculative decoding [32], prefill-decode disaggregation [33]—further reshaped the landscape.

At each transition, general-purpose approaches proved insufficient and purpose-built abstractions yielded transformative gains. This pattern—exploitable workload structure demanding specialized systems—is the central argument of this dissertation.

The Gap Between Model Research and Systems Research

Throughout this evolution, a persistent gap has existed between the rapid progress in model capabilities and the slower development of serving systems. New model architectures appear monthly; the systems to serve them efficiently lag by years.

This gap reflects a deeper truth: model researchers and systems researchers have different priorities, different tools, and different publication venues. Model researchers optimize for capability and accuracy; systems researchers optimize for efficiency and reliability. The MLOps movement [34] emerged to bridge this gap, but challenges remain. Bridging this gap requires researchers who understand both domains—who can identify the exploitable structure in new model architectures and translate it into systems optimizations.

Dean and Barroso [35] established that "temporary high latency episodes which are unimportant in moderate size systems may come to dominate overall service performance at large scale"—the famous "tail at scale" problem. This insight applies directly to inference serving: p99 latency matters more than mean latency, and building "latency tail-tolerant" systems requires purpose-built abstractions.

This dissertation aims to narrow that gap by treating inference serving as its own area of systems research—with its own abstractions, its own optimization techniques, and its own research agenda.

1.2 Thesis Statement

This dissertation advances a central argument: **inference serving deserves recognition as its own area of systems research, distinct from training systems, distributed computing, or database systems.**

Just as operating systems abstract machine resources (compute and memory) [2, 3], file systems and databases abstract storage [4, 7], and network stacks abstract communication [8, 10]—inference frameworks abstract accelerated computing devices for ma-

chine learning workloads. This parallel is not superficial. Each of these systems disciplines emerged because a new class of hardware demanded purpose-built abstractions that generic approaches could not provide. Inference serving is no different.

The reason inference serving warrants this recognition is not merely that it is economically important—though it is, with inference accounting for roughly three-fifths of Google's ML energy use in Patterson et al.'s 2019–2021 analysis [15]—but that inference workloads exhibit exploitable structure at every layer of the stack. From model composition patterns to GPU scheduling to memory management, inference presents opportunities for purpose-built optimizations that general-purpose systems abstractions cannot capture.

The thesis rests on three interconnected claims:

Claim 1: Inference workloads have exploitable structure. Unlike arbitrary computation, inference follows predictable patterns. Models have known computational graphs. Requests arrive with statistical regularity. KV caches grow in predictable ways. Clockwork [27] demonstrated that DNN inference has deterministic, predictable execution times that can be exploited for proactive scheduling. This structure can be exploited for efficiency gains that are impossible in general-purpose systems.

Claim 2: General-purpose abstractions fail to capture this structure. Treating model serving as "just another microservice" leaves enormous performance on the table. Generic schedulers cannot exploit model-specific batching opportunities [26]. Standard memory allocators cannot handle the unique growth patterns of KV caches [30]. Kubernetes autoscaling cannot react at the granularity that inference SLOs demand [28]. Jeon et al. [36] showed that GPUs are monolithic resources where fragmentation delay accounts for ~80% of queuing time.

Claim 3: Purpose-built abstractions yield order-of-magnitude improvements. When systems are designed specifically for inference—with awareness of model structure, workload patterns, and hardware characteristics—the results are dramatic. PagedAttention achieves near-zero memory waste (<4%) compared to 60–80% waste in prior systems [30]. Iteration-level scheduling enables 10–20× throughput improvements [29]. InferLine's SLO-aware pipeline optimization achieves up to 7.6× cost reduction [28]. Nexus achieves 9–12× throughput improvement through GPU cluster scheduling [26]. Clockwork meets SLOs for 99.9999% of requests [27]. These are not incremental gains; they are transformative.

This dissertation traces six years of work contributing to inference serving foundations, through open-source systems the author helped build. The first contribution, **model composition** through Clipper, InferLine, and Ray Serve, established principles of SLO-aware pipeline serving—treating serving pipelines as compositional, schedulable units [25, 28, 37]. The second, **vLLM**, built and led the dominant LLM serving infras-

tructure around a memory-first design for the LLM era [30]. The third, **GPU kernel multiplexing** through OoO JIT and GreenContext, pushed resource sharing below the model boundary toward fine-grained hardware utilization. Together, these contributions demonstrate that purpose-built abstractions are necessary and beneficial at every layer of the inference stack.

Scope and non-goals. This dissertation is about **serving** trained models to users and applications: request handling, scheduling, memory, and accelerator utilization. It does **not** attempt to cover training at scale (checkpointing, data pipelines, distributed optimization) except where training infrastructure overlaps with inference (e.g., model artifacts and distribution). It does **not** treat security, privacy, or fairness as primary subjects—those constraints matter for deployment but deserve their own treatments. Nor does it claim that every inference workload requires custom systems: simple batch scoring on CPUs remains a valid pattern. The claim is narrower and stronger: **when** latency, cost, or concurrency push deployments toward accelerators and multi-stage pipelines, **then** the abstractions developed here matter.

Anticipating objections. One might argue that faster hardware or larger memory will eventually make serving easy enough" that generic stacks suffice. That argument underestimates how quickly new bottlenecks emerge—sometimes in autoscaling, sometimes in memory bandwidth, sometimes in compute. Co-designing model and hardware systems alone will not solve this either, given that innovation and progress are happening in all directions. Another objection holds that compilers and graph runtimes will subsume serving systems; in practice, serving must integrate **online** policies (batching, preemption, admission control) and **dynamic** memory management (KV cache) that static compilation alone does not address [30]. A third objection treats inference as just microservices plus GPUs; the evidence of this dissertation is that **workload structure**—pipelines, paged KV state, kernel-level gaps—defeats that reductionism [28, 38].

1.3 Contributions Overview

This dissertation makes three primary contributions to the field of inference serving systems, each addressing a different layer of the serving stack. Together, they demonstrate that purpose-built abstractions yield transformative improvements at every level—from model orchestration to memory management to hardware resource allocation.

Model Composition

Real-world ML deployments rarely involve a single model. Recommendation systems combine dozens of models for candidate retrieval, ranking, and filtering [20]; multi-stage

pipelines chain preprocessing, prediction, and postprocessing; ensemble methods aggregate multiple outputs for improved accuracy [39]. Serving these workloads efficiently while meeting end-to-end latency objectives requires treating pipelines—not individual models—as the unit of deployment. Yet prior systems optimized models in isolation, ignoring the compositional structure of production deployments.

This dissertation traces three systems that progressively advanced the state of model composition. Clipper [25] established the vocabulary of prediction serving—caching, batching, model selection—and demonstrated that interposing between applications and ML frameworks could improve efficiency without modifying underlying models. The author contributed to the Clipper open source project. InferLine [28] formalized SLO-aware pipeline serving, showing that end-to-end latency constraints must be decomposed into per-stage budgets that interact with batching decisions in non-trivial ways; the author is a co-author on this work. The resulting profiler-planner-controller architecture achieved up to 7.6× cost reduction and 34.5× lower SLO miss rates. Ray Serve translated these academic insights into production infrastructure; the author started and maintained the project at Anyscale, designing core abstractions—models as actors, declarative deployment specifications, composition primitives—that evolved over three years into the model serving component of the Ray ecosystem [37].

The insight underlying this work is that the structure of model composition is exploitable. Treating serving pipelines as compositional, schedulable units enables systematic optimization of the resource-SLO tradeoff that per-model approaches cannot achieve.

vLLM

Large language models broke the assumptions of prior serving systems. Their autoregressive generation creates sequential dependencies; their KV caches grow unpredictably and dominate GPU memory; static memory allocation wastes 60–80% of available capacity on typical workloads. While Orca [29] demonstrated that continuous batching could help, memory management remained the fundamental bottleneck—and Orca remained a research prototype.

vLLM [30] introduced PagedAttention to address this gap. Inspired by operating system virtual memory, PagedAttention manages KV caches in non-contiguous blocks, achieving near-zero memory waste (<4%) while enabling flexible sharing for prefix caching, beam search, and parallel sampling. Built around this abstraction, vLLM provides a complete serving system—scheduler, memory manager, API server—designed from the ground up for LLM workloads, achieving 2–4× throughput improvement over state-of-the-art systems like FasterTransformer [40]. The author led the vLLM open source project as one of the lead since open source, collaborating on technical direction, managing releases, and building the contributor community. Under this role, vLLM grew to 75K+ GitHub stars and more than 2,000 contributors, and became a PyTorch Foundation project. The author's subsequent research collaboration extended these ideas: Jenga generalizes PagedAttention for hybrid architectures mixing attention and state-

space models, while Pie exploits NVLink interconnects for unified CPU-GPU memory management.

The insight is that LLM serving requires memory-first design. PagedAttention is to LLM serving what virtual memory [3] is to operating systems: a purpose-built abstraction that enables efficient management of a scarce resource.

GPU Kernel Multiplexing

Even with sophisticated serving systems, GPU utilization remains low. Inference workloads are bursty; individual models rarely saturate all streaming multiprocessors; "bubbles" between kernel executions waste cycles. Jeon et al. [36] characterized this problem at Microsoft scale, showing that GPUs are "monolithic resources that cannot be shared at fine granularity" and that fragmentation delay accounts for ~80% of queuing time.

This dissertation presents two approaches to kernel-level multiplexing. The OoO VLIW JIT compiler, developed with Paras Jain, Ajay Jain, and collaborators, intercepts kernel launches at runtime, analyzes dependencies, and combines compatible kernels for better spatial-temporal packing—a software-only solution that demonstrated the potential of fine-grained GPU sharing. GreenContext leverages newer CUDA APIs (12.4+) for hardware-supported SM partitioning, achieving similar goals with simpler implementation, better isolation, and reduced overhead. As hardware evolves to provide better partitioning support—following patterns established by Pollux [41] for adaptive resource allocation—the software burden decreases while the benefits remain.

The insight is that exploitable structure exists below the model boundary. Fine-grained GPU sharing can improve utilization without sacrificing isolation, pointing toward a future where inference systems manage not just models but the underlying compute fabric itself.

This chapter has established inference serving as a systems discipline worthy of dedicated research, introduced the three contributions that comprise this dissertation, and provided a roadmap for the chapters ahead. Chapter 2 provides the technical background needed to understand these contributions in depth.

Chapter 2

Background & The Model Serving Landscape

2.1 Defining Model Serving

Serving as a Systems Discipline

Model serving is the systems discipline of running machine learning inference at scale. While the term may appear self-explanatory—taking trained models and using them to make predictions—its scope encompasses a rich set of systems challenges that distinguish it from both model training and conventional web service deployment. This chapter establishes the vocabulary and conceptual framework used throughout this dissertation, defining model serving as a distinct layer in the ML infrastructure stack with its own objectives, constraints, and optimization opportunities.

The distinction between training and serving is fundamental. Training seeks to minimize loss over a fixed dataset; the time to convergence matters, but individual iteration latency does not. A training run that takes three days instead of two represents a 50% slowdown, but no user is waiting for any particular iteration to complete. Serving inverts these priorities entirely. Serving seeks to minimize response latency for individual requests while maximizing throughput across all requests. Every millisecond of latency impacts user experience [35]; every wasted GPU cycle represents lost revenue [15].

This distinction has profound implications for system design. Training systems exploit data parallelism freely—larger batches generally improve hardware utilization without penalty. Serving systems face tight latency constraints that limit batch sizes; a batch size that maximizes throughput may cause unacceptable tail latency. Training systems can checkpoint and restart; serving systems must maintain continuous availability. Training systems process a known dataset; serving systems face unpredictable query distributions that shift over time. These differences explain why training-optimized frameworks—PyTorch [42], TensorFlow [43]—are necessary but insufficient for production serving. A complete serving deployment requires additional layers: request handling, batching,

scheduling, scaling, and monitoring—the concerns this dissertation addresses.

The Model Serving Stack

A complete model serving deployment comprises multiple layers, each with distinct responsibilities. Understanding this stack is essential for understanding where purpose-built optimizations have impact.

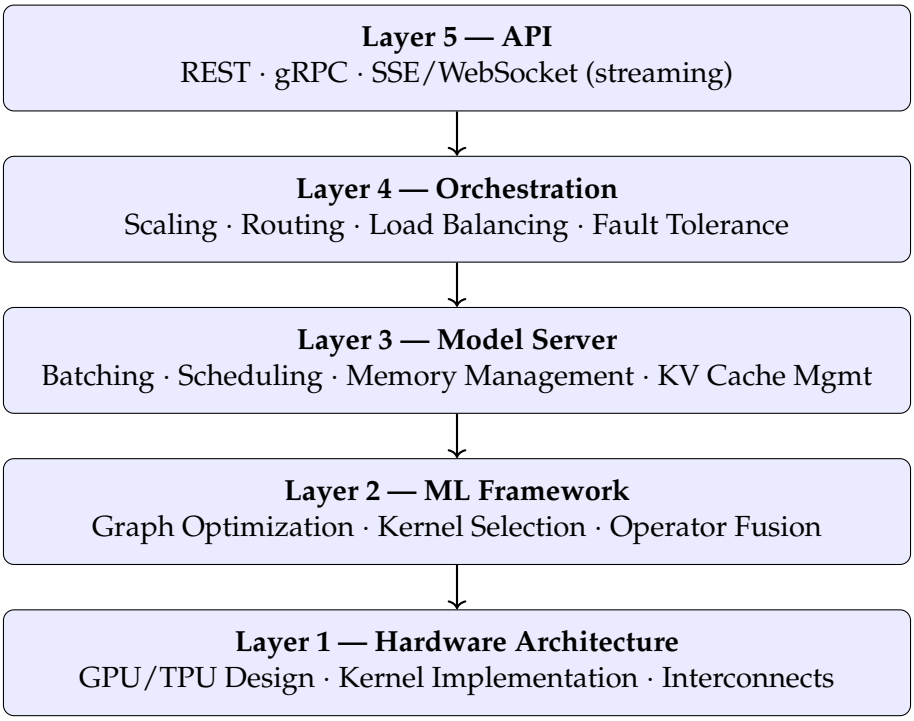

Figure 2.1: The model serving stack. Each layer depends on the layer below. This dissertation contributes to systems design across all layers of the stack.

Layer 1: Hardware Architecture. At the foundation lies the hardware on which inference executes and the kernels that drive it. GPU architecture decisions—streaming multiprocessor counts, memory hierarchy design, interconnect topology (NVLink, PCIe)—directly shape what serving strategies are viable. Kernel design translates high-level operators (attention, matrix multiply, activation) into hardware-specific instruction sequences; hand-tuned CUDA kernels, for instance, can outperform compiler-generated code by 2–5$\times$ on attention computation [31]. The shift from general-purpose GPUs to inference-optimized accelerators (NVIDIA H100 with its Transformer Engine, Google TPUv5e, AMD MI300X with its unified HBM3 pool) has made hardware-aware kernel design a first-class concern for serving systems.

Layer 2: ML Framework (PyTorch, TensorFlow, JAX). The ML framework defines how models are expressed and executed. Frameworks provide the programming model (eager vs. graph execution), automatic differentiation, and the operator library that maps neural network layers to hardware kernels. Critically for serving, frameworks perform graph-level optimizations—operator fusion, kernel selection, and precision calibration—that determine baseline inference performance. PyTorch [42] dominates research and increasingly production; TensorFlow [43] retains significant deployment footprint; JAX [44] enables functional transformations for compilation. Compiler stacks such as TensorRT [45] and XLA sit at the boundary between framework and hardware, lowering framework-level graphs into optimized kernel sequences that achieve 2–4× speedups through layer fusion and quantization.

Layer 3: Model Server (Batching, Scheduling, Memory Management). The model server orchestrates inference execution. It batches incoming requests to amortize fixed costs, schedules execution to meet latency objectives, and manages memory to maximize throughput. This layer is where serving-specific optimizations have the greatest impact, and many contributions in this dissertation center here. Systems like TensorFlow Serving [24], Triton Inference Server [46], and vLLM [30] operate at this layer.

Layer 4: Orchestration (Scaling, Routing, Load Balancing). Above the model server, orchestration handles deployment concerns: scaling replicas in response to load, routing requests to appropriate instances, balancing load across the deployment. Systems like Kubernetes, Ray [37], and cloud-native services (Amazon SageMaker, Google Cloud AI Platform) provide primitives at this layer. Autoscaling for ML workloads differs fundamentally from web services due to the high cost of cold starts and the bursty nature of inference traffic [47].

Layer 5: API Layer (REST, gRPC, Streaming). The API layer exposes inference capabilities to client applications. For traditional ML, REST and gRPC suffice. For LLMs, streaming APIs (Server-Sent Events, WebSockets) are essential to deliver tokens as they are generated rather than waiting for complete responses—reducing perceived latency even when total latency remains unchanged.

Metrics That Matter

Serving systems are evaluated along multiple dimensions that often trade off against each other. Choosing the right metrics—and understanding their interactions—is essential for system design.

Latency Metrics. Latency measures the time from request arrival to response completion. For serving systems, percentile latencies matter more than means:

- **P50 Latency:** Median response time—the typical user experience.

- **P99 Latency:** 99th percentile—the tail latency that affects 1% of requests but often correlates with user abandonment. Dean and Barroso [35] showed that tail latency dominates at scale.
- **P99.9 Latency:** For high-scale deployments serving millions of requests daily, even 0.1% tail latency affects thousands of users.
- **TTFT (Time to First Token):** For LLMs, the latency until the first token appears—critical for perceived responsiveness.
- **ITL (Inter-Token Latency):** For LLMs, the time between consecutive tokens—determines the "typing speed" users perceive.

The distinction between TTFT and ITL reflects a fundamental characteristic of LLM serving: latency is not a single number but a time series. A system might achieve excellent TTFT through prioritized prefill but suffer poor ITL due to memory bandwidth constraints during decode.

Throughput Metrics. Throughput measures system capacity:

- **Requests/second:** Traditional throughput measure for classification models.
- **Tokens/second:** For LLMs, both input (prefill) and output (decode) token throughput matter. Prefill throughput is typically compute-bound; decode throughput is memory-bound.
- **Concurrent Requests:** How many requests can be in-flight simultaneously—particularly important for LLMs where each request consumes significant GPU memory for its KV cache.

Efficiency Metrics. Efficiency metrics capture value delivered per resource consumed:

- **SLO Attainment:** Percentage of requests meeting latency targets—the primary objective for production systems.
- **Cost Efficiency:** Throughput per dollar of infrastructure cost. Cloud GPU pricing (A100: \$4–6/hour, H100: \$8–12/hour) makes this metric increasingly important.
- **GPU Utilization:** Actual compute utilization vs. theoretical peak. Inference workloads typically achieve 30–60% utilization [36]; systems research aims to close this gap.

Table 2.1: Metrics comparison across model types. Moving down the rows, the bottleneck shifts from compute to memory, output becomes unpredictable, and scheduling must become increasingly dynamic.

Model Type	Latency	Throughput	Bottleneck	Mem/req	Output	Scheduling
Classification (ResNet, VGG)	P99	Req/s	Compute (FLOPs)	Constant	Fixed (logits)	Static/adaptive batch
Object Detection (YOLO, DETR)	P99	Frames/s	Compute (FLOPs)	Constant	Variable (boxes)	Static/adaptive batch
NLP / Seq2Seq (BERT, T5)	P99	Req/s	Compute + memory	Small KV or none	Fixed or short seq.	Static batching
LLMs (Llama, GPT)	TTFT + ITL	Tokens/s (in+out)	Memory bandwidth	Linear in seq. length	Highly var. (10–4K+ tok)	Continuous (iteration-level)

Batching as the Fundamental Optimization

Batching—processing multiple requests together—is the most important optimization in inference serving. GPUs have massive parallel compute capacity but high fixed costs per invocation: kernel launch overhead (microseconds), memory transfer latency, and context setup. Batching amortizes these fixed costs across multiple requests.

The impact is dramatic. For image classification with ResNet-50, the difference between batch size 1 and batch size 32 can exceed **20× in throughput** [23]. At batch size 1, ResNet-50 achieves approximately 50 images/second on a modern GPU; at batch size 8, throughput exceeds 1100 images/second with ∼90% GPU utilization. The GPU has thousands of cores; a single image uses only a fraction of them. Batching multiple images fills the GPU and amortizes the kernel launch overhead.

However, batching introduces a fundamental tradeoff. If the system waits to accumulate a batch, early-arriving requests experience queuing delay. If it doesn't wait long enough, batches remain small and throughput suffers. This **batching-latency tradeoff** is fundamental to model serving.

Systems have developed increasingly sophisticated approaches to this tradeoff:

- **Static batching:** Fixed batch sizes with timeouts—simple but inflexible.
- **Adaptive batching:** Dynamic batch sizes based on queue depth and latency targets. Clipper [25] introduced AIMD-based (Additive Increase Multiplicative Decrease) adaptation using quantile regression to estimate P99 latency as a function of batch size.
- **Continuous batching:** For LLMs, requests join and leave batches dynamically at each iteration—a fundamental shift from request-level to iteration-level scheduling

(Section 2.2).

For LLMs, batching is more complex. Static batching—waiting for all requests in a batch to complete—performs poorly because generation lengths vary wildly. Consider a batch containing requests that will generate 50, 100, 150, and 500 tokens. With static batching, all sequences must be padded to 500 tokens. Then requests completing at 50, 100, 150 tokens sit idle, wasting GPU cycles. Additionally, new requests cannot enter until the 500-token request completes—a single long request delays the entire batch.

This observation motivated iteration-level scheduling (continuous batching), one of the key advances in LLM serving discussed in Section 2.2.

How LLM Serving Works

Before surveying the LLM era historically, it is useful to make the mechanics of a single LLM request explicit. Most of the systems problems in later chapters follow directly from this request lifecycle. An LLM serving system is not simply running a large neural network once; it is coordinating text preprocessing, transformer execution, incremental state management, scheduling, and streaming output under tight latency constraints.

From text to tokens. A user request arrives as text: a prompt, a chat transcript, system instructions, or tool context. The model, however, does not operate directly on characters or words. It operates on a sequence of integer **token IDs**. A tokenizer maps text into tokens, usually subword units learned from data so that common words may be single tokens while rare words are split into smaller pieces [48, 49]. For example, a word like "serving" might be one token in a model's vocabulary, while a rare technical term might be decomposed into several token fragments. The exact tokenization matters for systems because all major serving costs scale with token count rather than character count: prompt processing time, KV cache memory, batching decisions, and streaming latency are all measured in tokens. After generation, the server performs the inverse operation, **detokenization**, converting generated token IDs back into text fragments that can be streamed to the client.

The transformer forward pass. Modern LLMs are built from the transformer architecture [50]. At a high level, a transformer first maps each token ID to a dense vector embedding, adds positional information, and passes the sequence through a stack of transformer blocks. Each block contains two major computations. The first is **self-attention**: each token forms a query vector and compares it against key vectors from tokens in its context, producing weights used to combine value vectors. This mechanism lets the representation of one token depend on relevant earlier tokens, enabling long-range dependencies such as resolving what "it" refers to in a paragraph or carrying a code variable across many lines. The second is a feed-forward network applied independently to each position, providing additional nonlinear transformation after attention has mixed contextual

information. After many such layers, the model produces logits over the vocabulary; a sampling procedure chooses the next token according to parameters such as temperature, top-k, or top-p.

The key serving fact is that attention is powerful because each position can read from the context, but expensive because the context grows. For a prompt of length n, full self-attention compares positions against one another, creating work and memory traffic that grow with sequence length. During generation, the model must repeatedly attend to the prompt plus all previously generated tokens. Without a specialized execution strategy, each new token would require recomputing attention information for the entire prior sequence. That would make interactive generation prohibitively expensive.

KV caching. The standard solution is the **KV cache**. In every attention layer, the model projects token representations into queries (Q), keys (K), and values (V). For a newly generated token, the query is new, but the keys and values for all earlier tokens have already been computed. The serving system therefore stores those K and V tensors and reuses them on subsequent steps. Instead of recomputing the full history, each decode step computes the new token's K and V, appends them to the cache, and attends over the cached history.

KV caching changes the asymptotic and practical cost of generation. It avoids repeated computation, but it converts computation into long-lived memory state. Unlike model weights, which are loaded once and shared across all requests, the KV cache is per request and grows with prompt length plus output length. Unlike activation tensors in ordinary inference, it cannot be freed after a layer finishes; it must remain resident until the request completes, is cancelled, or is evicted and later recomputed or restored. This is the source of the memory-management problem that later motivates PagedAttention and vLLM.

Prefill and decode. LLM serving therefore has two distinct phases. The **prefill phase** processes the input prompt. Because all prompt tokens are known at once, the model can process them in parallel within a forward pass, using efficient attention kernels and filling the initial KV cache. Prefill is typically compute-intensive: a long prompt contains many tokens, but the GPU can exploit substantial parallelism across those tokens. The latency users experience before the first generated token, TTFT, is strongly affected by prefill time plus any queuing delay.

The **decode phase** begins after the prompt has been prefetched into the cache. Decode is autoregressive: the model produces one new token, appends that token's K and V tensors to the KV cache, samples the next token, and repeats until an end-of-sequence token, length limit, or cancellation. This phase exposes far less parallelism within a single request because token $i{+}1$ cannot be generated until token i exists. Serving systems recover efficiency by batching many requests at the same decode iteration, but each request may finish at a different time and each active request carries a growing KV cache. Decode is therefore often memory-bandwidth-bound rather than compute-bound: the GPU

repeatedly streams model weights and KV cache data from high-bandwidth memory to produce one more token per sequence.

The serving lifecycle. Putting these pieces together, a production LLM request follows a concrete path. The API layer accepts a prompt and generation parameters, then the server tokenizes the text and places the request in a scheduler queue. The scheduler admits the request when enough batch capacity and KV cache space are available. The engine runs prefill to populate the cache and produce the first token distribution; once a token is sampled, the server can start streaming detokenized text back to the client. The request then enters a loop: schedule the next decode iteration, assemble a batch of active requests, execute the transformer forward pass, append KV entries, sample output tokens, stream completed text fragments, and remove any requests that have finished. When a request completes or the client disconnects, the system must release its KV cache blocks, scheduler state, and output buffers promptly.

This lifecycle explains why LLM serving is not merely "large model inference." Tokenization creates the unit of work; prefill and decode create different bottlenecks; autoregression turns one request into many sequential scheduling decisions; and KV caching turns transient computation into dynamic per-request memory. The rest of this chapter shows how inference systems evolved toward abstractions that exploit exactly this structure.

2.2 A Brief History of Inference Systems: The Three-Era Framing

The evolution of inference serving systems closely tracks the evolution of the models they serve. As model architectures have changed—from classical ML to deep learning to LLMs—the systems challenges have changed with them. This section surveys that evolution through three eras, establishing the historical context for the author's contributions.

Figure 2.2 provides a visual roadmap: system milestones are plotted against the three eras defined by dominant model architectures, showing how inference complexity has grown from simple CPU-based batch scoring through GPU-accelerated model servers to the memory-management-dominated LLM era.

Era 1: Classical ML — Prediction as Microservice

In the era before deep learning dominance, inference was rarely recognized as a systems problem. Classical ML models—logistic regression, random forests, gradient boosted trees—were small (kilobytes to megabytes), CPU-efficient, and fast. A typical deployment wrapped a scikit-learn model [51] in a Flask endpoint:

```
1 @app.route('/predict')
```

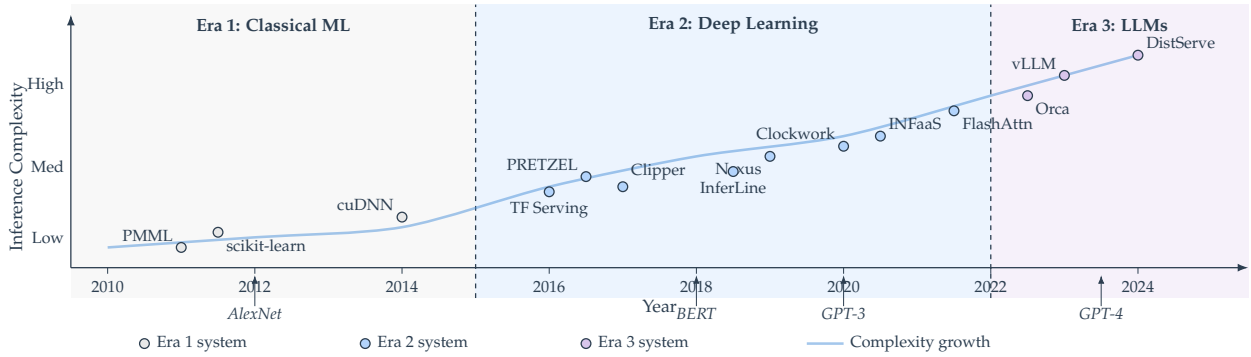

Figure 2.2: Three Eras of Inference Serving—Systems, Models, and Complexity. System milestones are plotted against the three eras defined by dominant model architectures. Inference complexity grows from simple CPU-based batch scoring through GPU-accelerated model servers to the memory-management-dominated LLM era.

```
def predict():
    features = extract_features(request.json)
    return model.predict(features)
```

This approach was sufficient because the models were cheap. A single CPU core could handle hundreds of predictions per second. Latency was dominated by network round-trips, not inference computation. When scale was needed, organizations simply deployed more containers—the same horizontal scaling used for stateless web services.

The systems complexity in this era lay elsewhere: in feature engineering pipelines that transformed raw data into model inputs, and in batch prediction systems that pre-computed predictions for large datasets. PMML [52] enabled model interchange between applications, reducing deployment time from months to days. Real-time serving, when needed, was handled by the same infrastructure that served web pages.

The exception was large-scale recommendation systems. Companies like Google, Facebook, and Netflix were already running inference at massive scale, but their systems looked more like distributed databases than model servers. Google's early ad ranking systems processed billions of predictions daily using MapReduce-style batch computation [53] and distributed key-value stores for feature retrieval. Facebook's recommendation infrastructure evolved from simple heuristics to elaborate pipelines spanning thousands of machines [19]. These systems borrowed heavily from HPC traditions—parameter servers [54], distributed hash tables, and scatter-gather patterns—rather than inventing new ML-specific abstractions.

For most practitioners, prediction serving was not recognized as a distinct systems problem because it did not need to be.

Era 2: Deep Learning Changes the Game

The ImageNet moment (2012)—when AlexNet [55] achieved a 10-percentage-point improvement over prior methods—changed inference serving fundamentally. Neural networks were orders of magnitude more expensive than classical models. GPU acceleration became essential, but GPUs introduced new systems challenges that CPU-based serving infrastructure couldn't address.

The Batching Imperative. GPU inference without batching is dramatically inefficient. The difference stems from GPU architecture: thousands of parallel compute units that remain idle with small batches, plus fixed costs (kernel launch, memory transfer) that are amortized over larger batches. Throughput and energy efficiency increase monotonically with batch size [23], making dynamic batching essential for efficient GPU utilization [56].

The Model Server Emerges. TensorFlow Serving [24], released by Google in 2016, was the first major "model server"—a system purpose-built for serving ML models rather than generic web services. It introduced foundational abstractions that influenced all subsequent systems:

- **Servables:** Versioned model objects that can be loaded, unloaded, and swapped.
- **Model Versioning:** Support for A/B testing and gradual rollouts with multiple model versions active simultaneously.
- **Batching Scheduler:** Configurable batching with timeout and size limits.

TensorFlow Serving handles "tens of millions of inferences per second" and is used in over 1100 of Google's own projects [24]. It established the model server as a distinct architectural component, but its tight coupling to TensorFlow limited flexibility, and its design predates the LLM era.

Model Composition Complexity. Production ML deployments rarely involve a single model. Recommendation systems combine user embeddings, item embeddings, ranking models, and diversity filters. Facebook's ML infrastructure serves "tens of millions of inferences per second" with embedding layers exceeding 10GB and complex multi-model pipelines [20]. Computer vision pipelines chain detection, classification, and postprocessing. This composition introduced orchestration complexity atop the batching problem—coordinating multiple models while meeting end-to-end latency objectives.

Seminal Academic Work

The academic community produced several influential systems during this era, charting a progression from problem definition through scheduling and composition to increas-

ingly declarative abstractions. Each advanced the field but also exposed limitations that motivated subsequent work.

Clipper (NSDI 2017). Clipper [25], from UC Berkeley's RISELab, established the vocabulary of model serving research. It introduced **prediction caching** to avoid redundant computation (achieving 30–50% reduction in model invocations for some workloads), **adaptive batching** via AIMD-controlled batch sizes with quantile regression for P99 latency estimation, and **model selection** using bandit and ensemble methods to navigate accuracy/latency tradeoffs at serving time. Clipper's contribution was conceptual as much as technical—it framed prediction serving as a distinct systems problem worthy of research attention. However, its design assumed relatively simple models and did not address the GPU-specific scheduling challenges that would become dominant.

Nexus (SOSP 2019). Nexus [26], from University of Washington and Microsoft Research, shifted focus to GPU cluster scheduling for DNN-based video analysis. Its central technique, **squishy bin packing**, recognized that required resources and latency vary jointly with batch size, enabling simultaneous optimization of batching and placement. Nexus also introduced **prefix batching**—detecting and batching common subtrees in model graphs to share computation across requests with different downstream processing—and a global cluster scheduler driven by load statistics. The system achieved 9.4–12.7× throughput improvements over Clipper and TensorFlow Serving while maintaining only 0.27% SLO violations on a 100-GPU cluster, demonstrating the value of workload-specific optimization. Its design, however, predated LLMs.

Clockwork (OSDI 2020). Clockwork [27], from Max Planck Institute and Emory University, exploited a property unique to inference: *predictability*. Unlike general computation, inference latencies for a given model and input size fall within tight, known bounds. Clockwork centralizes all scheduling in a single controller that issues timed actions (LOAD, INFER) with predicted execution windows, achieving end-to-end predictability from the bottom up. The result: thousands of models served concurrently while meeting 100ms latency targets for 99.9999% of requests—a six-nines guarantee that earned the Distinguished Artifact Award at OSDI'20. This reliance on predictable execution times, however, became a fundamental limitation for LLMs, where output length and thus execution time is inherently unpredictable.

Pipeline and Cost-Aware Systems. A parallel line of work addressed model composition and cost optimization. PRETZEL [57] enabled end-to-end optimizations by looking inside model pipelines, reducing 99th percentile latency by 5.5×, memory footprint by 25×, and increasing throughput by 4.7×. InferLine [28] formalized SLO-aware pipeline serving, achieving up to 7.6× cost reduction and 34.5× lower SLO miss rates through combinatorial planning and network calculus—the author is a co-author on this work.

MArk [47] combined dynamic batching with predictive autoscaling, reducing costs by up to 7.8× compared to Amazon SageMaker. GPU cluster schedulers complemented these efforts at the infrastructure layer: Gandiva [58] achieved 26% utilization improvement through time-slicing and migration, and Tiresias [59] improved average job completion time by 5.5× over YARN schedulers.

INFaaS (ATC 2021). INFaaS [60], from Stanford University, raised the abstraction level with a **model-less** interface: users specify accuracy requirements and latency constraints rather than selecting specific models. The system generates optimized model variants across batch sizes and hardware platforms via TensorRT and Neuron compilers, dynamically selects among them at serving time, and shares workers across users and models to increase utilization. Evaluated across 44 architectures and 270 model variants, INFaaS achieves 2× higher throughput and violates latency SLOs 3× less frequently than baselines. By hiding model selection complexity from users, it pointed toward declarative serving—but its design, like those before it, predated LLMs and their unique memory management demands.

AlpaServe (OSDI 2023). AlpaServe [61], from UC Berkeley, tackled statistical multiplexing for bursty inference workloads. Its key insight was that when multiple models share infrastructure, their bursty arrival patterns overlap, enabling higher utilization than dedicated deployments. More surprisingly, AlpaServe demonstrated that model parallelism can serve a multiplexing purpose even when a single model fits on one device, processing requests at up to 10× higher rates or handling 6× more burstiness within latency constraints for >99% of requests. AlpaServe operated at model-level granularity, however, and could not exploit finer-grained sharing opportunities within GPU execution.

Era 3: The LLM Discontinuity

The emergence of large language models represents not an incremental evolution but a discontinuity in inference serving. The mechanics described in Section 2.1 break fundamental assumptions of prior serving systems in ways that demand entirely new approaches. This section shifts from mechanics to systems consequences: why existing model servers were insufficient, and how the field responded.

LLMs break the assumptions of prior serving systems along every dimension that matters for system design. Autoregressive decode turns a single user request into many sequential scheduling decisions. Output length is **unpredictable**: a response might be 10 tokens or 4,000, and the system cannot know until generation completes. Pre-allocating resources for the maximum wastes memory; allocating dynamically demands memory management far more sophisticated than anything Era 2 systems provided.

The root cause of that memory pressure is the **KV cache**—the key and value tensors cached across attention layers so that earlier tokens need not be recomputed at each step.

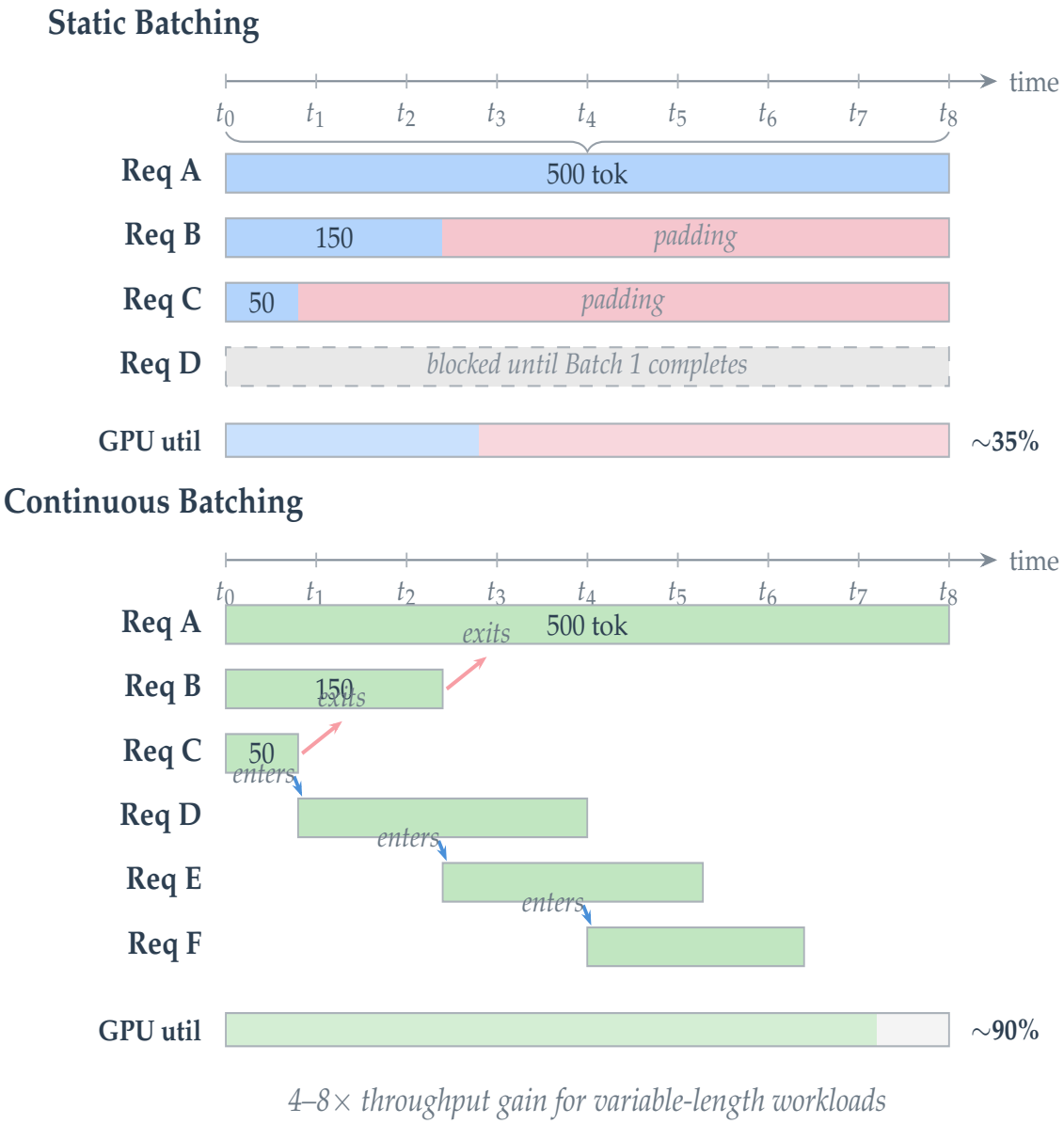

Figure 2.3: Static batching vs. continuous batching. In static batching, completed requests waste GPU cycles (shown in rose) while waiting for the longest request. Continuous batching allows requests to enter and exit at each iteration, achieving ∼90% effective utilization.

The KV cache grows linearly with sequence length and layer count; for a 70B-parameter model with 80 layers and 16K context, a single request's cache can exceed 10 GB. Unlike model weights, which are fixed at load time, KV cache size depends entirely on runtime behavior (prompt length plus generated tokens), so static reservation is inherently wasteful. Traditional memory allocators compound the problem: they assume contiguous allocation, but KV caches grow one token at a time, producing fragmentation that wastes 60–80% of GPU memory on typical workloads [30]. On modern GPUs, the KV cache—not the model weights—becomes the binding constraint on how many requests can execute concurrently.

These characteristics also invert the performance bottleneck. Era 2 models were compute-bound: more FLOPS meant more throughput. LLMs during the decode phase are **memory-bandwidth-bound**: model weights and KV cache must be loaded from HBM for every token, and the actual matrix multiplications finish faster than the memory transfers. Optimizing for FLOPS alone yields diminishing returns; the scarce resource is memory bandwidth and capacity.

The Orca Breakthrough. Orca [29], from Seoul National University and FriendliAI, demonstrated that LLM serving required fundamentally different scheduling. Its key insight was **iteration-level scheduling** (also called continuous batching): rather than treating a batch as an atomic unit that starts and finishes together, the scheduler operates at the granularity of individual token-generation steps. Completed requests exit immediately—no waiting for stragglers—and new requests join mid-batch without waiting for assembly. Figure 2.3 illustrates the contrast: static batching pads short requests to match the longest, wasting GPU cycles, while continuous batching keeps utilization high by filling vacated slots. The performance impact is dramatic—**4–8× throughput improvements** for workloads with high length variation—lifting GPU utilization from 30–60% to 80–95% and transforming practical deployment economics. Orca further introduced selective batching, recognizing that non-attention operations (Linear, GeLU, LayerNorm) are token-count-independent and can be batched aggressively while attention is handled separately.

Orca proved that the scheduling problem was solvable, but it remained a research prototype whose memory management still assumed contiguous allocation, suffering the same fragmentation described above. The author's work on vLLM addressed this limitation through PagedAttention (Chapter 4).

Complementary Advances. Alongside scheduling and memory innovations, several complementary techniques reshaped LLM serving. FlashAttention [31] introduced IO-aware attention that accounts for GPU memory hierarchy, achieving 2–4× speedup and 5–20× memory reduction by never materializing the full attention matrix; FlashAttention-2 [62] reached 50–73% of theoretical maximum FLOPS on A100. Speculative decoding [32, 63] uses smaller "draft" models to propose tokens verified in parallel by the target model,

achieving 2–3× decoding speedup without changing output distributions. DistServe [33] disaggregated prefill and decode phases onto separate machines, serving 7.4× more requests within latency constraints. KV cache compression through heavy-hitter retention [64] and quantization to 3 bits [65] reduces the KV-cache footprint needed to serve longer contexts. These techniques are orthogonal to the scheduling and memory management innovations above and are discussed further in Chapter 4.

2.3 The Open Source Landscape

The academic systems surveyed above solved key technical problems but most remained research prototypes; production deployment required open-source implementations that prioritized usability, maintenance, and ecosystem integration alongside raw performance. This section surveys the major open-source inference systems that practitioners actually use, establishing the context in which the author's contributions (Ray Serve, vLLM) emerged and competed.

TensorFlow Serving. TensorFlow Serving [24], discussed in Section 2.2 as the system that established the model server abstraction, also proved influential as open-source software. Its maturity (active development since 2016), model versioning support, and integration with Google Cloud made it the default for enterprises with TensorFlow training pipelines. However, its tight framework coupling (PyTorch models require conversion) and static batching approach limit its applicability to LLM workloads.

Triton Inference Server. NVIDIA's Triton Inference Server [46] positioned itself as a universal platform, supporting TensorFlow, PyTorch, ONNX, TensorRT, and custom backends with native multi-GPU scheduling, dynamic batching, and ensemble scheduling for model pipelines. Its strength is hardware-level optimization through tight TensorRT and CUDA integration, though this coupling limits portability to non-NVIDIA accelerators. For LLM serving, Triton delegates to TensorRT-LLM for continuous batching (NVIDIA's "in-flight batching"), an approach that delivers strong performance but requires navigating two systems with different configuration paradigms.

Text Generation Inference (TGI). HuggingFace's TGI, released in late 2022, was the first widely-used open-source system purpose-built for LLM serving. Its contribution was accessibility: direct integration with the HuggingFace model ecosystem, a familiar API, and continuous batching out of the box lowered the barrier for practitioners without deep systems expertise. However, its Rust core with Python bindings raised the contribution barrier for ML researchers, and its memory management remained simpler than PagedAttention-based systems, limiting its competitiveness as the field advanced.

The vLLM Moment

vLLM [30], the author's central contribution (detailed in Chapter 4), emerged in June 2023 and rapidly became the dominant open-source LLM serving system. **PagedAttention** achieved near-zero memory waste (<4%) through virtual memory abstractions for KV caches; it partitions the KV cache into blocks (typically 16 tokens per block), enabling non-contiguous storage analogous to OS virtual memory. This memory efficiency translated directly into **performance**: **2–4× throughput improvements** over state-of-the-art systems (FasterTransformer [40], Orca) at the same latency, with gains more pronounced for longer sequences, larger models, and complex decoding algorithms. Finally, **usability** accelerated adoption—an OpenAI-compatible API, a Python-native codebase accessible to ML researchers, and active community support lowered the barrier to deployment.

vLLM's trajectory demonstrates the dissertation's thesis: purpose-built abstractions for inference—in this case, PagedAttention for KV cache management—yield transformative improvements that general-purpose approaches cannot match.

Table 2.2: Open-source inference systems comparison.

System	Organization	Framework Support	LLM Optimizations	Primary Strength
TensorFlow Serving	Google	TensorFlow	Limited	Maturity, versioning
Triton	NVIDIA	Multi-framework	Via TensorRT-LLM	Hardware optimization
TGI	HuggingFace	PyTorch/HF	Continuous batching	HuggingFace ecosystem
vLLM	UC Berkeley	PyTorch	PagedAttention	Memory efficiency

The Economics Driving Open Source Adoption

The rapid adoption of open-source LLM serving systems reflects economic realities. Inference costs dominate the economics of deployed ML systems. Patterson et al. [15] found that inference accounts for roughly 60% of Google's ML energy use, consistently across 2019–2021. Gartner projects inference will consume 55% of AI-optimized IaaS spending in 2026, rising to 65% by 2029 [66]. The absolute numbers are staggering: reported leaked Microsoft financial documents indicate OpenAI's Azure inference spend reached $3.77 billion in calendar year 2024 and $8.67 billion through Q3 2025 alone [18]—dwarfing public estimates for the one-time training compute cost of GPT-4-class models [17].

Per-token costs have fallen precipitously—**280-fold** between November 2022 and October 2024 according to Stanford's 2025 AI Index Report [17]—a pace OpenAI CEO Sam Altman characterizes as approximately 10× per year, far exceeding Moore's law [67]. The a16z "LLMflation" analysis documents an even steeper 1,000× decline over three years: GPT-3-equivalent inference dropped from $60 per million tokens (November 2021) to $0.06 (late 2024, via Llama 3.2 3B) [68]. Open-weight models have accelerated this trend; DeepSeek-V3 [69] demonstrated that open-weight mixture-of-experts models could ap-

proach leading closed-source performance with aggressive efficiency-oriented design, intensifying price pressure on proprietary serving. Yet lower unit costs have not reduced total spending—they have *increased* it, as Gartner predicts overall inference expenditure will continue rising even as per-token costs fall by 90% through 2030 [70].

This economic dynamic explains why purpose-built serving systems matter: the throughput gains described in Section 2.3 translate directly to proportional cost reduction—potentially billions of dollars annually at frontier-model scale.

2.4 Setting the Stage

The preceding survey reveals a consistent pattern: as model architectures evolved, general-purpose serving approaches repeatedly proved insufficient, motivating purpose-built systems that exploited the specific structure of new workloads. This dissertation contributes three such purpose-built approaches, each addressing gaps in the prior art.

Gap 1: Pipeline-Aware Serving

Prior systems optimized individual models but struggled with model pipelines. Production ML deployments compose multiple models—preprocessing, candidate generation, ranking, postprocessing—into pipelines with end-to-end latency constraints. These constraints must be decomposed into per-stage budgets, but this decomposition interacts with batching decisions in complex ways.

Clipper [25] addressed single-model serving; it could not reason about pipeline composition. INFaaS [60] optimized model selection but assumed independent models. The gap: no system provided SLO-aware optimization across entire serving pipelines.

Chapter 3 addresses this gap through InferLine's SLO-aware pipeline optimization—achieving up to **7.6× cost reduction** and **34.5× lower SLO miss rates**—and Ray Serve's production implementation of these ideas.

Gap 2: Memory-First LLM Design

The LLM discontinuity demanded memory-first system design. Prior systems treated memory as a constraint to work around; LLM serving requires treating memory as the primary resource to optimize.

Orca [29] introduced continuous batching but left memory management unsolved—contiguous allocation still dominated, with the fragmentation costs described in Section 2.2. TGI and other systems inherited similar limitations. The gap: no system provided efficient memory management for the variable-size, dynamically-growing KV caches that LLMs require.

Chapter 4 addresses this gap through vLLM and PagedAttention—a complete rethinking of inference system architecture around KV cache management. The author's sub-

sequent work extends these ideas: Jenga generalizes PagedAttention for hybrid architectures mixing attention and state-space models; Pie exploits NVLink interconnects for unified CPU-GPU memory management.

Gap 3: Fine-Grained GPU Sharing

Even with sophisticated serving systems, GPU utilization remains low. Model-level multiplexing (AlpaServe) cannot address the idle cycles between kernel executions—the "bubbles" that waste compute. Jeon et al. [36] characterized this at Microsoft scale: GPUs are "monolithic resources that cannot be shared at fine granularity," and fragmentation delay accounts for $\sim$**80% of queuing time** for large jobs.

Prior systems operated at request-level or model-level granularity. The gap: no system exploited sharing opportunities within GPU execution, below the model boundary.

Chapter 5 addresses this gap through kernel-level multiplexing—first via software (OoO JIT kernel combining) and later via hardware-supported approaches (GreenContext leveraging CUDA 12.4+ SM partitioning). These systems point toward a future where inference frameworks manage not just models but the underlying compute fabric itself.

These three gaps are **logically independent**: a deployment can suffer from poor pipeline decomposition while using efficient per-model runtimes (Gap 1 without Gap 2), deploy vLLM-class memory management yet leave GPU SM time unused below the kernel level (Gap 2 without Gap 3), or exploit kernel-level multiplexing without addressing end-to-end SLO budgeting across pipeline stages (Gap 3 without Gap 1). This orthogonality explains why incremental fixes—better Kubernetes autoscaling here, a larger GPU there—often fail to move end-to-end metrics: the binding constraint may lie in a different layer than where engineering effort concentrates. Each gap also manifests through a characteristic failure mode. Gap 1 surfaces when per-model latency targets are met but end-to-end SLOs are violated because queuing and batching interact across stages—users experience timeouts even when each component in the monitoring dashboard is "green." Gap 2 manifests as OOM errors or severe batch-size collapse when sequence lengths vary, because contiguous memory reservation forces the system to plan for the worst case on every request. Gap 3 appears as high GPU bills paired with low utilization dashboards: memory-bound decode leaves SMs idle, yet coarse-grained sharing cannot fill the bubbles between kernels. These failure modes map directly to the contributions in Chapters 3–5.

This dissertation does not claim that closing these gaps eliminates the need for good models, sound MLOps, or careful product design. It reduces waste given a model and a workload—a necessary but not sufficient condition for efficient deployment. Training-system concerns—data pipelines, optimizer state, fault-tolerant backpropagation—are largely out of scope except where they intersect inference (artifact storage, model distribution, A/B testing of checkpoints).

With these foundations in place, the dissertation turns to its contributions. Chapter 3 addresses Gap 1 through model composition—InferLine's SLO-aware pipeline optimization and Ray Serve's production implementation. Chapter 4 tackles Gap 2 with vLLM and PagedAttention. Chapter 5 explores Gap 3 through kernel-level GPU multiplexing.

Chapter 3

Model Composition — From Clipper to Ray Serve

3.1 The Composition Problem

Why Serve Multiple Models on Shared Infrastructure?

Production machine learning systems rarely deploy a single model in isolation. For example, recommendation engines combine collaborative filtering with content-based models. Multi-stage pipelines chain preprocessing, embedding generation, ranking, and post-processing. Ensemble methods aggregate predictions from multiple models to improve accuracy. Facebook's ML infrastructure serves "tens of millions of inferences per second" across their data centers [19], with personalized feed/ranking, visual content understanding, and natural language processing workloads characterized by embedding layers with model sizes exceeding 10GB and low arithmetic intensities [20]. These composite workloads present a systems challenge that single-model serving cannot address: *how do we share infrastructure efficiently across multiple models while meeting end-to-end latency objectives?*

This chapter traces a lineage of systems for serving many models efficiently in production, especially when multiple models cooperate to answer the same user request. The progression runs from foundational vocabulary (Clipper), through SLO-aware model composition (InferLine), to production model multiplexing and composition (Ray Serve). Across these systems, the central lesson is that multi-model serving requires abstractions that expose pipelines, latency objectives, and shared resources rather than treating each model as an isolated microservice.

The case for multi-model serving rests on three interrelated observations about production ML infrastructure.

The first observation is **cost efficiency**—or more precisely, the staggering inefficiency of dedicated provisioning. Facebook reported that ML inference workloads across their data centers exhibit GPU utilization as low as 5–10% when models are provisioned with

dedicated resources [19]. A model serving 10 queries per second on a dedicated A100 GPU leaves 95% of the GPU's capacity idle. When organizations deploy dozens or hundreds of models, this waste compounds into millions of dollars of infrastructure sitting unused. Jeon et al. [36] characterized this problem at Microsoft scale, showing that GPUs are "monolithic resources that cannot be shared at fine granularity" and that fragmentation delay accounts for approximately 80% of queuing time for large jobs. The opportunity is clear: sharing GPUs across models can reduce infrastructure costs by 5–10$\times$ while maintaining the same aggregate throughput.

Why is utilization so low with dedicated resources? The answer lies in **workload burstiness**. Individual models exhibit highly variable arrival patterns. A product recommendation model spikes during sale events; a fraud detection model spikes during high-transaction periods; a content moderation model spikes when viral posts spread. These bursts rarely align. Prior work on inference serving shows that burstiness directly shapes SLO attainment and provisioned cost [47, 61]. Production ML workloads can exhibit CV values above 1, meaning variance exceeds the mean arrival rate.

Statistical multiplexing exploits this non-alignment of peaks. When models share infrastructure, peaks from one model fill valleys from another. The aggregate load is smoother than individual loads, enabling higher utilization without sacrificing latency. AlpaServe [61] demonstrated this principle dramatically, handling **10$\times$ higher request rates** or **6$\times$ more burstiness** while staying within latency constraints for more than 99% of requests.

The mathematics explain why multiplexing yields order-of-magnitude rather than incremental improvements. Consider two models with peak loads at different hours: Model A handles product recommendations, spiking to 500 req/s during lunch; Model B handles fraud detection, spiking to 500 req/s at market close. Serving each on dedicated A100 GPUs requires provisioning for peak capacity—500 req/s each, 1000 total—with most capacity sitting idle most of the time. Multiplexed on shared hardware with combined capacity of 600 req/s, the system serves both within SLOs because peaks don't overlap. Utilization rises from approximately 25% (dedicated) to approximately 80% (shared). More formally, consider measuring arrivals in fixed-size time windows. For N independent equal-rate workloads with similar variance, the aggregate mean grows linearly with N, while the aggregate coefficient of variation decreases as $1/\sqrt{N}$—the classical smoothing effect of superposed arrival processes in statistical multiplexers [71]. For workloads with CV $= 2.0$ (highly bursty), aggregating 16 independent workloads reduces effective burstiness to CV $= 0.5$, allowing substantially higher average utilization before tail-latency violations dominate.

Figure 3.2 provides an illustrative sanity check for this effect. Statistical multiplexing achieves a **25 percentage point** (pp) utilization improvement over dedicated provisioning—from $\sim$25% (single model) to $\sim$50% (both models shared)—while maintaining **100% SLO attainment** (SLO $=$ 2000 ms). p99 latency increases from $\sim$114 ms (dedicated) to $\sim$373 ms (shared) but remains well within the SLO. Measured on an

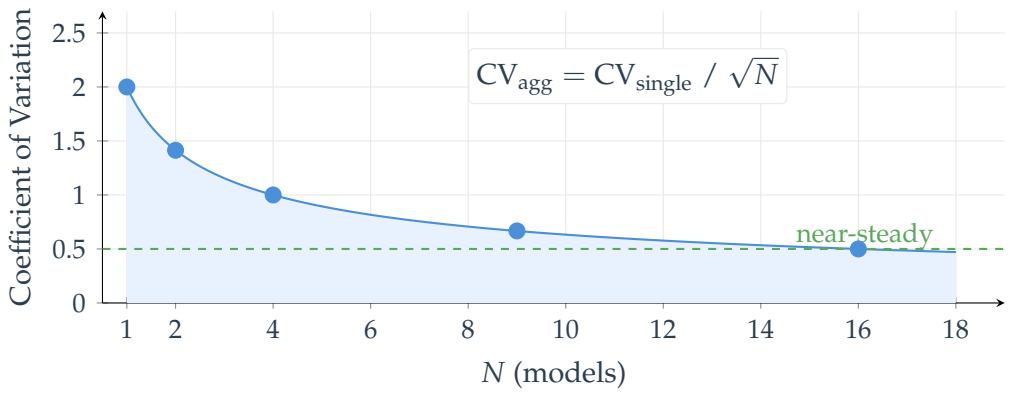

Figure 3.1: The smoothing effect of statistical multiplexing. As N independent bursty workloads are aggregated onto shared infrastructure, the aggregate coefficient of variation decreases as $1/\sqrt{N}$—transforming highly bursty traffic (CV $= 2.0$) into near-steady load (CV $= 0.5$) at $N = 16$.

NVIDIA H200 with ResNet-50; two independent bursty arrival streams (CV $= 1.5$, 20 req/s each).

The third observation shifts from individual models to **model composition**. Real workloads are DAGs (directed acyclic graphs), not isolated models. A visual search pipeline chains: image preprocessing $\rightarrow$ feature extraction $\rightarrow$ embedding generation $\rightarrow$ similarity search $\rightarrow$ ranking $\rightarrow$ postprocessing. Each stage has different compute characteristics, batch size preferences, and hardware affinities. The insight is that treating this pipeline as a compositional unit—rather than as independent stages—enables global optimization impossible with per-model systems. Anyscale's production-serving guidance distilled four recurring composition patterns: pipelines (sequential stages), ensembles (parallel models with aggregation), business logic integration (models embedded in application code), and online learning (models updated based on feedback) [72].

The Resource–SLO Tradeoff

Model composition introduces a fundamental tradeoff between resource efficiency and latency guarantees. More aggressive sharing improves utilization but increases interference between models. A model that would meet its 50 ms SLO on dedicated resources might experience 100 ms tail latency when sharing with bursty neighbors.

This tradeoff admits no universal solution. The right balance depends on workload characteristics (burstiness, arrival rates), SLO stringency (p50 vs. p99, target latency), and cost sensitivity (infrastructure budget constraints). Dean and Barroso [35] established that "temporary high latency episodes which are unimportant in moderate size systems may come to dominate overall service performance at large scale"—the famous "tail at scale"

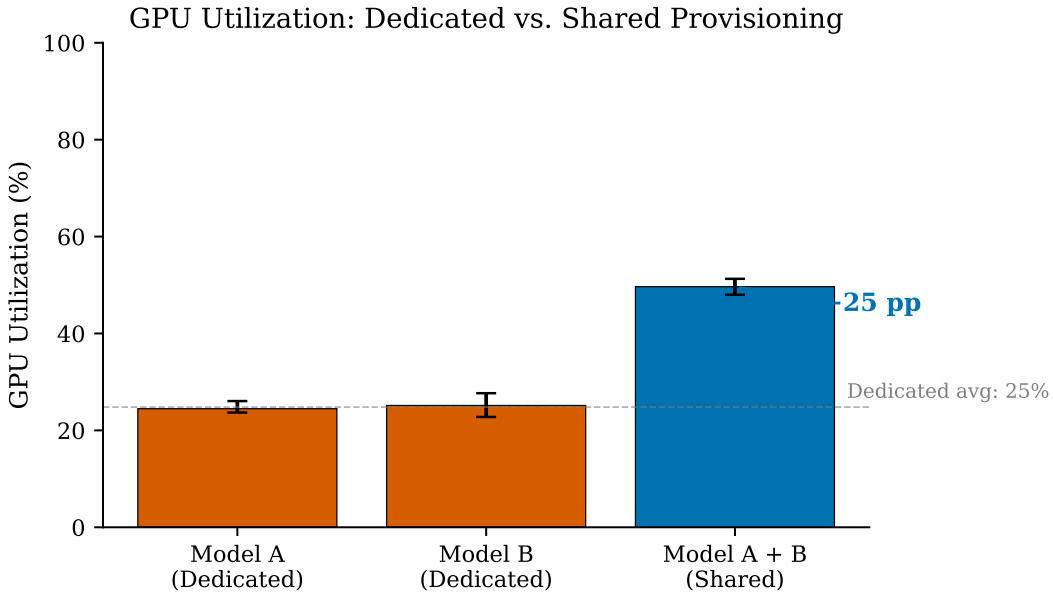

Figure 3.2: GPU utilization under dedicated vs. shared serving. Multiplexing two ResNet-50 models on one H200 GPU nearly doubles mean GPU utilization (24–27% → 50%) while maintaining 100% SLO attainment at a 2000 ms target. p99 latency increases from 114 ms (dedicated) to 373 ms (shared)—a 3.3× increase—but remains an order of magnitude below the SLO. Three independent runs; error bars show bootstrap 95% CI.

problem. This insight applies directly to inference serving: p99 latency matters more than mean latency, and building "latency tail-tolerant" systems requires purpose-built abstractions.

Systems must provide mechanisms for navigating this tradeoff, not fixed points on the curve. The central tension is clear:

- *Aggressive multiplexing* → higher utilization → lower cost → interference risk
- *Conservative isolation* → guaranteed SLOs → lower utilization → higher cost

The systems contributions of this chapter provide progressively more sophisticated tools for navigating this tradeoff: Clipper introduced dynamic batching to navigate it at the single-model level; InferLine introduced SLO decomposition to optimize at the pipeline level; Ray Serve introduced autoscaling and composition primitives to manage it at the application level.

Pipelines as the Unit of Serving

A key insight from this work is that **pipelines—not individual models—should be the unit of serving**. End-to-end SLOs apply to complete pipelines, not individual stages. A pipeline with stages taking 10 ms, 20 ms, and 30 ms has a 60 ms end-to-end latency, but optimizing each stage independently might not minimize end-to-end cost. Perhaps the 20 ms stage could be slowed to 25 ms (reducing its resource allocation) while the 30 ms stage is sped to 25 ms (using those freed resources), achieving the same 60 ms total at lower cost.

This observation motivates the **SLO decomposition problem**: given an end-to-end latency constraint, how should budget be allocated across pipeline stages to minimize total cost? Naive equal splitting—giving each of n stages $1/n$ of the budget—leaves significant savings on the table. InferLine demonstrated that optimized decomposition, reflecting stage characteristics and exploiting heterogeneity, achieves **30–50% additional cost savings** compared to equal splitting. This is the core technical problem that defines pipeline serving as distinct from per-model serving.

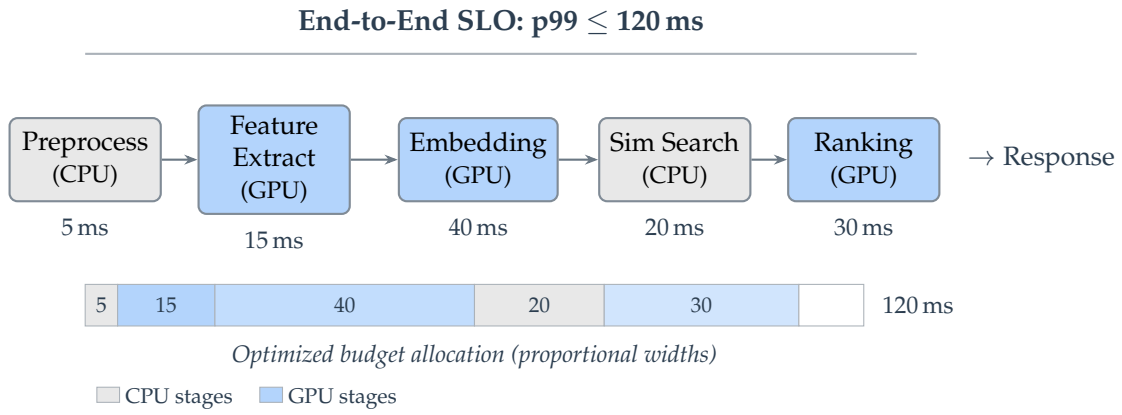

Figure 3.3: Pipeline serving example. A five-stage visual search pipeline with heterogeneous compute requirements and an end-to-end SLO of 120 ms. Optimized budget allocation assigns more time to GPU-bound stages (embedding, ranking) and less to lightweight CPU stages (preprocessing, search)—exploiting stage heterogeneity for cost savings. Equal splitting would over-provision CPU stages and under-provision GPU stages.

Amazon SageMaker's Inference Pipelines [73], which co-locate 2–15 containers on the same EC2 instance for preprocessing, prediction, and postprocessing, represent commercial validation of this insight. The pipeline abstraction enables feature processing to run with inference in a single call, avoiding the latency of separate service invocations.

What This Chapter Demonstrates

This chapter makes the dissertation's central argument concrete through the evolution from single-model serving to multi-model composition and multiplexing. The three systems examined—Clipper, InferLine, and Ray Serve—collectively demonstrate the thesis's three claims. The experimental figures in this chapter are intentionally modest: they serve as illustrative microbenchmarks and sanity checks for the systems principles, while the primary evidence comes from the published systems, cited deployments, and reported production outcomes.

Claim 1: Inference workloads have exploitable structure. The structure here is compositional: production workloads are pipelines with heterogeneous stages, bursty arrival patterns, and end-to-end latency constraints. This structure is not incidental—it emerges from how ML applications are built (feature extraction $\rightarrow$ prediction $\rightarrow$ postprocessing) and how users behave (bursty requests concentrated around peak hours). Statistical multiplexing exploits temporal misalignment; SLO decomposition exploits stage heterogeneity; pipeline composition exploits data flow structure. Each optimization targets a specific facet of inference workload structure.

Claim 2: General-purpose abstractions fail to capture this structure. Deploying pipelines on generic infrastructure—Kubernetes pods with round-robin load balancing—ignores the relationships between stages. Equal SLO allocation across stages leaves 30–50% cost savings unrealized. Generic autoscalers cannot exploit the predictability of profiled latency models. Standard memory allocators cannot handle the variable batch sizes that inference workloads require. The systems in this chapter succeed precisely because they reject general-purpose abstractions in favor of inference-specific ones.

Claim 3: Purpose-built abstractions yield order-of-magnitude improvements. InferLine's SLO decomposition achieves 7.6$\times$ cost reduction over naive allocation. Ray Serve's composition and multiplexing primitives enabled substantial production cost reductions, including an Anyscale-published report that Samsara reduced annual inference cost by approximately 50% after adopting Ray Serve [74]. These gains require abstractions designed specifically for pipeline serving—they cannot be achieved by better configuration of generic infrastructure.

The progression from Clipper to Ray Serve also illustrates how systems disciplines mature: from vocabulary establishment (Clipper named the problem and introduced caching, batching, model selection), through formalization (InferLine proved optimal solutions existed and characterized the configuration space), to production implementation (Ray Serve made the insights practical at scale). This maturation trajectory mirrors how database systems evolved from file managers to query optimizers to distributed transaction coordinators, and how networking evolved from packet handlers to congestion controllers to software-defined networks. Inference serving is following the same path—each system in this chapter advanced not just implementation but problem formulation. This is the mark of a maturing discipline.

3.2 Clipper: Establishing the Vocabulary

The Problem Clipper Addressed

Before the ML community could solve model serving problems systematically, it needed vocabulary to describe them. In 2016, deploying ML models to production was an ad-hoc process. Data scientists trained models in Python using scikit-learn or TensorFlow. Production engineers then rewrote those models in C++ or Java for performance, wrapped them in custom serving code, and deployed them as microservices. This process was slow (weeks to months per model), error-prone (reimplementation bugs), and inefficient (no sharing across models).

The gap between model development and production deployment was substantial. Sculley et al. [21] observed that "machine learning offers a fantastically powerful toolkit for building useful complex prediction systems quickly," but warned that "these quick wins are not free"—real-world ML systems incur "massive ongoing maintenance costs" from boundary erosion, entanglement, hidden feedback loops, and configuration complexity. Gartner [22] found that only 47% of AI projects reach production, with scaling operations and proving ROI as the primary blockers.

Clipper [25], published at NSDI 2017, addressed this gap by framing prediction serving as a distinct systems problem—separate from both training and general web serving. The paper provided a general-purpose prediction serving system that could accept models from any ML framework without modification, optimize serving through system-level techniques transparent to the model, and support multiple models on shared infrastructure. The author contributed to the Clipper open source project; later, the author carried several of Clipper's serving lessons into Ray Serve as project lead.

Key Abstractions

Clipper introduced four abstractions that became foundational to model serving:

Prediction caching stores results for previously seen queries and returns cached predictions for identical or similar future queries. The key systems insight was that prediction functions are often pure—deterministic given the same inputs—which makes caching safe and can directly reduce both latency and cost when applications issue repeated requests.

Adaptive batching treats the latency–throughput tradeoff as a control problem rather than a static configuration choice. Clipper used an AIMD policy: additively increase batch size while latency stays within the SLO, then multiplicatively back off when the SLO is exceeded. Under light load, this keeps batch size near one to minimize latency; under heavy load, it accumulates larger batches to improve throughput. Subsequent work confirmed the importance of dynamic batching for GPU utilization [56, 23].

Model selection routes queries among model variants with different accuracy–latency tradeoffs. A small model might answer quickly with lower accuracy, while a larger model

might be slower but more accurate. Clipper used this abstraction for graceful degradation: under heavy load or strict latency constraints, the system can shift traffic to faster variants and return to more accurate variants when capacity allows.

Model containers made Clipper framework independent. Models were packaged with their runtime dependencies in Docker containers exposing a standard prediction interface, providing isolation, reproducibility, and a path to supporting scikit-learn, TensorFlow, and other frameworks without rewriting the serving system. This container abstraction influenced later systems including TensorFlow Serving [24], NVIDIA Triton [75], and Ray Serve.

Contributions and Limitations

Clipper's primary contribution was conceptual: establishing prediction serving as a systems discipline with its own abstractions and optimization opportunities. The vocabulary Clipper introduced—caching, batching, model selection—became standard terminology in the field. Clipper influenced subsequent systems including MArk [47], INFaaS [60], and the broader model serving ecosystem.

However, Clipper's design reflected its era (2016–2017). It assumed relatively simple models that could be batched effectively with standard techniques. It didn't address GPU-specific scheduling challenges that became critical with larger neural networks. And it focused on individual models rather than pipelines—a limitation that motivated InferLine.

The limitations of Clipper—single-model scope, no pipeline awareness, no end-to-end SLO optimization—directly motivated the next system in this arc. Where Clipper established that prediction serving is a systems problem, InferLine demonstrated that the problem extends beyond individual models to the pipelines that compose them.

3.3 InferLine: SLO-Aware Pipeline Serving

The Pipeline Serving Problem

While Clipper established vocabulary for single-model serving, production ML workloads are predominantly pipelines. A recommendation request might flow through feature extraction, candidate generation, ranking, and filtering stages—each with different models, hardware requirements, and performance characteristics. Optimizing each stage independently ignores the end-to-end nature of SLOs and leaves cost savings unrealized.

InferLine [28], published at SoCC 2020, addressed this pipeline serving problem with a principled approach to SLO-aware configuration. The author was a co-author on this work, contributing to the formalization of SLO-aware pipeline serving and the evaluation.

Problem Formulation

InferLine formalized pipeline serving as an optimization problem. Consider a pipeline with n stages. Each stage i has:

- A set of model variants V_i with different accuracy–latency profiles
- A range of valid batch sizes B_i affecting latency and throughput
- A replication factor r_i controlling how many instances serve that stage

The configuration space is the Cartesian product: $\prod(V_i \times B_i \times R_i)$. For a pipeline with 5 stages, 3 variants per stage, 10 batch sizes, and 10 replication options, this space contains approximately 10^{12} **configurations**—far too large for exhaustive search.

The optimization objective is to minimize cost (total GPU-hours) subject to an end-to-end SLO constraint (e.g., p99 latency ≤ 100 ms). This requires:

1. Predicting latency for each configuration
2. Decomposing end-to-end SLO into per-stage budgets
3. Searching the configuration space efficiently

A closely related line of work, Nexus [26], framed this class of problem as **squishy bin packing**—a variant of bin packing where required resources and latency vary with batch size. The term "squishy" reflects that resources are elastic based on batching decisions, distinguishing it from traditional bin packing with fixed-size items.

Nexus extended this insight to cluster-scale GPU scheduling, demonstrating that serving-specific batch-aware placement dramatically outperforms general GPU job schedulers. On a 100-GPU cluster serving video analytics workloads, Nexus achieved **1.8–12.7× higher request rates** than Clipper and TensorFlow Serving while maintaining 99th percentile latency SLOs with only **0.27% SLO violations**. The key innovation was recognizing that batch size affects both latency *and* resource consumption—enabling joint optimization that general schedulers cannot perform. This result provides complementary evidence at cluster scale: purpose-built scheduling abstractions yield order-of-magnitude improvements over general-purpose GPU schedulers.

The Profiler: Characterizing Stage Performance

InferLine's **profiler** characterizes the latency distribution of each stage as a function of batch size and model variant. For each (model variant, batch size) pair, the profiler measures, mean latency, latency variance, and tail latency (p99, p99.9). These measurements enable latency prediction: given a configuration, the profiler estimates the latency distribution for each stage, which can then be composed into end-to-end latency predictions.

A key insight was that profiling is cheap relative to the optimization savings. Profiling each configuration takes seconds to minutes; the resulting profile enables optimization decisions that save hours to days of GPU time over the pipeline's lifetime. This insight—that offline characterization enables online optimization—recurs throughout systems research, from database query optimizers [6] to network protocol tuning.

The Planner: Searching Configuration Space

Given profiled performance characteristics and an end-to-end SLO, the **planner** searches for the minimum-cost configuration. InferLine's planner uses a two-phase approach:

Phase 1: SLO Decomposition. The end-to-end SLO must be decomposed into per-stage budgets. Naive equal splitting (give each stage $1/n$ of the budget) is suboptimal—stages have different latency profiles, and the decomposition should reflect these differences.

InferLine decomposes based on stage characteristics: stages with steeper latency–cost tradeoffs receive more budget (they can use it efficiently); stages with flat tradeoffs receive less. This decomposition is iteratively refined during optimization. The key insight is that heterogeneity is an opportunity, not a complication—exploiting differences between stages yields significant savings.

Phase 2: Per-Stage Optimization. Given per-stage budgets, each stage can be optimized independently. For each stage, find the minimum-cost configuration (variant, batch size, replication) that meets the latency budget. This reduces the combinatorial problem to independent per-stage searches, which are tractable.

The planner iterates between phases, adjusting decomposition based on per-stage optimization results, until convergence. This iterative approach draws on techniques from network calculus for reasoning about latency bounds in queuing systems.

The Controller: Runtime Adaptation

Static configuration is insufficient for real workloads. Arrival rates fluctuate. Upstream dependencies experience delays. Hardware performance varies. InferLine's **controller** monitors pipeline performance and triggers reconfiguration when SLOs are violated.

The controller operates on two timescales:

- **Fast adaptation:** Adjusting batch sizes and routing weights without reprovisioning (milliseconds)
- **Slow adaptation:** Triggering replanning when fast adaptation is insufficient (seconds to minutes)

This separation enables responsive adaptation while avoiding thrashing. The controller uses SLO violation rates as the primary signal, triggering replanning when violations exceed a threshold over a measurement window.

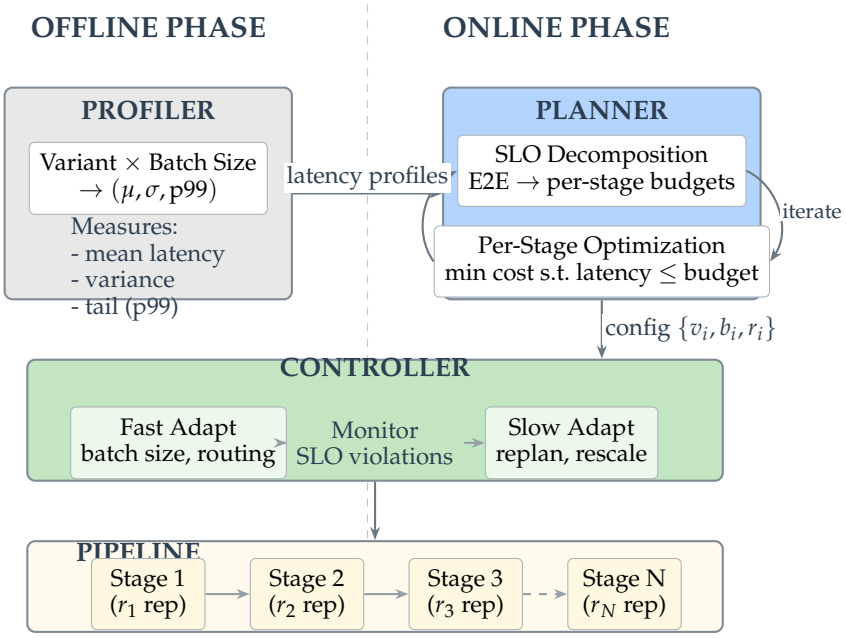

Figure 3.4: InferLine architecture. The Profiler (offline) characterizes each stage's latency as a function of model variant and batch size. The Planner decomposes the end-to-end SLO into per-stage budgets and finds minimum-cost configurations. The Controller monitors runtime SLO attainment and adapts on two timescales—fast batch/routing adjustments and slow replanning when violations persist.

Key Insights and Evaluation

InferLine's evaluation revealed several insights that informed subsequent work:

Batching decisions compound across stages. A suboptimal batch size at an early stage propagates effects downstream. Pipeline optimization requires joint consideration, not independent per-stage tuning.

SLO decomposition is non-trivial. Equal splitting across stages leaves **30–50% cost savings** on the table compared to optimized decomposition. The decomposition should reflect stage characteristics and exploit heterogeneity.

Profiling enables prediction. Despite workload variability, profiled latency models predict actual performance accurately. This supports the profiler-planner-controller architecture as a sound design pattern.

To illustrate why SLO decomposition yields such large savings, consider a concrete example. A three-stage pipeline—preprocessing, inference, postprocessing—must meet a 90 ms end-to-end SLO. Naive equal splitting allocates 30 ms per stage. But preprocessing has a flat latency–cost curve: relaxing its budget from 30 ms to 40 ms saves \$0.50/hour while adding only 10 ms. Postprocessing has a steep curve: tightening from 30 ms to 20 ms costs only \$0.30/hour additional. The rebalanced configuration (40 ms, 30 ms,

20 ms) achieves the same 90 ms SLO at $0.20/hour net savings—a 25% reduction from naive allocation. Across InferLine's evaluation workloads, where stage heterogeneity varied widely, these savings ranged from 30% to 50% depending on pipeline characteristics.

Figure 3.5 illustrates these findings on a simplified setup. Optimized SLO decomposition provides **0–36% cost savings** over equal splitting: zero savings when stages are homogeneous (equal split is already optimal), rising to 32% at medium heterogeneity ($\sigma = 0.3$) and 36% at high heterogeneity ($\sigma = 0.6$). Savings grow monotonically with heterogeneity. Measured on a 3-stage CPU stub pipeline; 3 independent runs.

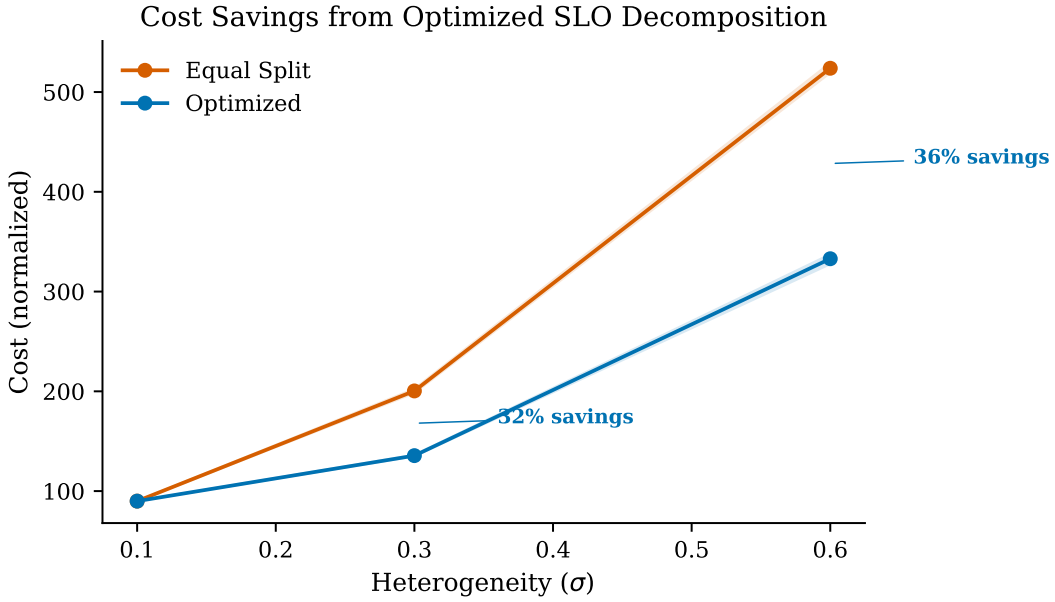

Figure 3.5: Cost savings from optimized SLO decomposition vs. equal splitting across three heterogeneity levels (3-stage pipeline, CPU stubs). At low heterogeneity ($\sigma = 0.1$), equal allocation is already optimal—the optimizer finds no room to improve. At medium and high heterogeneity, concentrated allocation yields 32–36% cost reduction. Savings are monotonically non-decreasing with heterogeneity as the optimizer exploits differences in per-stage latency–cost slopes.

Evaluation

InferLine was evaluated against two baselines representing common production practices:

Over-provisioned baseline: Provision each stage to meet its SLO independently at peak load, with no pipeline-level coordination. This guarantees SLOs but wastes resources during non-peak periods—the approach most organizations default to when deploying new pipelines.

Static configuration baseline: Profile the pipeline offline, select a single configuration, and run it unchanged regardless of load fluctuations. This avoids runtime overhead but cannot adapt to workload changes—representing the state-of-practice for organizations that have invested in some optimization.

Against these baselines, InferLine achieved dramatic improvements on multi-stage inference pipelines representative of production workloads. Cost savings reached **7.6× reduction** compared to over-provisioned baselines—for example, achieving 99.8% SLO attainment at $8.50 cost versus the baseline's 93.7% attainment at $36.30 cost. SLO attainment improved by up to **34.5× lower miss rate** compared to static configurations—99.3% SLO attainment at $15.27 versus baseline 75.8% at $24.63. These results generalized across different serving frameworks (Clipper, TensorFlow Serving), demonstrating that the optimization principles are framework-agnostic.

These results established that pipeline-aware serving is not merely desirable but essential for cost-efficient production ML. The 7.6× improvement is not achievable through better configuration of per-model systems—it requires treating the pipeline as the fundamental unit of optimization.

Limitations and Follow-on Work

InferLine, as a research prototype, had limitations that motivated production-focused follow-up work:

- **Simulation vs. production:** Evaluation used traces and simulations; production deployment raises additional challenges (fault tolerance, monitoring, deployment)
- **Static pipelines:** InferLine assumed fixed pipeline structure; production systems often have dynamic pipeline composition
- **Single-tenant:** InferLine optimized individual pipelines; production systems must multiplex across many pipelines

These limitations motivated Ray Serve—translating InferLine's insights into production infrastructure. Where Clipper lacked pipeline awareness and InferLine lacked production hardening, Ray Serve would need to combine InferLine's compositional thinking with the operational robustness that real deployments demand. The question was not whether InferLine's insights were correct—the 7.6× cost reduction proved they were—but whether those insights could survive contact with production reality.

InferLine has also influenced a broader line of SLO-aware inference-serving work. Some systems build directly on InferLine, while others developed concurrently around similar cost–latency concerns:

Table 3.1: Related and follow-on work around SLO-aware inference serving.

System	Year	Extension
MArk [47]	2019	Concurrent SLO-aware autoscaling + serverless for spikes (7.8× cost reduction)
IPA [76]	2023	Integer programming for accuracy–cost–latency tradeoffs
Loki [77]	2024	Hardware + accuracy scaling for edge-cloud pipelines

3.4 Ray Serve: From Research to Production

Ray Foundation and Design Decisions

The Ray Foundation

Research systems demonstrate what is possible; production systems demonstrate what is practical. **Ray Serve** represents the translation of academic insights—from Clipper, InferLine, and the broader serving literature—into production infrastructure. The author started Ray Serve at Anyscale in Q4 2019, defining its core abstractions and guiding its evolution over three years (2019–2022).

Ray Serve builds on **Ray** [37], the distributed computing framework developed at UC Berkeley's RISELab and published at OSDI 2018. Ray provides three foundations essential for serving:

The Actor Model. Ray actors are stateful, long-running processes that can receive messages and maintain state between invocations. This maps naturally to model serving: each model replica becomes an actor that loads model weights (state) and processes prediction requests (messages). The actor abstraction provides isolation between models, enables independent scaling, and simplifies state management.

Distributed Scheduling. Ray's scheduler places actors on nodes, manages resources, and handles failures. Ray scales beyond **1.8 million tasks per second** [37]. For serving, this means replica placement, resource allocation, and automatic restart after crashes. Ray Serve inherits this functionality rather than reimplementing it.

Shared Memory (Plasma). Ray's object store enables zero-copy data sharing between processes on the same node. For pipelines where data flows between stages, this eliminates serialization overhead that would otherwise dominate latency.

Starting from Zero: Design Decisions

Several foundational design decisions shaped Ray Serve's trajectory:

Models as Actors. The first design decision was mapping model replicas to Ray actors. Each **deployment** (a model or processing function with a specified resource allocation)

runs as one or more actor replicas. Incoming requests are routed to replicas, which process them and return responses.

This decision enabled:

- Natural scaling (create more actor replicas)
- Independent resource allocation (each actor has CPU/GPU limits)
- Fault tolerance (Ray restarts crashed actors)
- Framework independence (any Python code can be an actor)

Declarative Deployments: Kubernetes-Style State Reconciliation. Rather than imperative deployment commands ("start 3 replicas on GPU nodes"), Ray Serve uses declarative specifications: "I want 3 replicas with 1 GPU each." The system continuously reconciles actual state to declared state—a pattern borrowed directly from Kubernetes.

This declarative approach mirrors Kubernetes' control loop architecture. In Kubernetes, controllers watch for changes in Custom Resource Definitions (CRDs) and reconcile cluster state to match the declared specification. Ray Serve's Controller actor implements the same pattern: it maintains a desired state (from user configuration), observes actual state (running replicas, their health, queue depths), and takes corrective actions (spawning replicas, draining excess capacity, restarting failures) to converge the two.

When running on Kubernetes via **KubeRay**, this parallel becomes explicit. The RayService Custom Resource encapsulates both the Ray cluster and the Serve application in a single Kubernetes manifest. The KubeRay controller watches RayService resources, measures desired versus actual worker groups, and continuously adjusts to match the specification—the classic Kubernetes reconciliation loop. Status updates write back to the CR, including cluster health, restart events, and application-level diagnostics.

This declarative approach simplifies operations (users specify what they want, not how to achieve it) and enables infrastructure-as-code patterns (deployments can be version-controlled and reviewed).

Composition Primitives. Building on InferLine's insight that pipelines are the unit of serving, Ray Serve provides composition primitives: deployments can call other deployments, forming DAGs. The `DeploymentHandle` abstraction (formerly `ServeHandle`) enables synchronous and asynchronous invocations between deployments, with the system managing routing and load balancing.

```
@serve.deployment
class Preprocessor:
    async def __call__(self, request):
        return preprocess(request)

@serve.deployment
class Model:
    async def __call__(self, features):
```

```
        return self.model.predict(features)

@serve.deployment
class Pipeline:
    def __init__(self, preprocessor, model):
        self.preprocessor = preprocessor
        self.model = model

    async def __call__(self, request):
        features = await self.preprocessor.remote(request)
        return await self.model.remote(features)
```

Sensible Defaults. The initial Ray Serve design emphasized simplicity over configurability. User feedback—gathered through extensive sales calls, debugging sessions, and feature discussions—revealed that most users want things to "just work" rather than tuning dozens of parameters. Ray Serve provides sensible defaults for batching, routing, and scaling, with advanced configuration available for users who need it.

From InferLine to Ray Serve: Translation, Not Reimplementation

A critical design question was how Ray Serve should relate to InferLine's profiler-planner-controller architecture. InferLine demonstrated that automated profiling, planning, and runtime control could achieve **7.6× cost reductions** through sophisticated optimization. Should Ray Serve implement the same full optimization loop as its default production control plane?

The answer was no—not because the optimization was unimportant, but because production serving needed a smaller, more debuggable default. InferLine represents the academic optimum: a full-system planner that reasons across model variants, batch sizes, replicas, and SLO decomposition. Production environments introduce constraints that make this difficult to deploy as an always-on automatic layer:

- **Model diversity:** Production deployments use arbitrary Python code, not just standard ML models. Profiling arbitrary code is harder than profiling known model architectures.
- **Dynamic pipelines:** Production systems often have conditional routing, feature flags, and runtime-determined pipeline structure. Static profiling cannot capture this dynamism.
- **Operational simplicity:** Full automatic optimization introduces a "magic" layer that operators cannot easily debug when things go wrong.

Instead, Ray Serve **operationalized InferLine's insights through composition primitives and local control loops**. The deployment graph API makes pipelines explicit in user code. Users define the DAG structure; Ray Serve handles the execution mechanics

(routing, independent queue-based autoscaling, batching, and failure handling). In practice, independent per-deployment autoscaling based on queue depth often works well enough, while users retain enough visibility to diagnose behavior. The full InferLine-style optimizer remains the upper bound and a direction for future automation rather than the default production interface.

The translation is conceptual rather than architectural:

Table 3.2: Translation from InferLine components to Ray Serve.

InferLine Component	Ray Serve Translation
Profiler	User-provided metrics, external monitoring
Planner	Declarative deployment configuration
Controller	Queue-depth autoscaler + health monitoring
Pipeline DAG	Deployment graph API

This design choice reflects a broader lesson: research systems can afford deeper automation because they operate in controlled environments; production systems often start with explicit composition and simple control loops because operators need to understand and debug behavior.

Core Architecture

Ray Serve's architecture consists of four core components:

Controller. A global actor managing the control plane—creates, updates, and destroys actors. Handles all Serve API calls. All controller state is checkpointed to Ray's **Global Control Store (GCS)** for fault tolerance. The Controller implements the reconciliation loop: it observes deployment specifications, compares them to running state, and takes corrective actions. This is the Kubernetes controller pattern implemented at the application layer.

HTTP Proxy. A Uvicorn HTTP server (one per head node by default, scalable to all nodes). Accepts requests, forwards them to replicas, and returns responses. Supports both HTTP and gRPC protocols.

Replicas. Actors executing prediction code. Each replica processes individual requests and may batch them using the `@serve.batch` decorator. Replicas are the workhorses of the system.

DeploymentHandle Router. Each node runs a router actor that buffers incoming requests and selects replicas. The router implements the **Power of Two Choices** algorithm [78]: randomly sample two replicas, then route to the one with fewer outstanding requests. This algorithm—drawn from load balancing theory—provides near-optimal load distribution without requiring global state. It avoids "herd behavior" when multi-

ple routers exist and works well in Ray's decentralized architecture where each router has incomplete visibility into cluster-wide state.

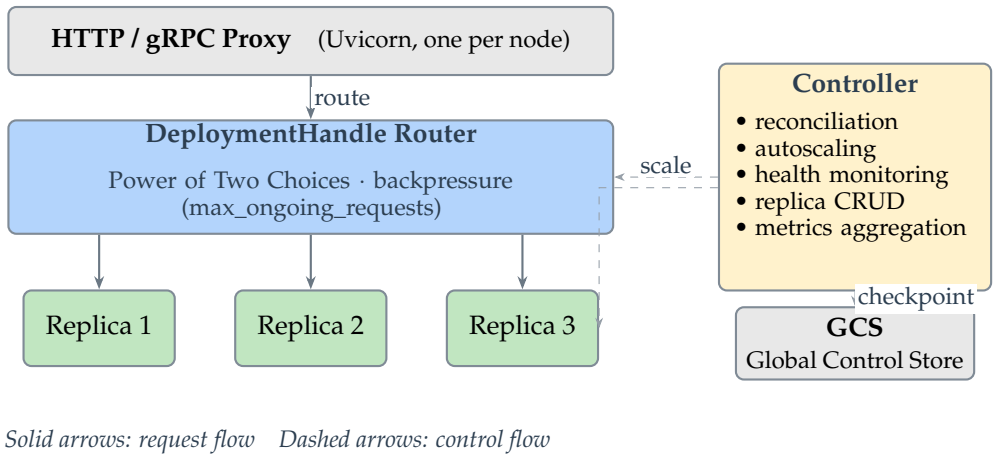

Figure 3.6: Ray Serve request flow. The data plane consists of HTTP proxies, per-node routers implementing Power of Two Choices load balancing, and replica actors executing prediction code. The control plane consists of a global Controller actor implementing Kubernetes-style reconciliation, backed by the Global Control Store (GCS).

Technical Contributions: Algorithms in Detail

Beyond the foundational design, Ray Serve contributed several technical innovations with specific algorithmic implementations:

Queue-Depth Autoscaling. Ray Serve autoscales based on **target ongoing requests**—the desired number of concurrent requests each replica should handle. The scaling formula is:

$$\texttt{num\_replicas} = \lceil \texttt{current\_ongoing\_requests} \,/\, \texttt{target\_ongoing\_requests} \rceil$$

With `target_ongoing_requests` defaulting to 2.0, a deployment receiving 10 concurrent requests scales to 5 replicas. This queue-depth approach differs from latency-based scaling (which requires measuring end-to-end latency) and throughput-based scaling (which requires knowing maximum capacity). Queue depth is observable without profiling.

The autoscaler includes timing controls to prevent thrashing.

The asymmetric delays (30 s up, 600 s down) reflect operational wisdom: scaling up quickly prevents SLO violations; scaling down slowly prevents capacity oscillation.

Power of Two Choices Routing. Request routing uses the Power of Two Choices algorithm [78], which provides near-optimal load balancing with minimal coordination:

Table 3.3: Ray Serve autoscaler parameters.

Parameter	Default	Purpose
`upscale_delay_s`	30 s	Wait before scaling up (avoid reacting to transient spikes)
`downscale_delay_s`	600 s	Wait before scaling down (retain capacity for bursts)
`look_back_period_s`	30 s	Aggregation window for metrics
`metrics_interval_s`	10 s	Metrics reporting frequency

1. Randomly sample two replicas from the deployment
2. Query outstanding request count for each
3. Route to the replica with fewer outstanding requests

This algorithm reduces maximum queue length from $O(\log n / \log \log n)$ (random routing) to $O(\log \log n)$ (power of two choices)—an exponential improvement. It works particularly well in distributed systems where maintaining global state is expensive. The router implements backpressure through `max_ongoing_requests` (default: 5); when a replica reaches this limit, the router retries with exponential backoff (initial: 25 ms, multiplier: 2×, max: 500 ms).

Figure 3.7 illustrates why P2C is a strong default for Ray Serve. Two results are noteworthy. First, P2C reduces p99 latency by **70%** vs. random routing at saturation (5 replicas, $\rho = 1.0$; 1,647 ms vs. 5,446 ms) and **54%** at moderate load (10 replicas; 100 ms vs. 220 ms), consistent with the $O(\log \log n)$ vs. $O(\log n / \log \log n)$ theoretical guarantee. Second—and more practically relevant—P2C's advantage over round-robin grows with service time variability. With uniform jitter ($\sigma \approx 0.1$), round-robin's perfect sequential distribution achieves lower tail latency (63 ms vs. 100 ms at $n = 10$). But as service times become heavier-tailed (modeling GC pauses, cache misses, mixed hardware), P2C overtakes round-robin: at $\sigma = 0.5$ the policies cross over, and at $\sigma = 1.0$ P2C achieves **55% lower p99** than round-robin (333 ms vs. 746 ms). In production, service time distributions are often heavier-tailed than uniform; P2C's ability to route around temporarily slow replicas justifies the modest coordination cost. Measured via discrete-event simulation; 100 req/s, 50 ms mean service, 10 independent runs per configuration.

Batching: From Clipper's AIMD to Explicit Configuration. Clipper introduced AIMD (Additive Increase Multiplicative Decrease) for dynamic batch sizing: increase batch size until latency exceeds SLO, then decrease by 10%. This closed-loop approach adapts automatically but introduces tuning complexity and control overhead.

Ray Serve took a different approach with the `@serve.batch` decorator—explicit, user-configured batching:

```
@serve.batch(
    max_batch_size=16,          # Maximum requests per batch
```

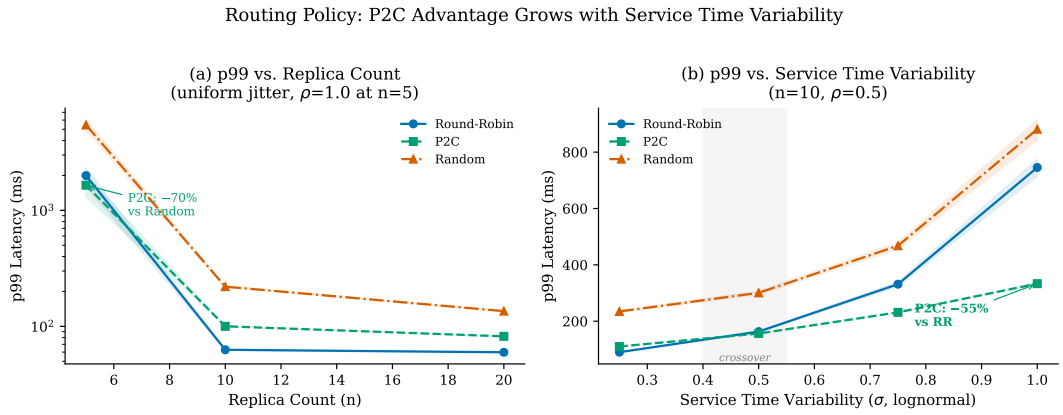

Figure 3.7: Two-panel routing policy comparison (discrete-event simulation; 100 req/s, 50 ms mean service time, 10 runs per configuration). (a) p99 vs. replica count with uniform jitter: P2C reduces tail latency 39–70% vs. random at all load levels; round-robin achieves slightly lower p99 at light load ($n \geq 10$) due to perfect balance. (b) p99 vs. service time variability at $n = 10$ with lognormal service times: P2C overtakes round-robin at $\sigma \approx 0.5$ and achieves 55% lower p99 at $\sigma = 1.0$. Production service times are heavy-tailed (GC pauses, cache misses, mixed hardware), placing real workloads in the regime where P2C dominates.

```
    batch_wait_timeout_s=0.01,    # 10ms wait for full batch
)
async def batch_handler(self, requests: List[Request]) -> List[Response]:
    return self.model.predict_batch([r.data for r in requests])
```

This explicit approach reflects the design philosophy that users understand their workloads better than automated systems. A user serving a model with 50 ms latency budget knows that 10 ms batch wait is acceptable; a user serving a 5 ms model knows it is not. Making batching configuration explicit enables users to apply this knowledge directly.

The tradeoff: Clipper's AIMD adapts automatically to changing conditions; Ray Serve's explicit batching requires users to configure appropriately. In practice, most workloads have stable characteristics, making explicit configuration sufficient. For workloads requiring adaptation, users can implement custom batching logic.

Figure 3.8 illustrates this design tradeoff. All three algorithms (fixed, AIMD, explicit `@serve.batch`) achieve identical throughput of ∼30 req/s at all CV levels (0.5–2.0), confirming that throughput is arrival-rate limited and batching strategy does not affect it. The algorithms diverge on tail latency under burstiness. At CV = 2.0: explicit achieves p99 = 44.5 ms (45% lower than AIMD's 80.7 ms); fixed achieves p99 = 53.2 ms (34% lower than AIMD). AIMD's adaptive batch accumulation amplifies tail latency under bursty traffic—it enlarges batches during bursts, increasing queueing delay precisely when requests are already delayed. Explicit `@serve.batch` with a tuned timeout delivers consis-

tently lower p99 across all CV values (34–45 ms vs. AIMD's 53–81 ms), supporting the Ray Serve design philosophy: users who know their latency budgets can often achieve better tail behavior than a generic automated controller.

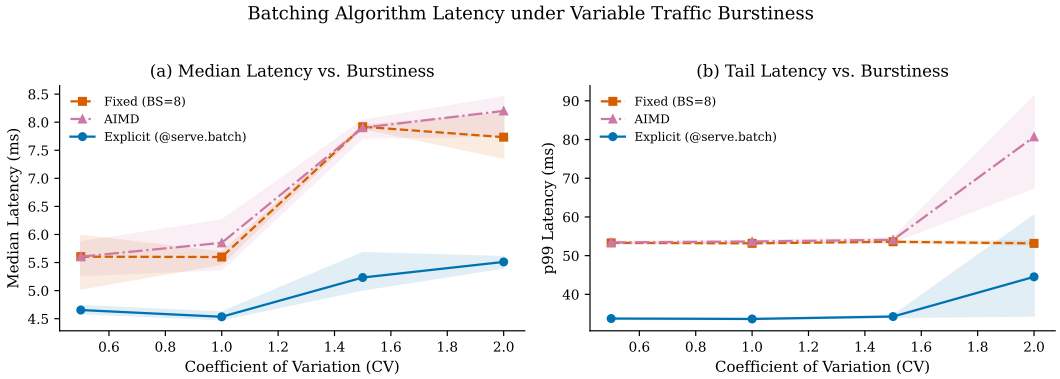

Figure 3.8: Tail latency (p99) vs. arrival burstiness for three batching algorithms. All algorithms achieve identical throughput (∼30 req/s, arrival-rate limited), but diverge on tail latency at high CV. AIMD's adaptive batch accumulation amplifies queueing delay under bursts; explicit `@serve.batch` with tuned timeout achieves 45% lower p99 than AIMD at CV = 2.0.

Fault Tolerance via GCS Checkpointing. Ray Serve achieves fault tolerance through Ray's Global Control Store (GCS). All Controller state—deployment configurations, replica assignments, routing policies—is checkpointed to GCS, which can be backed by external Redis for high availability.

Failure recovery combines Ray's actor restart mechanisms with Serve's reconciliation loop. If a replica or proxy fails, the Controller recreates it and routes traffic to healthy processes; if the Controller fails, Ray restarts it and Serve reloads deployment state from the GCS checkpoint; with GCS high availability, worker pods can continue serving while control-plane state recovers after a head-node failure. This layered approach provides graceful degradation: individual failures do not become system-wide outages, and recovery remains automatic.

Pipeline Composition via Deployment Graphs. The **deployment graph** abstraction directly translates InferLine's insight that pipelines are the unit of serving. Users define an explicit DAG using the `InputNode` API, making data flow visible in code structure—each deployment scales independently, and the system handles routing and failure across the graph.

```
from ray.serve.dag import InputNode

with InputNode() as user_input:
    preprocessed = preprocessor.preprocess.bind(user_input)
    embedding = encoder.encode.bind(preprocessed)
```

```
    candidates = retriever.retrieve.bind(embedding)
    ranked = ranker.rank.bind(candidates, embedding)

app = serve.build(ranked)
```

Deployment graphs support the four composition patterns identified from 100+ production deployments:

- **Chaining:** Sequential stages (preprocessing → inference → postprocessing)
- **Parallelization:** Concurrent branches (multiple models on same input)
- **Ensemble:** Aggregating outputs (bagging, voting, stacking)
- **Dynamic dispatch:** Conditional routing (small model for simple queries, large model for complex)

The gap from InferLine is intentional: InferLine's profiler-planner automatically optimized pipeline configuration for cost–SLO tradeoffs, achieving 30–50% additional savings through optimized SLO decomposition. Ray Serve makes composition and model multiplexing practical in production, while using independent queue-based autoscaling as the default control mechanism. This reflects the production reality that a robust, understandable local policy often delivers enough benefit, while more global optimization remains available to expert users or future systems.

FastAPI Integration. Modern ML deployments need HTTP APIs. Ray Serve integrates with FastAPI, enabling users to define endpoints with FastAPI's syntax while Ray Serve handles scaling and execution. This integration brought Ray Serve to users familiar with FastAPI's patterns.

Translation from Research to Production

The journey from prototype to production spanned three years (2019–2022) and involved challenges not anticipated in research systems:

Working with Users. The author's work involved extensive user engagement—sales calls, debugging sessions, feature discussions. User feedback drove priorities: production hardening over new features, documentation over API elegance. A key realization was that "simple API with great docs beats great API with missing docs." Approximately one-third of development time was spent on non-Serve projects (demos, CI, infrastructure) required to support production users.

Integration with Anyscale. Ray Serve became part of Anyscale's commercial offering, requiring enterprise features: high availability, observability, security. These requirements often conflicted with simplicity goals, requiring careful tradeoff navigation. The deployment/Kubernetes story took two years to get right—a timeline that would have been unacceptable for a research prototype but was necessary for production reliability.

What Broke at Scale. Production scale revealed issues invisible in development:

- Memory leaks in long-running actors requiring careful resource management
- Thundering herd on autoscale events when many replicas start simultaneously
- Race conditions in deployment updates during rolling upgrades
- Observability blind spots in distributed traces spanning multiple deployments

Each issue required instrumentation, diagnosis, and targeted fixes—engineering work that dwarfed the initial implementation effort.

Production Impact. Real-world deployments demonstrated significant value:

- Anyscale published a Samsara deployment report stating that Ray Serve improved production ML pipeline performance and reduced annual ML inference cost by approximately 50% [74].
- Ray Serve and Anyscale Services usage grew by more than 600% in 2023 after general availability [74].
- By Ray Summit 2024, Anyscale reported that Ray was orchestrating more than 1 million clusters per month [79].

Understanding the Research–Production Gap. The reported Samsara 50% cost reduction (approximately 2×) is more modest than InferLine's 7.6× research result. This gap reflects a fundamental design tradeoff between global automation and operational transparency. InferLine's sophisticated profiler-planner architecture automatically optimizes pipeline configurations, achieving greater savings under controlled assumptions. Ray Serve instead exposes composition and model multiplexing through transparent production mechanisms, with independent queue-based autoscaling handling the common case. The 7.6× result requires automated end-to-end optimization across model variants, batch sizes, and replica counts; the reported production result comes from workload-specific operator choices applied through transparent tools. Both show that inference-specific abstractions outperform generic microservice deployments—they differ in how much of the optimization is automated by the system.

Figure 3.9 illustrates this automation gap. On a 5-stage heterogeneous pipeline (CPU–GPU–CPU–GPU–CPU with stage cost weights $[1, 6, 1, 4, 0.5]$ and base latencies $[5, 200, 5, 150, 5]$ ms), the equal-allocation baseline incurs normalized cost 4073. Automated grid search reduces cost to 2853 (30% savings); expert allocation guided by InferLine-style analysis ($a_i \propto w_i \times \text{base}_i$) achieves 2705 (33.6% savings). The automation gap is small: the automated optimizer reaches within 5% of expert-guided allocation without any domain knowledge. Naive proportional allocation ($a_i \propto w_i$, ignoring base latency) achieves 28.6% savings—worse than automated grid search because it misallocates budget away from the GPU bottleneck stages. These results support the InferLine thesis: inference-aware optimization substantially outperforms equal allocation, and automated

optimization can approximate expert tuning in controlled settings. (Note: InferLine's published 7.6× result compares against a single-server non-replicated baseline; the 30–34% here measures savings over a properly-replicated equal-allocation baseline, which is the relevant comparison for Ray Serve's explicit composition model.)

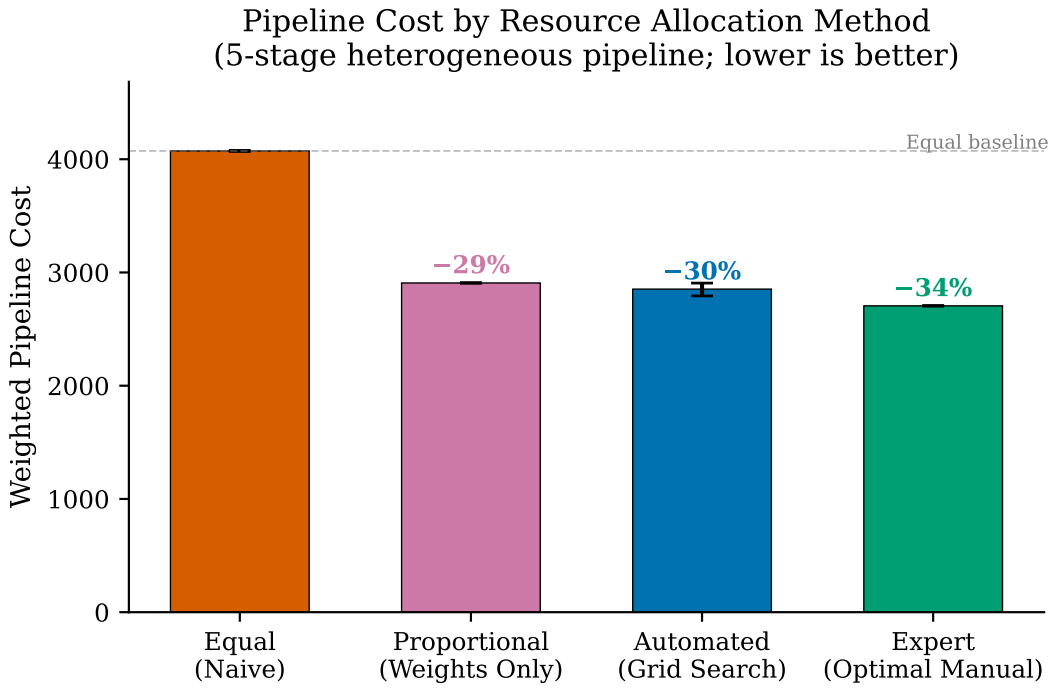

Figure 3.9: Weighted pipeline cost by resource allocation method (5-stage heterogeneous pipeline; lower is better). Automated grid search achieves 30% savings over equal allocation, within 5% of expert-guided (InferLine-style) optimization.

Table 3.4: Evolution of Ray Serve.

Year	Focus	Key Developments
2019	Foundation	Actor-based architecture, basic routing
2020	Users	FastAPI integration, documentation, user engagement
2021	Scale	Autoscaling, pipeline composition, observability
2022	Production	Ray 2.0 integration, deployment graphs, high availability

Production Impact and Lessons

Comparison to Alternative Systems

Ray Serve occupies a unique position in the model serving landscape. A comparison with alternative systems reveals its distinctive characteristics:

TensorFlow Serving [24] provides high performance for TensorFlow models with a C++ backend, battle-tested at Google, Uber, and Airbnb serving billions of predictions daily. However, it supports only TensorFlow models and offers limited flexibility for complex pipelines.

TorchServe provides similar benefits for PyTorch workloads, with default handlers for common tasks (object detection, text classification) and cloud-agnostic deployment. Like TensorFlow Serving, it is framework-specific.

NVIDIA Triton [75] achieves the highest raw throughput on GPU-intensive workloads through sophisticated optimizations (dynamic batching, model ensembles, TensorRT integration). Six years of development have produced a mature system achieving virtually identical performance to bare-metal on MLPerf benchmarks. However, it requires more expertise to configure optimally and offers less flexibility for arbitrary Python business logic.

KServe and **Seldon Core** provide Kubernetes-native serving with advanced features (canary deployments, A/B testing, serverless scaling) but require Kubernetes expertise and infrastructure.

Ray Serve's unique position is combining **framework-agnostic**, **Python-first**, **pipeline-native** design without requiring Kubernetes. It optimizes for multi-model composition and statistical multiplexing while enabling integration with Triton for workloads requiring maximum GPU performance.

Lessons Learned

Three years of Ray Serve development yielded lessons that inform future systems work:

Research informs but doesn't dictate production design. InferLine's abstractions (profiler, planner, controller) influenced Ray Serve's conceptual model but not its implementation. Production constraints—fault tolerance, backward compatibility, operational simplicity—often dominate research-driven design choices. The gap between research prototype and production system is substantial; bridging it requires engineering effort that exceeds the initial research implementation.

Sensible defaults matter more than configurability. Users rarely tune parameters. A system that works well with default settings serves 90% of users better than a system that achieves optimal performance with expert tuning. Configuration should be available but not required. This lesson echoes across systems research—from database query optimizers that work without manual tuning to network protocols that adapt without configuration.

Building for the 90% case. The temptation to handle every edge case is strong. Production experience taught prioritization: build for common cases first, address edge cases through documentation or workarounds rather than complexity. A simpler system that handles 90% of use cases is better than a complex system that handles 99%.

3.5 Reflections: Model Serving as a Discipline

Primitives That Matter

The journey from Clipper through InferLine to Ray Serve traces the maturation of model serving from research curiosity to production necessity. Across all three systems, certain primitives emerged as essential—and the algorithmic choices within each primitive reveal the tradeoffs between automation and control.

Batching remains the fundamental throughput optimization. Whether for CPU-based classical models or GPU-based deep learning, amortizing fixed costs across multiple requests is essential for efficiency. The evolution of batching algorithms illustrates a broader pattern:

Clipper's AIMD dynamically adapts batch size based on observed latency—a closed-loop control system. Ray Serve's explicit batching places configuration responsibility on users who understand their workloads. Neither is strictly superior; they represent different points on the automation–control tradeoff.

Routing determines which requests go to which replicas. Good routing improves load balance, reduces tail latency, and enables gradual rollouts. Ray Serve's adoption of the **Power of Two Choices** algorithm [78] demonstrates how theoretical insights translate into production systems—this algorithm provides near-optimal load distribution ($O(\log \log n)$ maximum queue length) with minimal coordination overhead, well-suited to Ray's decentralized architecture.

Scaling matches capacity to demand. The evolution from InferLine's sophisticated profiler-planner-controller to Ray Serve's queue-depth formula

```
num_replicas = ceil(ongoing_requests / target)
```

reflects a deliberate simplification. InferLine achieved 7.6× cost reduction through automated optimization; Ray Serve uses independent queue-based autoscaling as the production default because it is observable, robust, and often sufficient. The tradeoff: InferLine requires less user expertise but more system complexity; Ray Serve requires more user expertise but provides operational transparency.

Composition treats pipelines as first-class citizens. End-to-end optimization requires treating the pipeline—not individual models—as the unit of management. InferLine formalized this insight through SLO decomposition algorithms; Ray Serve operationalized it through the deployment graph API. The translation preserves the conceptual insight

(pipelines matter) while adapting the implementation (explicit DAGs vs. automated optimization).

SLOs Require End-to-End Thinking

A recurring theme is that SLOs apply end-to-end. Per-model optimization without end-to-end consideration leaves efficiency gains unrealized. This insight—central to InferLine and embedded in Ray Serve—distinguishes model serving from simpler request-processing systems.

The 7.6× cost reduction achieved by InferLine and the production cost reductions reported for Ray Serve both stem from the same insight: optimizing the whole pipeline yields dramatically better results than optimizing its parts independently. This is a general principle that applies beyond ML serving to any system with end-to-end constraints.

The analogy to web serving and databases is useful but limited. Web stacks optimize connection pools and HTTP routing around lightweight requests and stable cache keys; database pools amortize setup cost across queries but do not manage GPU-resident model state. Model serving adds GPU memory pressure, variable compute per request, batching, and accuracy–latency tradeoffs, which justify purpose-built serving abstractions rather than "Kubernetes plus GPU" alone. The next chapter shows where classical multiplexing stops: autoregressive generation and paged KV storage require abstractions that web and OLTP stacks were never designed to provide.

The Discipline Argument

The progression from Clipper through InferLine to Ray Serve demonstrates why inference serving constitutes a distinct systems discipline—not merely an application of existing distributed systems techniques.

Clipper established that ML inference has exploitable structure. The system introduced serving-specific abstractions—prediction caching, adaptive batching, model selection—that have no direct analogues in web serving or database systems. These abstractions exploit properties unique to ML inference: deterministic outputs for identical inputs (enabling caching), variable computational intensity across queries (enabling model selection), and GPU-amenable parallelism (enabling batching). General request-processing systems could not provide these optimizations because they cannot assume these properties.

InferLine demonstrated that this structure extends to composition. The SLO decomposition problem—allocating end-to-end latency budgets across heterogeneous pipeline stages—is specific to inference pipelines. Web request routing, database query optimization, and network flow control all face analogous problems, but the specific constraints (batching affects both latency and throughput, model variants trade accuracy for speed, GPU memory limits parallelism) require purpose-built algorithms. InferLine's 7.6× cost

reduction over baselines represents the gap between general-purpose approaches and inference-aware optimization.

Ray Serve carried these abstractions to production scale. The translation from research prototype to production system—involving thousands of engineering hours across fault tolerance, API design, operational tooling, and integration—demonstrates that the abstractions are robust enough to support real workloads. Public production reports, including the Anyscale-published Samsara Ray Serve deployment report, show that these abstractions can translate into substantial serving-cost reductions in practice [74].

This trajectory—vocabulary establishment, formal treatment, production validation—mirrors how other systems disciplines matured. Database systems evolved from file managers (vocabulary) through relational algebra (formalization) to distributed transaction protocols (production). Network stacks evolved from packet handlers through congestion control theory to software-defined networks. Inference serving is following the same arc. The systems in this chapter are not merely implementations; they are contributions to problem formulation. This is the mark of a maturing discipline.

The Serving Stack Is Established… But LLMs Change the Game

By 2022, model serving had matured into a discipline with established primitives, production systems, and best practices. Ray Serve, alongside TensorFlow Serving, Triton, and other systems, provided the infrastructure for production ML. The author's contributions—from open-source contribution to Clipper, through co-authorship on InferLine, to leading Ray Serve from zero to one—helped build this foundation.

Then LLMs arrived. The techniques in this chapter—adaptive batching, replica-based scaling, pipeline composition—assume predictable model execution. LLMs break this assumption:

- **Autoregressive generation** creates sequential dependencies that prevent straightforward parallelization
- **KV caches** grow unpredictably during generation, dominating GPU memory
- **Variable-length outputs** make resource allocation fundamentally harder

Recent systems have begun extending multiplexing concepts to LLMs. MuxServe [80] adopts spatial-temporal multiplexing via CUDA MPS with unified KV cache management. AlpaServe [61] combines model parallelism with statistical multiplexing. But these systems require fundamentally new abstractions—not just extensions of the primitives established in this chapter.

Chapter 4 addresses this discontinuity through vLLM—a serving system designed from first principles for the LLM era. The foundation established in this chapter remains relevant (batching, routing, composition) but must be reinterpreted for a fundamentally

different workload. PagedAttention, continuous batching, and memory-first design represent the next evolution of model serving—purpose-built abstractions for a new class of models.

This chapter traced the evolution of model composition from Clipper's prediction-serving abstraction through InferLine's SLO-aware pipeline optimization to Ray Serve's production-grade deployment framework. Together, these systems established the orchestration layer of the inference stack. Chapter 4 descends one level to address the memory management crisis that LLMs introduced—and the purpose-built abstractions that resolved it.

Chapter 4

vLLM — A Comprehensive System Study

4.1 Introduction to LLM Serving

The Memory Wall in Language Model Serving

Large language models break every assumption that prior inference serving systems were built upon. Classical models accept a fixed input, execute a single forward pass, and produce a fixed output—compute-bound, predictable, and amenable to the batching techniques that Clipper [25], Nexus [26], and Clockwork [27] perfected over half a decade. LLMs discard all three properties simultaneously: they generate output token by token, produce responses of unpredictable length, and are bound by memory capacity rather than compute throughput. This combination constitutes a discontinuity—not a gradual evolution—in the systems challenges of inference serving.

The discontinuity is best understood through an analogy to one of the oldest problems in operating systems. In the 1960s, multiprogramming required running multiple programs concurrently on machines with scarce physical memory. Virtual memory [3] introduced a level of indirection between logical addresses and physical storage, enabling on-demand paging, cross-process sharing through copy-on-write, and fragmentation elimination through non-contiguous allocation. The insight was architectural: memory management required a purpose-built abstraction that generic allocation could not provide. Six decades later, LLM serving faces a parallel problem. GPU memory is the scarce resource. Concurrent requests are the competing "programs." The KV cache—the attention state that accumulates during generation—is the memory that must be managed. The solution, as this chapter argues, is the same in spirit: a virtual memory system purpose-built for the new domain. That system is **PagedAttention**, the core innovation of vLLM.

This chapter presents vLLM, a serving system designed from first principles for the LLM workload. PagedAttention was invented by Kwon, Li, et al. and published at SOSP 2023 [30]; the author was not a co-author on that paper. The author joined vLLM after the

SOSP publication and led it as project co-lead since 2023, shepherding it from research prototype to widely adopted open-source LLM inference infrastructure (Section 4.6 documents the project's scale). This chapter treats PagedAttention as technical background essential to the systems contributions that follow: project growth, architectural evolution, production hardening, Jenga memory-management extension that the author co-authored, and a brief Pie follow-on contribution. The chapter proceeds from the characteristics that make LLMs different (Section 4.1), through PagedAttention's design (Section 4.2), vLLM's system architecture (Section 4.3), memory management extensions (Section 4.4), inference optimizations (Section 4.5), the open-source journey (Section 4.6), and empirical evaluation (Section 4.7).

Why LLMs Are Different

Autoregressive generation is the first and most consequential departure. A language model generates text one token at a time, sampling each token from a distribution conditioned on every preceding token—both the input prompt and all previously generated output. Token n cannot be produced until tokens 1 through $n-1$ exist. A 500-token response requires 500 sequential forward passes, each building on the last. This sequential dependency is architectural, not an implementation artifact. Parallelism exists within each forward pass—across attention heads, layers, and batch elements—but not across the tokens of a single response.

The contrast with classical inference is stark. An image classifier processes 32 images in a single forward pass with full intra-batch data parallelism. A speech recognition model transforms a spectrogram into text in one shot. Even diffusion networks produce fixed-resolution outputs through a predetermined number of denoising steps. LLMs alone combine generation with sequential dependency, limiting per-request parallelism.

The second departure is the **KV cache**. Transformer attention [50] computes interactions between all token positions in a sequence. At each generation step, the model attends to every preceding token—the full prompt plus all tokens generated so far. Recomputing attention from scratch at every step would be prohibitively expensive: cost grows quadratically with sequence length.

The standard solution is to cache the key (K) and value (V) projection tensors from each attention layer after they are first computed. Reusing stored K and V vectors reduces per-step attention cost from $O(n^2)$ to $O(n)$. This optimization is essential—without it, autoregressive generation would be infeasible for long sequences—but it introduces a new cost: memory. The KV cache grows linearly with sequence length and must persist for the entire lifetime of a request.

The numbers are concrete and large. For Llama-70B with 80 transformer layers, each KV cache entry stores a key and value vector per token per layer—approximately 640 bytes in FP16. A single request with a 16,384-token context accumulates roughly 640 bytes $\times$ 80 layers $\times$ 16,384 tokens $=$ 838 MB of KV cache. On an 80 GB A100 GPU, model weights consume 35–40 GB in FP16; the remainder must be shared among all

concurrent requests. The KV cache—not model weights, not activations, not gradients—becomes the binding constraint on serving capacity.

The third departure is **variable output length**. Unlike classifiers, detectors, and image generators—all of which produce fixed-size outputs—LLMs determine response length through the generation process itself. A request might generate 10 tokens or 2,000 tokens, and the serving system cannot know which until generation completes.

This unpredictability creates a resource allocation dilemma. Allocating memory for the maximum possible sequence length wastes capacity when most responses are short—and empirical distributions show that they usually are. Allocating for the expected length risks exhausting memory mid-generation, forcing the system to abort requests or trigger expensive recovery. The problem demands a mechanism that allocates memory incrementally, on demand, as the sequence grows.

The fourth departure is the inversion from **compute-bound to memory-bound** execution. Prior models—CNNs, RNNs—were typically compute-bound: throughput improved with more FLOPS, and GPU utilization was high because arithmetic kept compute units busy.

LLM decode inverts this relationship. Each decode step loads model weights and accumulated KV cache from GPU high-bandwidth memory (HBM) into compute units; the matrix multiplications complete faster than the memory transfers that feed them. **Arithmetic intensity**—the ratio of FLOPS to bytes of memory traffic—is low during decode. On an A100 with 2,039 GB/s HBM bandwidth and 312 TFLOPS of FP16 compute, the roofline model [81] places decode operations squarely in the memory-bandwidth-limited regime.

This inversion reshapes system design priorities. Optimizations that reduce memory traffic—quantization, caching, efficient memory layouts—matter more than those that reduce raw computation. A system designed around compute efficiency, as prior serving systems were, leaves the primary bottleneck unaddressed. LLM serving demands memory-first design.

Table 4.1: Characteristics of LLM inference compared to traditional deep learning inference.

Characteristic	Traditional DL (e.g., Image Classification)	LLM Inference
Generation model	Single forward pass	Autoregressive iteration (one pass per token)
Output size	Fixed (class probabilities, bounding boxes)	Variable (10–2,000+ tokens, unpredictable)
Primary bottleneck	Compute (FLOPS)	Memory bandwidth (decode phase)
Batch benefit	Linear throughput scaling	Sub-linear (diminishing returns per added request)
Memory scaling	Constant per request	O(sequence length) per request
Parallelism	Full data parallelism across samples	Limited to within-step parallelism
Typical GPU utilization	70–90%	20–40% (without memory optimization)
Arithmetic intensity	High (compute-bound regime)	Low (memory-bound in decode)

The Serving Landscape Before vLLM

Before vLLM's release in June 2023, the options for LLM serving ranged from trivially simple to performant but inflexible. None addressed the memory management problem that PagedAttention would later solve.

The most common approach was **HuggingFace Transformers**—loading a model and calling `model.generate()`. This offered simplicity and broad model coverage but no serving optimization: no dynamic batching, no memory management beyond PyTorch's default allocator, no concurrency beyond a single Python process. Throughput was orders of magnitude below hardware capability, sufficient only until traffic exceeded a handful of requests per minute.

FasterTransformer [40], NVIDIA's C++ inference library, occupied the opposite extreme. It provided hand-tuned CUDA kernels for every transformer operation, achieving excellent single-request latency through deep hardware optimization. But FasterTransformer used **static batching**: requests were grouped into fixed batches that proceeded in lockstep from start to finish. Every request in a batch waited for the slowest—the one generating the longest response—before any results could be returned. For LLM workloads with highly variable output length, this meant that a 10-token response sat idle while a 2,000-token response completed, wasting GPU cycles and inflating latency for short requests.

Text Generation Inference (TGI) [82], released by HuggingFace, advanced the state of the art by implementing continuous batching—allowing requests to enter and exit the batch dynamically. This eliminated the worst pathology of static batching. But early TGI versions allocated contiguous memory for each request's KV cache, reserving space for the maximum possible sequence length. The 60–80% memory waste went unaddressed: TGI could batch requests efficiently in time but not in memory. (TGI v3, released in late 2024, later adopted related KV-cache and scheduling ideas, providing external evidence for the abstraction this chapter describes.)

The landscape had solved the scheduling problem but not the memory problem. The gap came down to a single question: how should GPU memory be managed for KV caches that grow unpredictably, vary in lifetime, and must coexist in the hundreds on a single device?

Continuous Batching: The Orca Breakthrough

The intellectual foundation for modern LLM scheduling was laid by **Orca** (Yu et al., OSDI 2022) [29], which introduced a simple idea with transformative consequences: schedule at the granularity of a single iteration, not an entire request.

In a statically batched system, a group of requests enters the model together and proceeds through generation in lockstep. A batch containing requests that will generate 10, 50, and 200 tokens forces all three to wait for the 200-token response. The 10-token request idles for 190 passes; the 50-token request idles for 150. GPU cycles are consumed but no

useful tokens produced. The inefficiency compounds with output length variance—and LLM output lengths are highly variable.

Orca's insight was that each forward pass is a natural scheduling boundary. At every iteration, the scheduler decides which requests participate in the next pass. Completed requests exit immediately; queued requests join mid-batch without waiting for the current batch to drain. Orca termed this **iteration-level scheduling**, and the broader community adopted the term **continuous batching** [83].

Implementation required **selective batching**. Non-attention operations—linear projections, normalization, activations—accept concatenated inputs regardless of per-request sequence boundaries and are batched normally. Attention operations must respect per-request boundaries because each request attends to its own context. Orca handles attention per-request (or with careful padding) while batching everything else.

The throughput gains were enormous. On GPT-3 175B, Orca demonstrated **36.9× throughput improvement** over FasterTransformer at equivalent latency [29]. On workloads with high output-length variance—the common case for chat and instruction-following—continuous batching routinely achieved 2–10× higher throughput than static batching, entirely by eliminating idle time.

But Orca left a critical problem unsolved. Its memory management followed the same strategy as every prior system: pre-allocating a contiguous GPU memory block sized for the maximum sequence length per request. If the model supported 16,384 tokens, every request reserved that much—even if the typical response was 200 tokens. This created **three sources of waste**: reserved slots for tokens never generated, internal fragmentation from alignment rounding, and external fragmentation as requests of varying lifetimes left unusable gaps in the address space.

The combined effect was devastating. Effective memory utilization dropped to as low as **20.4%** [30], with 60–80% of reserved KV cache memory going unused. A GPU that could theoretically serve 40 requests simultaneously served only 10 because each hoarded memory it would never use.

Orca demonstrated continuous batching's value but left the memory problem—the defining constraint of LLM serving—unsolved. The KV cache remained monolithic, contiguously allocated, and reserved for the worst case. Solving this required an approach inspired not by prior ML systems but by operating systems. That approach is PagedAttention.

4.2 PagedAttention: The Core Innovation

PagedAttention is vLLM's foundational contribution: a virtual memory system for KV caches that eliminates fragmentation and enables flexible memory sharing. Kwon, Li, et al. [30] introduced PagedAttention at SOSP 2023, and the design has since become widely adopted in KV cache management across the LLM serving ecosystem. The author did not

co-author that paper; this section gives only the algorithmic background needed for the systems contributions that follow.

From Contiguous Reservation to Paged KV State

The problem PagedAttention solves is simple but severe: traditional LLM servers reserved a contiguous KV-cache region for each request, sized for the maximum context length rather than the sequence actually generated. On an 80 GB GPU serving a 13B-parameter model, weights and runtime buffers can leave roughly 40 GB for KV state. With a 16,384-token context and roughly 320 KB of KV data per token, worst-case reservation consumes about 5 GB per request, so only eight requests fit. Yet conversational workloads are usually far shorter; a few hundred generated tokens consume tens of megabytes, not gigabytes. The result is memory hoarding: Kwon et al. measured effective KV-cache utilization as low as **20.4%**, with **60–80%** of reserved memory wasted [30]. The waste comes from three places at once: slots reserved for tokens never produced, allocator rounding within each reservation, and external fragmentation as variable-length requests arrive and finish. General-purpose dynamic allocation avoids some over-reservation but still fragments GPU memory because CUDA allocators are not specialized for the regular, token-by-token growth pattern of KV caches.

PagedAttention replaces per-request contiguous reservation with fixed-size **KV blocks**, usually covering 16 tokens. Each request owns a **block table** mapping logical token ranges to physical blocks in GPU memory; the logical KV cache remains a contiguous sequence, but its physical storage can be scattered across the global block pool. Blocks are allocated on demand as prompts are processed and new tokens are generated, and they return to the free pool immediately when a request completes. Because every block has the same size, any free block can satisfy any future allocation, eliminating external fragmentation. The only remaining waste is the unused tail of the final block: with 16-token blocks and a 500-token sequence, at most 15 token slots are unused, or roughly 3%. In practice, vLLM reaches **95–99% memory utilization**, turning memory savings directly into larger batches and higher concurrency.

Sharing and Kernel Support

Block tables also make sharing natural. When several sequences share a prefix—beam search, parallel sampling, speculative decoding, or repeated prompts—their block tables can point to the same physical KV blocks, protected by reference counts. If one sequence later diverges and needs to write into a shared block, vLLM applies copy-on-write: allocate one new block, copy the old contents, update that sequence's block table, and leave the rest of the prefix shared. For beam search, this reduces prefix storage from one copy per beam to one copy total until the beams diverge; Kwon et al. reported up to **55% total memory savings** on beam-search workloads [30].

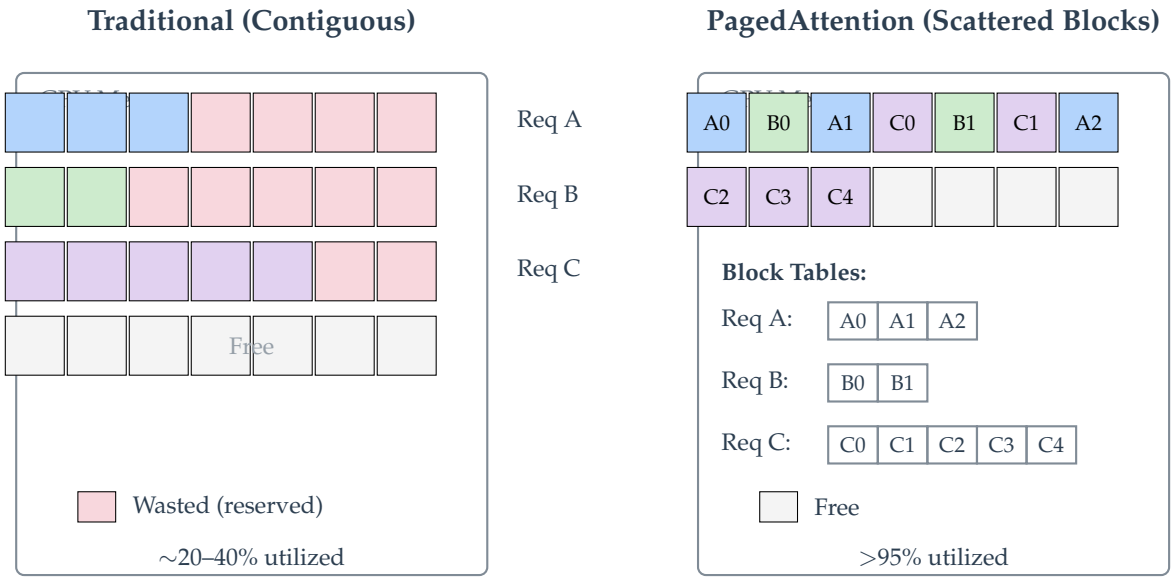

Figure 4.1: Traditional contiguous allocation versus PagedAttention's scattered block allocation. Left: each request reserves a contiguous region sized for maximum sequence length, leaving most memory unused (rose blocks). Right: PagedAttention allocates small blocks on demand, scattering them across GPU memory and achieving greater than 95% utilization. Block tables map each request's logical sequence to scattered physical locations.

The abstraction is not free, because attention kernels historically assume contiguous K and V tensors. During **prefill**, when the prompt is processed in one pass and KV data is still naturally contiguous, vLLM uses FlashAttention-style kernels [31, 62, 84]. During **decode**, each request attends over a long, potentially scattered history, so vLLM uses a PagedAttention kernel that walks the request's block table, loads each physical KV block, computes attention scores with a running softmax, and reduces partial results. The implementation preserves memory coalescing by assigning GPU thread blocks to contiguous physical KV blocks even though the logical sequence is non-contiguous. This indirection costs roughly **5–10%** per request relative to an ideal contiguous layout, but the memory savings allow much larger decode batches. The tradeoff is therefore system-positive: a small kernel penalty buys several-fold more concurrent requests, and those larger batches amortize scheduler and launch overheads.

PagedAttention should therefore be read as the enabling memory abstraction, not the whole system. It gives vLLM a compact representation of dynamic attention state: on-demand allocation for variable-length requests, reference-counted sharing for common prefixes, and a decode kernel capable of reading scattered blocks efficiently. The remainder of this chapter focuses on how vLLM builds a production serving stack around that abstraction.

4.3 vLLM System Architecture

Design Philosophy

PagedAttention is a memory management innovation; vLLM is the complete serving system built around it. A novel algorithm without a production-grade system remains a research prototype. vLLM transforms PagedAttention from idea into infrastructure: the engine orchestrates inference, the scheduler manages requests, workers execute on GPUs, and the API server handles external communication.

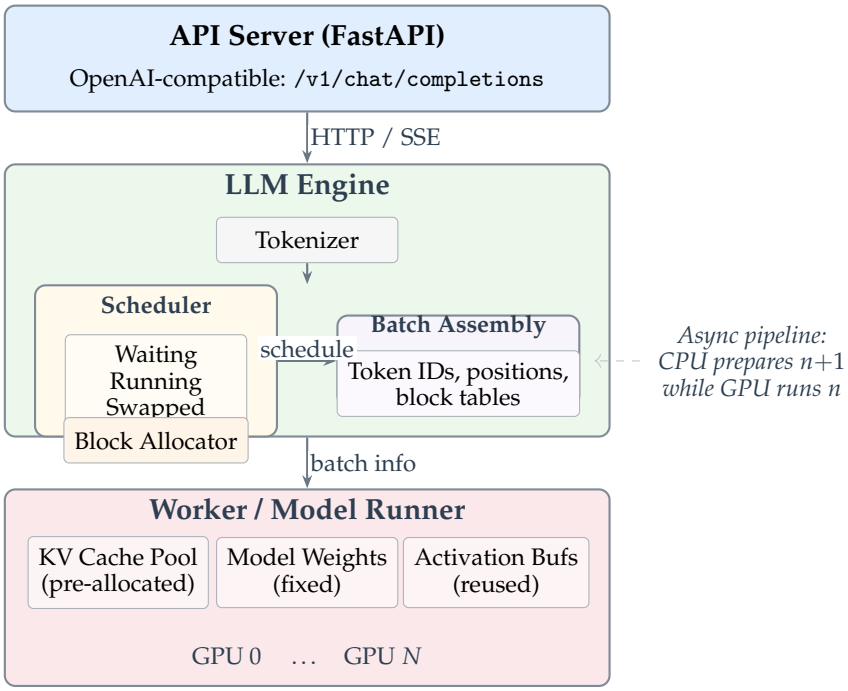

Figure 4.2: vLLM high-level architecture. The Engine sits at the center, coordinating the Scheduler (which manages request queues and memory allocation via the Block Allocator), the Worker/Model Runner (which executes inference on GPUs with pre-allocated KV cache, fixed model weights, and reused activation buffers), and the API Server (which handles external HTTP requests and streaming responses).

The vLLM Engine

A request arrives at the API server—via HTTP or direct Python call—and passes to the **LLMEngine**, which tokenizes the input into integer token IDs. The tokenized request enters the scheduler's waiting queue until sufficient KV cache blocks are available.

When resources become available, the scheduler promotes the request to its running queue and allocates initial blocks. The **prefill phase** processes all prompt tokens in a single forward pass, populating the KV cache—this is compute-intensive and resembles traditional inference. The request then transitions to the **decode phase**: each forward pass produces one new token and updates the KV cache, repeating until an end-of-sequence token or length limit. Generated tokens pass through a detokenizer and stream back to the client.

The `LLMEngine` maintains a request registry, a scheduler, handles to GPU workers, and a tokenizer. Its main loop calls `engine.step()` repeatedly: each step retrieves scheduling decisions, dispatches the batch to GPU workers, processes output tokens, and handles completions. This loop executes once per iteration, typically every 20–50 milliseconds.

Streaming responses are a first-class architectural concern, not a cosmetic feature. For chat applications, users expect tokens as they are generated. vLLM streams via Server-Sent Events (HTTP) and async generators (Python), with configurable flush policies. A ten-second response feels fast when tokens appear progressively but intolerably slow when delivered as a single block. This perception asymmetry makes streaming latency as important as total generation time.

The **async execution pipeline** overlaps CPU preparation with GPU execution. Without pipelining, the GPU would sit idle during scheduling, tokenization, and batch assembly. vLLM prepares iteration $n+1$ on the CPU while the GPU executes iteration n. Since CPU work (microseconds to low milliseconds) is almost always shorter than GPU work (tens of milliseconds), the pipeline hides CPU latency entirely.

The Scheduler

vLLM's default scheduler uses **First-Come-First-Served (FCFS)** ordering. FCFS is simple, fair, and avoids starvation. Alternative policies—priority-based, shortest-job-first—are available but rarely needed for general LLM serving.

The scheduler maintains three request queues whose transitions define a state machine governing every request's lifecycle.

The **Waiting** queue holds tokenized requests lacking sufficient blocks for prefill. The **Running** queue contains requests actively generating with allocated GPU blocks. The **Swapped** queue holds previously running requests whose KV cache was evicted to CPU memory under pressure. Each iteration, the scheduler decides which waiting requests to promote and whether running requests must be preempted.

Preemption handles memory pressure when running requests collectively demand more blocks than available. The scheduler selects victims—typically the most recently admitted, to minimize wasted computation—and applies one of two strategies.

Recompute discards the victim's KV cache entirely; when resources free up, the request restarts from its prompt. This wastes prior computation but requires no CPU-side state. **Swap** copies KV blocks to CPU memory via PCIe, preserving computation but requiring CPU memory and introducing swap-back latency. The choice is a classic time–

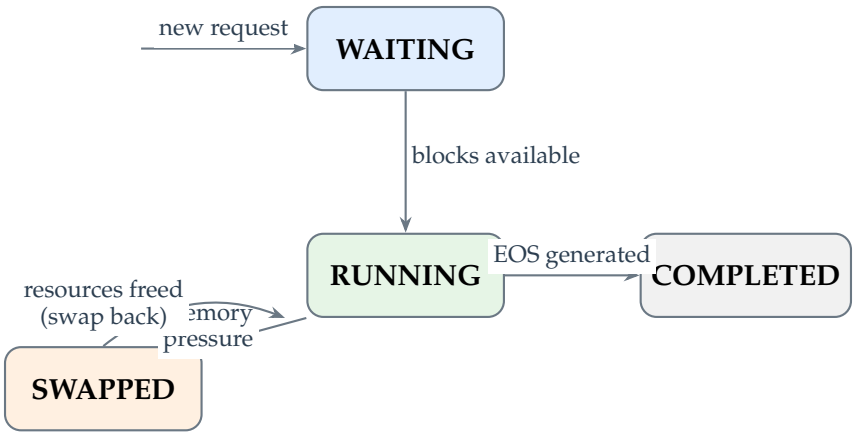

Figure 4.3: Scheduler state machine. Requests enter the WAITING queue on arrival. The scheduler promotes them to RUNNING when sufficient KV cache blocks are available. Memory pressure can demote RUNNING requests to SWAPPED, where their KV cache is saved to CPU memory. When resources free up, SWAPPED requests return to RUNNING and eventually reach COMPLETED.

space tradeoff: recompute suits short requests; swap suits long ones where recomputation cost is substantial.

Chunked prefill addresses head-of-line blocking from long prompts. A 10,000-token prefill may take 500 milliseconds, stalling all decode iterations and disrupting streaming output. Chunked prefill breaks long prefills into smaller chunks (typically 512 tokens), interleaving decode iterations between chunks. Total prefill time increases slightly, but decode latency for concurrent requests remains bounded—an easy tradeoff for interactive applications.

The Worker and Model Runner

The **Worker** manages GPU-side resources; the **Model Runner** handles kernel dispatch and tensor management.

At initialization, vLLM profiles available GPU memory and pre-allocates the entire remainder as a KV cache block pool—counterintuitive but critical for avoiding runtime allocation overhead and fragmentation. GPU memory is partitioned into three categories: **model weights** (fixed, loaded once), the **KV cache block pool** (fixed total size, blocks dynamically assigned by the scheduler), and **activation buffers** (transient intermediates reused across iterations).

Each iteration, the Model Runner gathers input tensors—token IDs, position IDs, and block tables—and dispatches CUDA kernels layer by layer: embedding lookup, attention (consulting block tables for KV access), feed-forward network, normalization, and output projection for token sampling.

For models exceeding single-GPU memory, vLLM supports tensor parallelism, pipeline parallelism, expert parallelism, attention data parallelism, and context/sequence parallelism. These methods allow a single instance of vLLM to handle arbitrarily large model across a small cluster of GPUs as a compute unit.

The OpenAI-Compatible API Server

Implementing an **OpenAI-compatible API** was vLLM's most impactful product decision. OpenAI's Chat Completions API (`/v1/chat/completions`) has become the de facto standard; thousands of applications are built against it. By matching request/response formats, streaming behavior, and parameter handling, vLLM enables applications to switch from OpenAI to self-hosted inference by changing a single configuration: the API endpoint URL.

This drop-in compatibility lowered adoption barriers and contributed directly to vLLM's growth during the author's tenure as Project Lead. The lesson: technical superiority is necessary but insufficient for adoption. Compatibility with existing workflows amplifies impact by orders of magnitude.

4.4 Memory Management Deep Dive

Memory is the fundamental resource in LLM serving. Model weights are fixed; the KV cache is dynamic, growing token by token and varying across requests. Section 4.2 introduced PagedAttention as the core abstraction. This section explores the full depth of memory management in vLLM: block allocation strategies, prefix caching, and the author's contributions extending PagedAttention for heterogeneous architectures (Jenga) and emerging hardware interconnects (Pie). Together, these techniques transform memory management into a rich optimization surface that directly determines throughput, latency, and cost.

Block size is a design decision with meaningful consequences. **Smaller blocks** (e.g., 8 tokens) reduce internal fragmentation—the final block wastes fewer slots—but increase block table overhead. **Larger blocks** (e.g., 64 tokens) reduce overhead but waste more memory in partial blocks: a 65-token sequence in 64-token blocks uses only 65 of 128 slots.

vLLM defaults to **16-token blocks**, a balance that keeps internal fragmentation under 3% for typical workloads (where average sequence lengths are hundreds of tokens) while maintaining manageable block table sizes. The optimal block size depends on workload characteristics: high-variance sequence lengths—where some requests generate 50 tokens and others generate 2,000—favor smaller blocks that minimize worst-case waste. Consistent sequence lengths—batch processing of summaries at a fixed 200-token cap, for instance—favor larger blocks that reduce per-block overhead.

Many production requests share common prefixes: chatbot system prompts, RAG context documents, few-shot exemplars. Without optimization, each request independently computes KV cache for these shared tokens—redundant work wasting both compute and memory.

Prefix caching eliminates this redundancy through automatic detection and reuse. The system hashes token prefixes at block granularity, forming a hash chain (each block's hash incorporates the preceding block's hash) that uniquely identifies any prefix. New requests walk their token sequence block by block; matching blocks are reused directly, skipping prefill for all cached tokens.

The impact depends on the ratio of shared prefix to unique suffix. A chatbot with a 1,000-token system prompt eliminates that prefill cost for every subsequent request. For RAG workloads referencing the same retrieved documents, prefix caching can eliminate much of the repeated per-request prefill computation. Cache eviction uses a LRU strategy, that the block least recently touched will be evicted.

The **V1 architecture**—vLLM's major rewrite—achieves **zero-overhead prefix caching** by integrating the mechanism directly into the block allocator. Every allocation checks the hash table as a matter of course, adding no measurable latency versus the non-cached path.

Jenga: PagedAttention for Hybrid LLMs

The author co-authored Jenga with Chen Zhang and others published at SOSP 2025 [85]. Jenga addresses a limitation that emerged as architectures diversified beyond vanilla transformers: PagedAttention assumes uniform attention across all layers, but modern models violate this assumption.

Mistral [86] uses **sliding window attention**. Llama 2 and successors employ **grouped-query attention (GQA)** [87], sharing KV heads across query groups. Hybrid models like Jamba interleave transformer layers with **state-space model (SSM)** layers. Each choice produces layers with different embedding sizes, access patterns, and optimal caching strategies. Uniform allocation either wastes memory on layers that need less or under-provisions layers that need more.

Jenga solves this with a **two-level memory allocator** that accommodates heterogeneous per-layer requirements while maintaining the benefits of PagedAttention's paged approach.

At level one, Jenga allocates **super-blocks** sized to the LCM of all layer embedding sizes, ensuring every layer's per-token embedding fits evenly with zero internal fragmentation. At level two, each super-block is subdivided into **per-layer regions**: GQA layers receive compact allocations reflecting their reduced footprint, full-attention layers receive proportionally larger allocations, and sliding-window layers receive allocation for only the tokens within their window.

This subdivision enables **per-layer caching policies**. For a sliding-window layer with window size w, tokens older than position w are unreachable and can be evicted. Full-

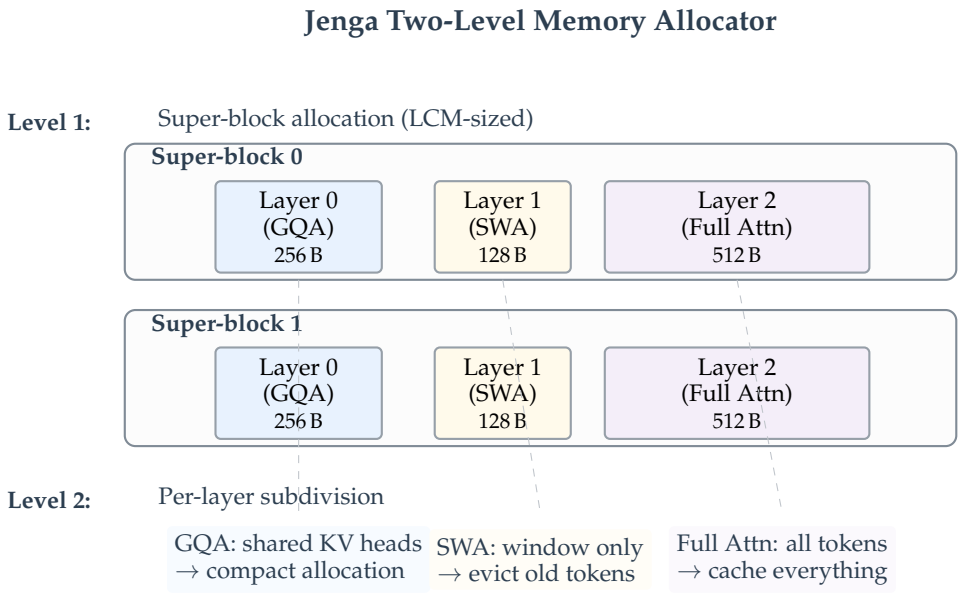

Figure 4.4: Jenga's two-level memory allocator. Level 1 allocates super-blocks sized to the least common multiple (LCM) of all layer embedding sizes, ensuring zero internal fragmentation at the super-block level. Level 2 subdivides each super-block into per-layer regions optimized for each layer's specific memory requirements and access patterns—GQA layers receive compact allocations, sliding-window layers evict tokens outside the window, and full-attention layers cache all tokens.

attention layers cache everything. No memory is spent storing state the model will never read.

Jenga also introduces **flexible block tables** that are optimized per layer rather than uniform across the model. A sliding-window layer's block table need only track blocks within the window; a full-attention layer's block table tracks all blocks. This reduces metadata overhead for layers with bounded access patterns.

On hybrid models, Jenga improves memory utilization by **up to 79.6%** over standard PagedAttention, achieving **up to 83% GPU memory utilization** and **up to 2.16× throughput improvement** (1.46× average). Implemented on vLLM and evaluated across diverse models, datasets, and GPU configurations, Jenga's benefits generalize across the heterogeneous model landscape.

Jenga demonstrates PagedAttention's extensibility. The original abstraction assumed uniform layers—reasonable for vanilla transformers but a liability as architectures diversified. Jenga provides evidence that the paging abstraction remains useful under architectural heterogeneity; what needed updating was the allocator's awareness of per-layer differences. This mirrors OS memory management: virtual memory pages have remained stable for decades while page tables, TLB hierarchies, and allocation policies evolved.

Good abstractions survive by adapting implementation, not interface.

Pie: NVLink-Aware Memory Extensions

The author also contributed to Pie, a follow-on memory-management effort that explores how PagedAttention-style block tables can extend beyond GPU-resident KV cache when hardware provides a much faster CPU–GPU interconnect. On Grace Hopper-class systems, NVLink-C2C changes CPU memory from cold swap space into a plausible secondary tier for selected KV blocks. The important dissertation point is architectural rather than a standalone performance claim: the same logical-to-physical indirection that made PagedAttention effective on GPU memory also provides a natural hook for tiered placement, prefetch, and eviction policies as hardware memory hierarchies evolve.

4.5 Inference Optimizations

The Optimization Landscape

By 2026, LLM serving optimization had shifted from one-off tricks to **bottleneck routing**. PagedAttention makes high concurrency possible; the remaining levers target different bottlenecks: speculative decoding reduces inter-token latency, quantization reduces memory traffic and footprint, attention backends exploit hardware-specific kernels, distributed execution fits larger models or scales throughput, and graph/fusion work removes CPU and launch overheads [88]. The practical question is no longer "which optimization is fastest?" but "which resource is currently limiting this workload?"

Optimization Decision Guide

The operating rule is simple: diagnose first, then change one primary lever. Long prompts with poor time-to-first-token (TTFT) call for chunked prefill, prefix caching, disaggregated prefill/decode, or faster prefill kernels. High inter-token latency (ITL) at low or moderate load calls for speculative decoding. Memory pressure calls for weight or KV-cache quantization. Models that do not fit on one device require tensor, pipeline, expert, context, or data parallelism. Small-batch decode overhead calls for CUDA/HIP graphs, fused kernels, and scheduler/IPC improvements. Each step must be validated on percentile TTFT, percentile ITL, and goodput, because the interactions in §4.5 often dominate isolated benchmark numbers.

Speculative Decoding

Speculative decoding accelerates decode by letting a cheap proposer draft several tokens and asking the target model to verify them in one pass [32, 63]. The method is

distribution-preserving under rejection sampling, but it is not free: the scheduler must run proposer and verifier work, maintain additional KV state, and roll back rejected draft tokens. As of 2026, vLLM supports draft-model speculation, n-gram and suffix proposers, EAGLE-family learned speculators, MTP, PARD, and MLP speculators [89, 90]. The frontier is reducing proposer overhead itself: P-EAGLE parallelizes EAGLE-style drafting so multiple draft tokens are produced in one pass, improving over vanilla EAGLE-3 on B200-class hardware [91]. The rule of thumb is unchanged: speculation helps latency-focused, memory-bound serving at low-to-moderate QPS; near saturation, the draft work often consumes capacity that should have served queued requests.

Quantization Support

Quantization reduces weight, activation, or KV-cache precision to lower memory footprint and bandwidth pressure. By 2026, vLLM supports a broad menu: FP8 W8A8 on hardware with native FP8 acceleration, FP8 weight-only paths on older GPUs, INT8/INT4 weight formats such as AWQ and GPTQ, compressed-tensors, TorchAO/-ModelOpt flows, and FP8 KV-cache quantization [92, 93, 94, 88]. The most important update is KV-cache quantization. For long-context serving, KV state can dominate memory; vLLM's FP8 KV-cache path supports per-tensor and per-attention-head scaling, and recent validation shows near-baseline long-context accuracy on supported Hopper and Blackwell paths while substantially reducing decode memory traffic [95]. Weight-only INT4 is best understood as a capacity optimization: it may fit a larger model or leave more space for KV cache, but dequantization and mixed kernel paths can erase throughput gains when the workload is compute-bound.

Attention Backends and Kernel Fusion

Attention backends specialize the hottest kernel in transformer inference. FlashAttention avoids materializing the $n \times n$ attention matrix through IO-aware tiling [31]; FlashAttention-2 improves parallelism [62]; FlashAttention-3 exploits Hopper asynchrony and FP8 support [84]; FlashInfer adds inference-oriented attention kernels and serving-friendly customization [96]. As of 2026, vLLM's backend matrix includes FlashAttention versions 2–4, FlashInfer/TRT-LLM paths, FlashMLA, Triton, ROCm-specific backends, and quantized KV-cache variants with feature support varying by GPU generation, head size, block size, and attention type [97]. The system implication is phase-specific: prefill prefers contiguous, compute-dense tiled attention; decode prefers PagedAttention-compatible kernels that read scattered KV blocks efficiently. Beyond attention, vLLM uses fused normalization, RoPE, sampling, custom MoE/GEMM kernels, CUDA/HIP graphs, and torch compilation to reduce launch overhead and CPU involvement on small decode steps.

Distributed Execution

Distributed execution now covers more than tensor parallelism. Tensor parallelism shards layer weights and uses all-reduce; pipeline parallelism assigns layer ranges to devices; data parallelism replicates model instances for throughput; expert parallelism balances MoE experts; context/decode-context parallelism splits long-context attention work [88]. The right choice depends on whether the goal is fitting the model, increasing throughput, balancing MoE load, or reducing long-context latency. Disaggregated prefill/decode adds a serving-level split: prefill and decode run in separate instances so operators can tune TTFT and ITL independently and control tail ITL, though vLLM's own documentation emphasizes that disaggregation is primarily a latency-isolation mechanism, not a throughput multiplier [33, 98, 99].

Interaction Effects and When Optimizations Backfire

The hard part of production serving is that optimizations do not compose linearly. **Speculation versus load** is the canonical example: at low load, otherwise idle compute verifies multiple draft tokens per target pass; at high load, the same draft work steals capacity from queued users. **Speculation versus quantization** is subtler: FP8 or INT4 target paths can change logits enough to lower draft acceptance, and heterogeneous draft/target precisions increase kernel switching and graph-capture complexity. **Quantization versus attention backend** also matters; FP8 KV cache is attractive only on backend/hardware combinations whose attention kernels actually operate efficiently in the quantized domain.

Parallelism versus latency creates another set of reversals. Tensor parallelism may be mandatory for fitting a model, but all-reduce overhead is paid every layer; for small interactive decode batches, fewer GPUs can beat more GPUs. Pipeline parallelism can fit larger models but introduces bubbles and complicates preemption. Expert parallelism improves MoE capacity only if routing is balanced; skewed expert popularity creates stragglers. Disaggregated prefill/decode reduces ITL tail interference, but it introduces KV transfer paths and operational placement decisions that can dominate if the prefill/decode split is poorly matched to traffic.

Caching versus memory pressure is similarly non-monotonic. Prefix caching reduces TTFT when prefixes repeat, but cached blocks compete with active requests for KV memory; under pressure, aggressive cache retention can increase preemption or admission delay. Chunked prefill protects decode latency, yet too-small chunks add scheduling overhead and too-large chunks recreate head-of-line blocking. Graph capture and fused kernels reduce launch overhead, but dynamic shapes from LoRA, structured outputs, multimodal inputs, or variable speculation depth can force graph breaks.

These interactions explain the evaluation strategy in §4.7: isolate one primary variable, report repeated runs and percentiles, and treat "best" as workload-specific. For a production operator, the mature 2026 lesson is less glamorous than any single optimiza-

tion: measure the bottleneck, apply the smallest lever, then remeasure after interactions have had a chance to appear.

4.6 The Open Source Journey

From Research Artifact to Infrastructure

The preceding sections documented vLLM's technical contributions: PagedAttention's virtual memory abstraction, the system architecture that operationalizes it, memory management extensions for emerging hardware, and the optimization stack that pushes performance toward hardware limits. These contributions are necessary but not sufficient to explain vLLM's impact. The best algorithm has no impact if nobody can use it. Systems research is littered with technically superior projects that never achieved adoption because they remained research prototypes: difficult to install, impossible to extend, abandoned when their creators graduated.

vLLM's trajectory from a research paper to widely adopted open-source LLM inference infrastructure is itself a systems contribution—one that required engineering judgment, community cultivation, and sustained technical leadership over two years. This section documents that journey, with emphasis on the author's role as one of the project lead guiding the project through a period of exponential growth in both usage and complexity.

From Paper to Project

vLLM launched in June 2023 alongside the PagedAttention paper [100]. The timing proved consequential. ChatGPT's late-2022 demonstration had created urgent demand for efficient serving infrastructure, and the gap between "model that works in a notebook" and "model that serves production traffic" was the primary bottleneck. HuggingFace Transformers' naive `model.generate()` was too slow, NVIDIA's FasterTransformer required deep CUDA expertise, and HuggingFace's Text Generation Inference—while implementing continuous batching—left memory fragmentation unsolved.

Three factors drove rapid adoption. First, **performance**: the 2–4× throughput improvement was immediately measurable—practitioners could run a benchmark in minutes and see the difference. Second, **simplicity**: `pip install vllm` followed by a few lines of Python was sufficient to serve any HuggingFace-hosted model, enabling bottom-up adoption by individual developers. Third, **API compatibility**: vLLM's OpenAI-compatible API meant applications could switch to self-hosted vLLM by changing a single endpoint URL—no code changes, no format adaptations.

Within months, vLLM became a common choice for open-source LLM deployment. Initial adoption brought external contributors—users submitted bug fixes, organizations added model support, hardware vendors contributed backends. This organic growth

provided evidence of the project's value but created an engineering challenge: managing pull requests of varying quality, reviewing performance-critical code, and maintaining coherence as the codebase expanded rapidly. The transition from research project to community-driven infrastructure required new processes, new communication channels, and dedicated leadership.

The Author's Role as Project Lead

The author served as Project Lead for vLLM from 2023 to 2025, joining after the SOSP 2023 paper was published—at the inflection point when the project had attracted its first outside contributors. The PagedAttention algorithm and initial implementation were complete; what the project needed was sustained technical leadership to transform a research artifact into production infrastructure. This transition demanded skills fundamentally different from writing a paper or implementing an algorithm.

Beyond shepherding technical projects, the author helps scale the project to tens of thousands of GitHub stars and thousands of contributors through release management, contributor onboarding, and ecosystem partnerships with NVIDIA, AMD, Intel, and Red Hat.

The Project Lead role encompassed three dimensions. As **product manager**, the author defined the roadmap, prioritized features, and made architectural decisions constraining or enabling future development. What should vLLM become—a narrowly optimized engine, or a broadly compatible platform? Which features deserved investment: multi-modal support, speculative decoding, quantization backends, hardware portability? When did complexity justify the capability it unlocked, and when did it threaten the simplicity driving adoption? These questions required judgment, not algorithms.

As **project manager**, the author coordinated releases, managed contributors, and maintained quality under exponential growth. Shipping reliable releases while hundreds of contributors submitted changes across CUDA kernels, Python scheduling logic, and HTTP servers demanded rigorous review, testing, and release validation. The author established workflows balancing velocity (users demanded new features weekly) against stability (production deployments could not tolerate regressions).

As **community leader**, the author built governance structures, communicated project direction transparently, and grew the ecosystem of investing organizations. Open-source projects succeed or fail on community health. The author organized bi-monthly meetups bringing together contributors from NVIDIA, AMD, Intel, Meta, Google, IBM, AWS, Red Hat, and dozens of other organizations, and established communication channels and recognition structures for significant contributions.

Three strategic decisions shaped vLLM's trajectory. The first: **model addition as simple as possible**. Clean interfaces enabled contributors to add model support with minimal coupling to the scheduler, memory manager, or kernel layer. This compounded: each new model attracted users, some of whom became contributors, further expanding cov-

erage. By 2025, vLLM supported over 100 model architectures across text, multi-modal, sparse, embedding, and classification workloads [101].

The second: **component isolation**. Quantization backends, vision encoders, attention kernels, and hardware-specific code were isolated behind well-defined interfaces. A quantization contributor need not understand preemption logic; a hardware vendor need not modify the attention algorithm. This modularity enabled parallel development—multiple teams working on different components without coordination overhead—and was essential as the contributor base grew from dozens to thousands.

The third: **multi-hardware support from the beginning**. Abstracting over NVIDIA, AMD, Intel, and Google TPU hardware differences imposed real engineering cost—code that could assume CUDA primitives became more complex to accommodate ROCm, oneAPI, and XLA. But hardware vendors became invested in vLLM's success: AMD contributed ROCm and MI300X optimizations, Intel contributed Gaudi and GPU backends, AWS contributed Neuron support. Each platform brought its own user base and contributor community, expanding reach far beyond a single-vendor strategy.

The growth metrics quantify this trajectory without needing the removed growth figure. The PyTorch Foundation reported more than 46,500 GitHub stars and more than 1,000 contributors when vLLM became a hosted project in May 2025 [101]. Public GitHub metadata showed roughly 79,000 stars and 2,590 contributors by May 2026 [102]. The exact numbers change weekly, but the order of magnitude matters: a research contribution had become shared infrastructure maintained by a broad ML community.

Evolution of the System

vLLM's architecture evolved through distinct phases, each driven by user demand and implemented against the modular interfaces established in the initial design. **Early releases** (second half of 2023) established the core—PagedAttention memory management, centralized scheduler, continuous batching, and the OpenAI-compatible API server—showing that PagedAttention's throughput improvements translated from benchmarks to production workloads. Subsequent releases through mid-2024 expanded along multiple axes: multi-modal support for vision-language models, speculative decoding integration (Section 4.5), and new quantization backends (FP8, AWQ, GPTQ).

Mid-2024 expansion accelerated the project's transition from research prototype to production platform. FP8 quantization, pipeline parallelism, and state-space model support (Jamba) broadened the system's reach. The Llama 3.1 launch in July 2024 was a watershed moment: Meta had partnered with vLLM for pre-release testing, and 8 of 10 official Meta inference partners ran vLLM [103]. The same month, vLLM's incubation into the **LF AI & Data Foundation** [104] formalized open governance—a deliberate strategic decision, initiated during the author's tenure, to ensure long-term sustainability beyond any individual contributor or company. By the end of 2024, vLLM reported **2.7× throughput** over its mid-year baseline through architectural refinements—ZMQ IPC (eliminating Python GIL contention) and multi-step scheduling [105]—and Red Hat's acquisition of

Neural Magic, a leading vLLM contributor, signaled growing enterprise investment in the project [106]. Production deployments expanded to include Roblox (4 billion tokens per week across ∼1,000 GPUs [107]) alongside Amazon and LinkedIn (§4.6), and the project supported approximately 100 model architectures across NVIDIA, AMD, AWS Inferentia, Google TPU, and Intel hardware. This trajectory made the V1 architectural investment both necessary and feasible.

The most significant architectural change was the **V1 rewrite**. Exponential growth had created codebase complexity threatening maintainability: clean abstractions had accumulated special cases for different model families and hardware platforms, test coverage lagged, and onboarding required understanding undocumented implicit dependencies.

V1 addressed these concerns through three structural changes: cleaner separation between scheduler, worker, and API server with explicit interfaces; a **plugin system** for runtime-registered backends; and improved testing infrastructure with automated correctness validation and performance regression detection. The rewrite reported up to a 1.7× speedup—evidence that maintainability and performance can be complementary—along with **zero-overhead prefix caching** by eliminating unnecessary checks in the hot loop [108].

The V1 rollout followed a phased strategy. On **January 27, 2025**, v0.7.0 shipped V1 as an **alpha** behind an opt-in flag [108]. On **March 18, 2025**, v0.8.0 promoted V1 to the **default engine** [109], with V0 remaining available for unported features. A May 2025 RFC formalized V0 deprecation, designating v0.9.0 as the last dual-engine version [110]. By v0.11.0 (October 4, 2025), V0 was fully removed—a ten-month migration from alpha to sole engine that balanced urgency against production stability.

Consolidation, Governance, and Maturity

The same process discipline that enabled model support also made the V1 migration feasible. Each new architecture required architecture analysis, implementation against standard interfaces, correctness validation against reference implementations, performance benchmarking, and documentation. That workflow converted external model pressure into repeatable engineering practice rather than one-off patches.

The 2025–2026 period compressed three forms of maturation into a single arc. DeepSeek-V3/R1 and later MLA-heavy MoE models stressed vLLM's attention kernels, expert-parallel execution, and scheduler at production scale; PyTorch Foundation hosting gave the project a neutral governance home [101]; and the V1 migration turned accumulated production lessons into a cleaner default engine [109]. Commercialization through Inferact in January 2026 further indicated that vLLM had become infrastructure worth supporting as both an open-source project and a managed serving stack [111].

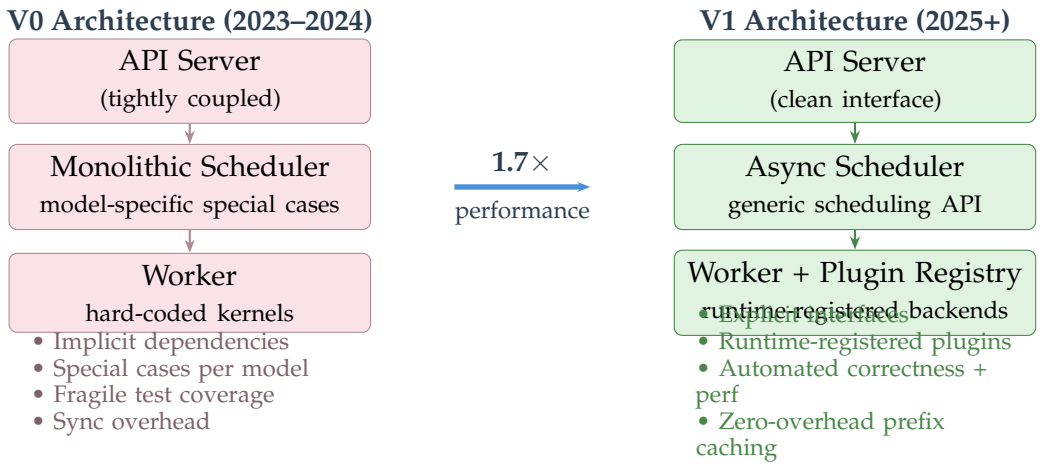

Figure 4.5: V0 vs. V1 architecture comparison. The V0 architecture accumulated model-specific special cases and implicit dependencies over 18 months of rapid development. The V1 rewrite introduced explicit interfaces, a plugin registry for runtime-selectable backends, and automated testing infrastructure. The result was both cleaner and faster: a 1.7× performance improvement alongside substantially reduced maintenance burden.

Model Architecture Co-Evolution

The preceding timeline documents *when* vLLM adapted; this subsection examines *what forced each adaptation*. Chapter 2 introduced the general characteristics of LLM inference. Here the focus is specific: how concrete shifts in model architecture demanded concrete changes in vLLM's memory manager, scheduler, and execution engine. The pattern is consistent—each new architectural family altered either the memory footprint per token, the compute-to-memory ratio, or both, and vLLM's modular abstractions absorbed each shift without rewriting the scheduler or API layer.

Attention mechanism evolution. The original PagedAttention design targeted dense multi-head attention (MHA), where every attention head maintains its own key and value projections. Grouped-query attention (GQA) [87], adopted by Llama-2 and most subsequent open-weight models, shares KV heads across groups of query heads—Llama-2 70B uses 8 KV heads versus 64 query heads, reducing per-token KV cache size by approximately 8×. Multi-query attention (MQA) [112] pushes this further to a single KV head. For PagedAttention, smaller per-token KV footprints mean more sequences fit per memory block, shifting the optimization bottleneck from memory capacity toward memory bandwidth and compute utilization. vLLM adapted its block size calculations and memory profiling to account for GQA's reduced footprint, enabling substantially higher concurrency on identical hardware. Growingly, sliding window attention lets models attend to most recent tokens only and key-value tying reduces kv footprint by half. These opti-

mizations, when applied correctly during training time, drastically reduced the memory requirements per token.

Mixture-of-Experts (MoE). Mixtral 8×7B [113] introduced sparse expert routing to open-weight serving: only 2 of 8 experts activate per token, so FLOPs scale sub-linearly with total parameter count while weight memory remains high (all experts must be loaded). For vLLM, this created a new scheduling challenge—expert-parallel execution across GPUs required load-balancing logic absent from the dense-model scheduler, and the mismatch between high weight memory and low per-token compute altered the memory-compute tradeoff that PagedAttention was designed to optimize. DeepSeek-V2 and V3 pushed MoE further with fine-grained experts (160 experts, 8 active per token), demanding wide expert parallelism and specialized communication patterns that consumed months of engineering effort.

Hybrid attention-SSM architectures. Jamba [114]—52B parameters, with attention layers interleaved among Mamba SSM layers at roughly one attention layer per eight total layers—broke the assumption that every layer produces KV cache. Attention layers grow KV cache linearly with sequence length; SSM layers maintain fixed-size recurrent state independent of context length. This heterogeneity is precisely the motivation for Jenga's per-layer memory management (§4.4): a uniform block allocation strategy that treats every layer identically either wastes memory on SSM layers or under-provisions attention layers. Supporting Jamba anticipated Jenga's design thesis months before the SOSP 2025 publication.

Multi-modal models. LLaVA [115], Pixtral, and Qwen-VL prepend fixed-length vision encoder embeddings to text token sequences. The serving implications are twofold: image embeddings shared across multi-turn conversations are natural candidates for prefix caching, and heterogeneous token sizes (vision tokens are typically larger than text tokens) complicate memory planning. vLLM v0.5 added multi-modal support through a modular vision encoder interface that isolated image processing from the text generation pipeline, enabling prefix caching to operate across modalities without scheduler modifications.

Reasoning models. DeepSeek-R1 [116] and OpenAI's o1-style models produce thousands of reasoning tokens before the final answer—inverting the traditional workload profile from prefill-dominated to decode-dominated. KV cache pressure shifts from prompt processing to sustained generation, and the scheduler must prioritize long-running decode sequences that occupy memory for extended durations. This workload pattern stressed vLLM's preemption and eviction mechanisms in ways that conversational chat never did.

Multi-Latent Attention (MLA). DeepSeek-V2 and V3's compressed KV representation [117] reduces per-token KV cache by 10–20× compared to standard MHA through low-rank projection of keys and values into a compact latent space. Exploiting this compression requires specialized kernels—FlashMLA—that operate on the compressed representation directly rather than decompressing and then applying standard attention. The 2025 DeepSeek optimization effort required integrating FlashMLA kernels, adapting the

block manager for compressed block formats, and implementing data-parallel pipelining (DPP)—a collaboration spanning the core team, Red Hat, and NVIDIA contributors.

The common thread across these architectural shifts is that vLLM's layered design—model definitions, attention backends, block manager, scheduler, API server—allowed each adaptation to be localized. GQA required changes to memory profiling and block sizing, not the scheduler. MoE required new parallelism strategies, not changes to the attention backend. Hybrid architectures required per-layer memory policies (Jenga), not API changes. This modularity was not accidental; it was the direct consequence of the component isolation strategy described earlier in this section.

Hardware Generations and System Adaptation

GPU hardware co-evolved with model architectures during vLLM's development. Chapter 2 surveys GPU generations in detail; Chapter 5 examines their capabilities from the perspective of kernel-level GPU multiplexing. Here the key point is that each GPU generation shifted the optimization landscape, and vLLM adapted its memory profiling, kernel dispatch, and distributed execution accordingly.

HBM bandwidth grew approximately 9× from V100 to B200 (900 GB/s to ∼8 TB/s), while FP8 compute grew from nonexistent to nearly 2,000 TFLOPS—a widening gap between prefill (compute-bound) and decode (memory-bandwidth-bound) performance. This divergence is the hardware-level force behind disaggregated prefill/decode serving, chunked prefill, and speculative decoding. Exploiting Blackwell's NVLink-C2C interconnect—the hardware assumption underlying Pie's memory management (§4.4)—required CUDA 12.8+, custom PyTorch builds, and over 100 pull requests [118]. Non-NVIDIA accelerators (AMD ROCm, Google TPU, AWS Neuron, Intel oneAPI, IBM Spyre, Huawei Ascend) each brought their own contributor communities and constraints, requiring the clean abstraction boundaries described earlier. A serving system designed for one hardware generation's compute-to-bandwidth ratio cannot optimally exploit the next without architectural adaptation—and vLLM's hardware abstraction layer was designed precisely to accommodate this ongoing evolution.

Table 4.2: GPU generation impact on vLLM serving design. Each hardware generation introduced capabilities that demanded specific adaptations in vLLM's memory manager, scheduler, or execution engine. HBM bandwidth grew approximately 9× from V100 to B200, while FP8 compute grew from nonexistent to nearly 2,000 TFLOPS.

GPU	HBM Capacity	HBM Bandwidth	FP8/FP16 TFLOPS	NVLink BW	Key vLLM Adaptation
V100 (2017)	16/32 GB HBM2	900 GB/s	— / 125	300 GB/s (gen 2)	Memory efficiency validation
A100 (2020)	40/80 GB HBM2e	2,039 GB/s	— / 312	600 GB/s (gen 3)	Primary benchmark target; MIG
H100 (2023)	80 GB HBM3	3,350 GB/s	1,979 / 990	900 GB/s (gen 4)	FP8 quantization; TP across 8 GPUs
H200 (2024)	141 GB HBM3e	4,800 GB/s	1,979 / 990	900 GB/s (gen 4)	Higher concurrency via memory headroom
B200 (2025)	192 GB HBM3e	∼8,000 GB/s	∼2,250 / 1,125	1,800 GB/s (gen 5)	NVLink-C2C KV tier; 100+ PRs

Workload Diversity: From Chat to Agents

vLLM began with conversational chat traces such as ShareGPT, but by 2025 it had to serve a broader set of workload shapes: chat, code generation, long-context RAG, reasoning, offline batch inference, multi-modal inputs, structured outputs, and agentic loops. The point is not that each workload needs its own serving system; it is that a shared serving system needs abstractions general enough to absorb changing prompt lengths, output lengths, cache locality, and latency objectives. Table 4.3 summarizes the resulting stress patterns.

Table 4.3: Workload taxonomy and vLLM feature dependencies. Each workload type stresses different subsystems and demands different optimization strategies. The diversity of these workloads is the strongest empirical argument that inference serving requires purpose-built systems with general abstractions.

Workload	Input Tokens	Output Tokens	Dominant Phase	Key Metric	Critical vLLM Feature	Bottleneck
Conversational chat	200–500	100–500	Balanced	ITL	Streaming, prefix caching	Memory BW
Code generation	1K–5K	200–1K	Prefill	TTFT + ITL	Parallel sampling, spec. decode	Compute
Long-context RAG	10K–100K	100–500	Prefill	TTFT	Chunked prefill, prefix cache	Compute + memory
Reasoning / CoT	200–1K	1K–10K+	Decode	Throughput	Preemption, max batching	KV cache capacity
Batch / offline	Variable	Variable	Both	Throughput	Max batching mode	GPU utilization
Multi-modal	500–2K (img)	100–500	Prefill (image)	TTFT	Vision encoder, prefix cache	Compute
Structured output	200–1K	50–500	Decode	TPOT	Constrained decoding	Decode overhead
Agentic / multi-step	Variable	Variable	Both (bursty)	TTFT	Prefix caching, KV persistence	Cache management

The taxonomy reinforces the abstraction argument: PagedAttention handles variable KV footprints, chunked prefill protects decode latency under long prompts, prefix caching exploits shared context, and pluggable backends localize model- and hardware-specific work. Those mechanisms do not make serving solved, but they give vLLM a stable surface for adapting as workloads shift.

Production Hardening

The distance between a system that works in benchmarks and one that runs reliably in production is measured in the unglamorous work of diagnosing failures, fixing edge cases, and preventing regressions. Production deployments—serving real traffic for days without restart—exposed issue categories invisible during development.

Memory leaks—a few kilobytes per request in an improperly cleaned metadata structure—eventually exhaust memory and crash the process. **Race conditions**—two requests simultaneously triggering prefix cache eviction, corrupting the free block list—produced intermittent, hard-to-reproduce failures. **Performance regressions** from seemingly innocent changes—a logging statement in a hot path, a garbage collection trigger—could degrade throughput by 10–20% without functional tests failing. **Hardware-specific bugs**—correct results on A100 but incorrect on H100 due to FP8 rounding differences—

required specific hardware for diagnosis. Each failure mode demanded a targeted fix and a regression test.

Large-scale production deployments—Amazon Rufus, LinkedIn, Roblox, and others described in Section 4.7—drove systematic reliability investment: health checks for load balancer routing; graceful degradation under memory pressure; Prometheus and Jaeger integration for observability; startup configuration validation catching incompatible quantization methods or insufficient memory; and automated testing across diverse workloads, hardware, and model families to prevent regressions. Production feedback flowed back as contributions: deployers contributed targeted optimizations that benefited the entire community.

The hardware ecosystem expanded dramatically: production-quality backends for AMD (ROCm), Intel (oneAPI), Google TPU (XLA), AWS Neuron, IBM Spyre, Huawei Ascend, and Blackwell. By 2025, over 15 full-time engineers across 6+ organizations contributed, with 20+ stakeholder organizations participating in governance. This breadth transformed vLLM from a research project into shared industry infrastructure, analogous to how Linux evolved from a personal project into collectively maintained OS infrastructure [21].

4.7 Evaluation

This evaluation asks whether the chapter's memory-first design claims survive contact with measurement. The evidence is organized around three questions: does paging eliminate KV-cache fragmentation, does the full engine sustain interactive throughput under load, and which optimizations help only under particular workload conditions? The goal is not an exhaustive benchmark suite, but a compact empirical argument for why purpose-built inference abstractions matter.

Controlled Experiments

All controlled experiments ran on a single NVIDIA H200 141 GB HBM3e GPU with exclusive access, using Llama-3.1-8B-Instruct (BF16). Each configuration ran for at most 5 minutes, with up to 60 seconds of warmup, at least 180 seconds of measurement, and at least 3 independent runs. The TGI comparison is historical: TGI v3 adopted related KV-cache and scheduling mechanisms, and by 2026 the repository was archived and marked maintenance-only [82, 119].

Experiment 1: PagedAttention Algorithm Effectiveness. A simulation replays 1,000 requests from three sequence-length distributions—Uniform $[128, 4096]$, Zipf ($\alpha = 1.0$, range $[32, 8192]$), and Bimodal (modes at 256 and 4096)—against static pre-allocation, power-of-two buddy allocation, and PagedAttention with 16-token blocks. PagedAttention reaches **99.5–99.6%** utilization because waste is bounded to the final partially

Table 4.4: Controlled evaluation summary. Each experiment isolates one system claim: paging improves capacity, load tests expose the saturation boundary, and workload-specific optimizations help only when their overhead matches the request mix. (1× H200 141 GB, Llama-3.1-8B-Instruct, BF16, 3 runs per configuration.)

Experiment	Question	Result
1. PA Algorithm	Does paging remove KV waste?	PA achieves **99.5–99.6%** util vs 18–26% static; **3.9–5.6**× more concurrent requests
2. Throughput Scaling	Where is the single-GPU capacity ceiling?	Near-linear to **13,400 tok/s**; saturation at QPS $\geq$ 200 (achieved $\sim$118 QPS)
3. Concurrency Scaling	Does streaming latency hold before saturation?	Stable ITL p99 (**14–31 ms**) up to QPS = 100; tail degradation at saturation
4. Chunked Prefill	Can long prefills coexist with decoding?	Chunk 512 bounds ITL from **218 ms to 62 ms** during 50K prefill; +15% TTFT cost
5. Speculative Decoding	When does draft-model overhead pay off?	**+4%** for open-ended chat; **−34%** for short-output summarization
6. Prefix Caching	How much does shared context reuse matter?	**10–16× TTFT reduction**: 534 ms→53 ms at 10K, 4,535 ms→281 ms at 100K

filled block, while static allocation reserves `max_model_len=8192` for every request and therefore uses only 18–26% of the available KV budget.

Table 4.5: Experiment 1 allocator comparison: the same 1,000-request trace is replayed against three allocation policies under a fixed 115 GB KV budget. The result explains the capacity mechanism behind PagedAttention: eliminating reservation waste translates directly into 3.9–5.6× more concurrent requests.

Distribution	Static util (%)	Buddy util (%)	PA util (%)	PA max concurrency	PA/Static ratio
Uniform	25.8	75.0	**99.6**	444	**3.9**×
Zipf	18.8	72.8	**99.5**	610	**5.3**×
Bimodal	17.6	66.9	**99.5**	649	**5.6**×

Experiments 2–3: Throughput, Latency, and Saturation. Offered load increases from 5 to 400 QPS in six steps under open-loop Poisson arrivals. Throughput scales near-linearly through 100 QPS, sustaining **10,618 tok/s** with ITL p99 of **31 ms**. At 200 QPS the system saturates: throughput plateaus at **13,451 tok/s**, achieved QPS remains near 118, and TTFT p99 exceeds **3 seconds**. The important distinction is that queueing pressure lands mostly in TTFT, while streaming ITL stays below 60 ms even at saturation.

Experiments 4–6: Workload-Dependent Optimizations. A single **50,000-token prompt** is injected while short requests decode, and ITL p99 is measured before and during the prefill under four configurations: chunking OFF, and chunk sizes 8,192, 2,048, and 512. Chunk size 512 bounds interference to **62 ms**, compared with **218 ms** without chunking, but increases long-prompt TTFT by 15%. A 1B draft model (Llama-3.2-1B) speculates 5 tokens ahead for the 8B target on ShareGPT ($\sim$220 output tokens) and synthetic summarization ($\sim$13 output tokens), with 150 prompts at concurrency 16. Speculation is even more conditional: it improves open-ended chat by **4%**, but regresses short-output summarization by **34%** because draft execution and verification dominate.

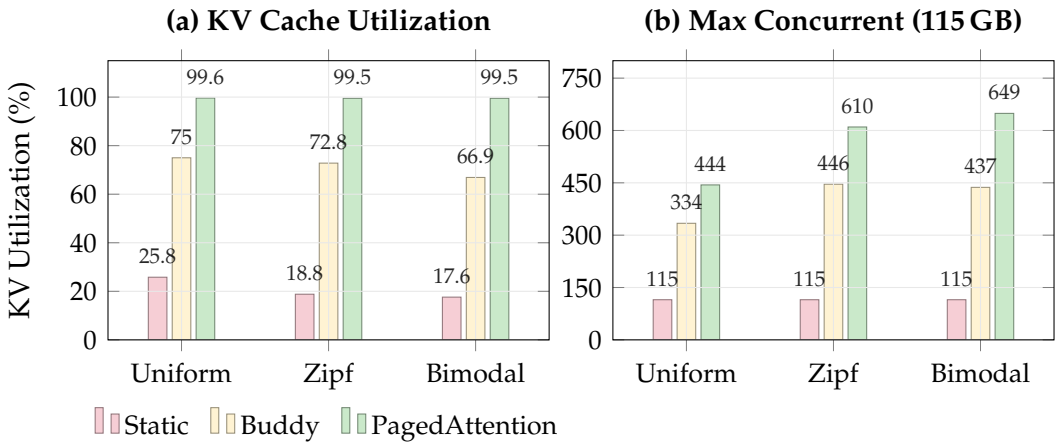

Figure 4.6: Experiment 1: PagedAttention allocator effectiveness under a fixed 115 GB KV budget. Panel (a) shows the mechanism: paging keeps KV utilization above 99% across Uniform, Zipf, and Bimodal sequence-length distributions, while static allocation wastes most reserved memory. Panel (b) shows why this matters for serving: the same memory budget admits 3.9–5.6× more concurrent requests than static allocation. (1,000 requests, 3 runs.)

Table 4.6: Experiments 2–3 latency under increasing open-loop load. Offered QPS creates pressure, achieved QPS reveals the capacity ceiling, and the TTFT/ITL split shows where overload appears: waiting for the first token grows into seconds after saturation, while decode smoothness degrades more slowly.

Offered QPS	TTFT p50 (ms)	TTFT p99 (ms)	ITL p50 (ms)	ITL p99 (ms)	Achieved QPS	Notes
5	40	59	12.7	17	4.9	Minimal queuing
20	41	56	12.9	17	18.9	Near-linear scaling
50	50	73	14.6	22	46.7	Moderate batching
100	88	158	17.2	31	92.9	Last stable regime
200	1,998	3,135	18.3	57	118.1	**Saturation**
400	2,155	3,196	18.1	58	117.0	Capacity wall

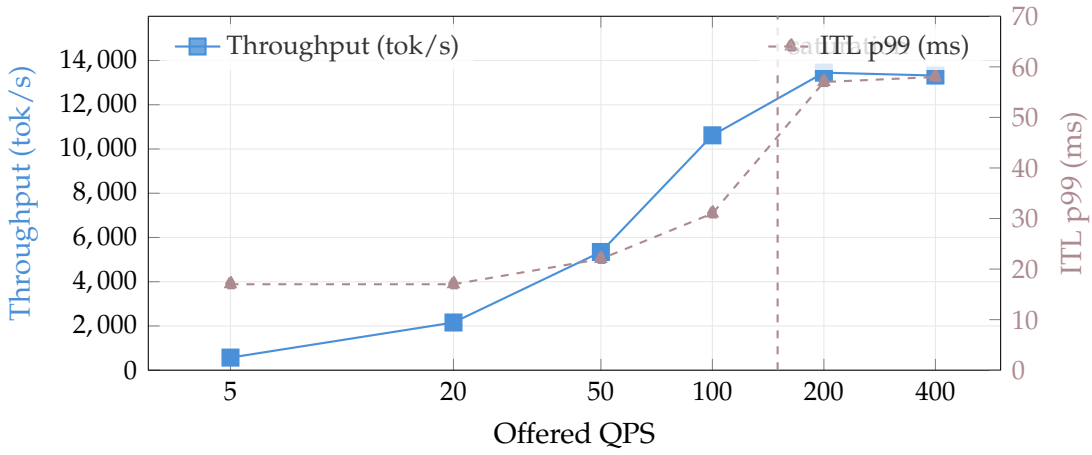

Figure 4.7: Experiments 2–3: throughput scaling under open-loop ShareGPT load. The blue line shows total token throughput rising nearly linearly until the single-H200 capacity wall near 13.4K tok/s; the red dashed line shows ITL p99 staying within interactive bounds until saturation. The figure explains why throughput and latency must be read together: after the dashed boundary, additional offered QPS mostly becomes queueing delay rather than useful work.

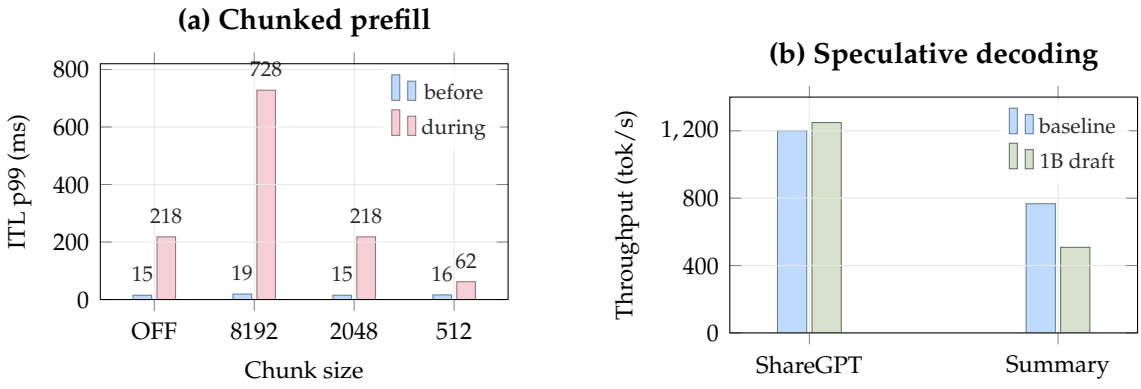

Figure 4.8: Experiments 4 and 5: optimization tradeoffs. Panel (a) shows why chunked prefill is a scheduler control rather than a universal win: 512-token chunks reduce decode interference during a 50K-token prefill from 218 ms to 62 ms, but at a 15% TTFT cost for the long request. Panel (b) shows why speculative decoding must be workload-aware: the 1B draft model slightly improves long-output ShareGPT throughput, but hurts short-output summarization because draft and verification overhead dominate.

Experiment 6: Prefix Caching. Prefix caching (`-enable-prefix-caching`) is tested at prefix lengths of 10,000, 50,000, and 100,000 tokens, comparing **cold** (no cache) versus **warm** (cache hit) TTFT across 3 runs. The effect is large and monotonic: warm-cache TTFT falls from **534 ms to 53 ms** at 10K tokens, from **2,001 ms to 150 ms** at 50K, and from **4,535 ms to 281 ms** at 100K. This is the most production-relevant optimization in the evaluation because many chat, RAG, and agentic workloads repeatedly serve shared system prompts or document prefixes.

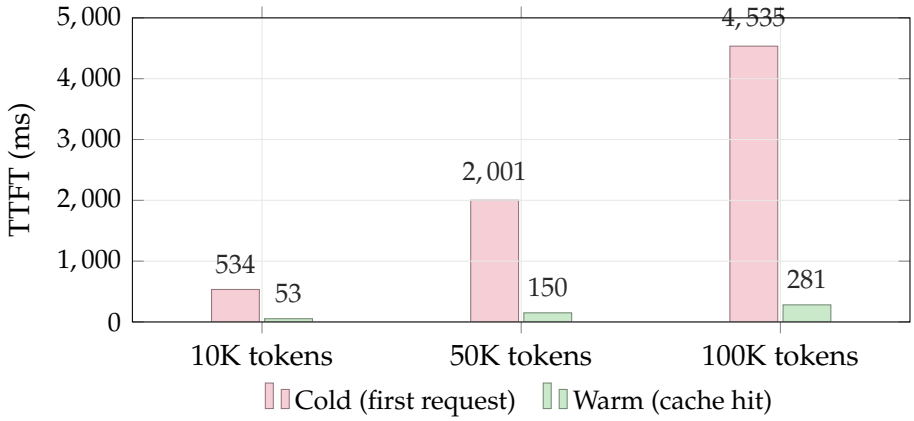

Figure 4.9: Experiment 6: prefix caching effectiveness for repeated long prefixes. The rose bars show cold TTFT when the prefix must be prefetched; the green bars show warm-cache TTFT when the shared prefix is reused. The experiment explains why prefix caching is a production-critical optimization for shared system prompts and RAG documents: at 100K tokens it reduces TTFT from 4,535 ms to 281 ms.

Across these experiments, the narrative is consistent: PagedAttention creates the capacity headroom, while scheduler and cache optimizations decide whether that headroom becomes lower TTFT, smoother streaming, or higher throughput for a specific workload. Run-to-run variance was small (coefficient of variation $<5\%$ for throughput and $<10\%$ for tail latency), consistent with fixed seeds and exclusive GPU access.

Production Evidence

Production reports validate that the same abstractions matter outside synthetic traces. Amazon Rufus reports using vLLM with AWS Trainium in a multi-node inference architecture across tens of thousands of chips for Prime Day traffic [120]; LinkedIn reports vLLM across **50+ generative AI use cases** on thousands of hosts, including workloads with nearly **1,000 tokens per candidate** and more than 50% shared prefix tokens [121]; and LMSYS Chatbot Arena reported **30,000 daily requests**, a **60,000-request peak**, and a **50% GPU-count reduction** through the FastChat-vLLM integration [122, 123]. These re-

ports are not controlled benchmarks, but they show the system-level value of the abstraction: memory efficiency becomes useful only when embedded in a deployable engine with scheduling, portability, and operational hooks.

Limitations

The evaluation scopes its claims deliberately. It uses one model size and one GPU; larger models and multi-GPU tensor parallelism would shift bottlenecks toward memory capacity and communication. The TGI comparison relies on published data rather than direct reproduction. Experiment 4 uses vLLM 0.6.6 because V0 exposes chunked-prefill controls directly, while V1 internalizes them in the scheduler. Experiment 6 has well-replicated warm-cache measurements, but each cold-cache point is a single miss observation because the cache persists across the server session.

Positioning

vLLM's position is best understood through analogy. Linux is not the fastest OS for every workload; Kubernetes is not the simplest orchestrator; PyTorch is not the most performant inference runtime. Yet each became a **default choice** for its domain by optimizing for the common case while remaining extensible. vLLM occupies an analogous position: not the absolute best on every dimension, but a practical center of gravity for many deployments—flexible enough for diverse models and hardware, performant enough for production SLOs, and open enough to extend for specialized requirements.

The data supports this positioning: the project's scale—documented in Section 4.6—spans contributors from NVIDIA, AMD, Intel, Meta, Google, AWS, and IBM, alongside reported production deployments at Amazon, LinkedIn, LMSYS, Roblox, and other organizations. The PyTorch Foundation's adoption of vLLM as a hosted project [101] formalizes this role within neutral open-source governance.

4.8 Summary

This chapter presented vLLM as the dissertation's memory-layer systems contribution: a serving engine built around the observation that the KV cache, not raw compute, is often the binding constraint for LLM inference. PagedAttention applies virtual-memory ideas to this new setting—demand allocation, fixed-size blocks, block tables, and copy-on-write—and turns them into a practical abstraction for continuous batching. The result is not a small allocator improvement: the controlled evaluation shows **99.5–99.6% KV utilization**, **3.9–5.6**$\times$ more concurrent requests than static allocation, and a system design that can absorb variable-length, long-context, and shared-prefix workloads.

The broader lesson is that a single good abstraction is necessary but insufficient. vLLM became useful because PagedAttention was embedded in a complete serving system: a

scheduler, memory manager, kernel path, prefix cache, optimization surface, hardware backends, and OpenAI-compatible API. The evaluation reinforces this system view. Paging creates headroom, chunked prefill protects decode latency, prefix caching collapses repeated-prefill TTFT, and speculative decoding helps only when workload structure justifies its overhead. Production reports from LMSYS, LinkedIn, Amazon, and others show that these mechanisms mattered beyond the benchmark setting, while the V1 rewrite and follow-on work such as Jenga and Pie show that the abstraction continued to evolve as models and hardware changed.

vLLM therefore supports the dissertation's thesis at the memory-management layer: purpose-built abstractions for inference can produce gains that general-purpose systems do not provide out of the box. It also exposes the next bottleneck. Even after better batching and KV-cache management, LLM inference still leaves substantial GPU resources idle because decode is memory-bound, batches are dynamic, and kernel launches remain fine-grained. Chapter 5 moves below the serving engine to ask whether the same systems principle—make the right resource boundary explicit—can improve utilization at the GPU kernel level.

This chapter presented vLLM as the dissertation's anchor contribution: PagedAttention eliminated KV cache fragmentation, Jenga generalized paging to hybrid architectures, and Pie explored NVLink-aware tiered memory. Together, they demonstrate that memory-first design can produce large gains. Chapter 5 pushes below the model boundary to address the remaining utilization gap at the GPU kernel level.

Chapter 5

GPU Kernel Multiplexing

5.1 The Case for Kernel-Level Multiplexing

The Utilization Paradox

The preceding chapters addressed efficiency at the model level: batching requests across users, scheduling work across multi-model pipelines, and managing memory for concurrent sequences. Ray Serve (Chapter 3) demonstrated that treating pipelines as compositional units enables systematic resource optimization. vLLM (Chapter 4) showed that PagedAttention can achieve near-zero memory waste while continuous batching maximizes request throughput. These innovations represent genuine advances—yet they leave a troubling gap unaddressed.

Even with these optimizations, GPU utilization in inference remains disappointingly low. Profiling production inference workloads reveals utilization rates of **20–40%**—the majority of GPU capacity sits idle [38]. For LLM inference specifically, independent GPU-level profiling finds that large-batch decoding can remain memory-bound: DRAM bandwidth saturation in attention leaves compute resources underutilized even when batching increases throughput [124].

This utilization paradox—sophisticated serving systems achieving excellent memory efficiency while leaving compute massively underutilized—reflects a fundamental mismatch between GPU architecture and inference workload characteristics. Modern GPUs contain thousands of compute units designed for massively parallel computation. The NVIDIA A100 provides 6,912 CUDA cores and 432 Tensor Cores; the H100 scales to 16,896 CUDA cores and 528 Tensor Cores delivering 1,979 TFLOPS in FP8. Yet inference workloads—even well-optimized ones—leave most of this capacity unused.

This chapter explores why model-level optimization is insufficient and presents the author's work pushing efficiency below the model boundary through kernel-level GPU multiplexing.

Sources of Underutilization

The gap between GPU capability and actual utilization stems from four interconnected sources, each representing a fundamental characteristic of inference workloads that model-level optimizations cannot address.

Small Batches. Latency constraints impose hard limits on batch sizes. A user waiting for a chatbot response cannot tolerate the delay of accumulating a large batch. A batch of 8 requests uses perhaps **5% of an A100's peak flops**—the remaining 95% of compute have no work. This is not a failure of scheduling; it is the inherent tension between latency SLOs and throughput optimization. Inoue [23] demonstrated that throughput and energy efficiency increase monotonically with batch size, but latency-sensitive applications cannot exploit this relationship.

Small batches leave SMs starved for work, but even when sufficient work exists, the nature of that work can leave compute idle for a different reason.

Memory-Bound Phases. LLM decode is fundamentally memory-bound—the GPU waits for memory transfers while compute units idle. Each token generation requires reading the entire KV cache from HBM, but performs relatively little computation per byte transferred. The A100's 2,039 GB/s memory bandwidth becomes the bottleneck long before its 312 TFLOPS of compute capacity is saturated. Even with optimal memory management through PagedAttention, memory bandwidth—not compute—limits throughput. Recasens et al. [124] find that DRAM bandwidth saturation in attention is the main limiter in large-batch LLM decoding, with more than half of attention-kernel cycles stalled on data access across their tested models.

Where small batches waste SMs for lack of parallel work and memory-bound phases waste them while waiting on data, a third source of underutilization wastes time in the gaps between useful operations altogether.

Kernel Launch Overhead and the On-Chip State Problem. Each CUDA kernel launch incurs **2–20 microseconds of overhead**—grid setup, kernel dispatch, driver traversal, and synchronization [125]. For large training kernels running milliseconds, this overhead is negligible. But inference involves many small kernels: layer normalizations, activation functions, residual additions, attention score computations. These kernels often complete in **10–100 microseconds** of actual work, making launch overhead approach or exceed computation time. In extreme cases, latency between API call and kernel execution can reach **175 microseconds** [38].

The overhead story does not end at launch latency. A more insidious cost lurks in the GPU memory hierarchy itself: between kernel invocations, all on-chip state is lost. Registers, shared memory (SRAM), and L1 cache contents do not persist across kernel boundaries—only global memory (HBM) survives [125]. This means that when kernel K_1 completes and kernel K_2 begins, any intermediate results that K_2 needs must round-trip through HBM, even if both kernels would have fit in on-chip storage. The bandwidth gap is enormous: on-chip SRAM delivers **~31–33 TB/s** of aggregate bandwidth on the H100, while HBM provides only **~3.35 TB/s**—roughly a **10× disparity** [126]. The latency gap is

similarly stark: L1/shared memory access takes **~30 cycles** while HBM access takes **~400 cycles**, a **13× penalty** [126]. For the many small, memory-bound kernels in inference (layer norms, activations, residual connections), this forced HBM round-trip between every operation dominates execution time. PyTorch's default implementation of RMSNorm, for instance, launches multiple small CUDA kernels each incurring **~5–10 microseconds** of overhead, leaving **89% of available bandwidth** unused compared to a fused implementation [127].

The fundamental problem remains: the CUDA programming model treats each kernel launch as an isolated scheduling event, discarding all on-chip state between invocations and paying launch overhead for each transition. CUDA Graphs, kernel fusion, and PDL each chip away at specific aspects of this overhead, but none address the structural issue that GPU resources sit idle between kernels when they could be serving other tenants' work. Kernel-level multiplexing attacks this problem from a different angle: rather than optimizing the sequential kernel pipeline, it fills the gaps with useful work from concurrent workloads.

The Opportunity

These four sources of underutilization share a common characteristic: they represent *temporal* gaps in resource usage. During memory stalls, compute is idle. During kernel launches, SMs wait. Between kernels, the entire GPU pauses. If multiple workloads could share the GPU simultaneously—different users' kernels running on different compute units during these gaps—utilization could approach the theoretical maximum.

The opportunity is substantial. We demonstrated a **7.7× opportunity gap** between naive sequential execution and optimal spatial-temporal packing of GPU kernels [38]. ResNet-50 inference on NVIDIA V100 achieves **under 30% utilization** at interactive latencies; even with larger batches, it struggles to reach **40%** of the 15.7 TFLOPS peak. The TPU v1, despite being purpose-built for inference, achieved **just 8.2% utilization** for text processing workloads [128]. These gaps represent wasted hardware capacity, not merely lower benchmark scores.

The challenge is achieving this sharing without sacrificing isolation, predictability, or correctness. Multiple tenants' kernels running concurrently must not interfere with each other's results, must meet their respective latency targets, and must handle failures gracefully. This chapter presents two approaches to this challenge: a software-only solution that demonstrated the opportunity, and a hardware-supported solution that makes kernel-level multiplexing practical.

Why Model-Level Multiplexing Is Not Enough

Chapter 3's model composition improves utilization by sharing infrastructure across models. When Model A is idle, Model B can use its resources. Ray Serve's actor-based architecture enables efficient context switching between models, and InferLine's pipeline op-

timization ensures end-to-end SLOs are met even with complex multi-stage workloads. But model-level sharing cannot address underutilization *within* a model's execution:

Temporal Gaps. Model A runs a kernel, then waits for memory. Model B's kernel could run during that wait—but model-level scheduling does not exploit this opportunity. The granularity is wrong: by the time the scheduler detects Model A's idle period and schedules Model B, the opportunity has passed.

Spatial Underuse. Model A's kernel uses 10% of SMs. Model B's kernel could use the other 90% concurrently—but without fine-grained coordination, they execute sequentially. The GPU's spatial parallelism is unexploited because the scheduling abstraction operates at too coarse a level.

Kernel Granularity Mismatch. Scheduling decisions in model-level systems happen at the request or batch level—milliseconds to seconds of granularity. But the opportunities for sharing exist at the kernel level—microseconds of granularity. This mismatch between request-scale decisions and kernel-scale opportunities means model-level schedulers cannot exploit kernel-level parallelism.

Achieving higher utilization requires scheduling *below* the model level—at the kernel level.

Evolution of Approaches

This chapter presents two approaches the author contributed to, representing an evolution in thinking about kernel-level multiplexing:

OoO JIT (2019): A software-only approach that intercepts kernel launches at runtime, analyzes dependencies, and combines compatible kernels for concurrent execution. Developed with Paras Jain, Ajay Jain, and collaborators at UC Berkeley [38], this work demonstrated the **7.7× opportunity gap** and achieved **1.25–2× throughput improvements** for multi-tenant workloads. The work revealed both the value of kernel-level multiplexing and the complexity of software-only solutions.

Green Contexts (CUDA 12.4+): Leveraging CUDA's Green Context API for hardware-supported SM partitioning. Rather than combining kernels in software, Green Contexts provision contexts with bounded SM resources, enabling isolation and sharing with less software complexity. This hardware support—arriving after the OoO JIT work demonstrated the value of kernel-level multiplexing—enables cleaner implementations with stronger guarantees.

The evolution from complex software solutions to simpler hardware-supported approaches reflects a broader pattern in systems research: demonstrating value through software prototypes motivates hardware support, which then enables production-quality implementations. Just as software RAID demonstrated value before hardware RAID controllers emerged, and software virtualization preceded Intel VT-x, the OoO JIT work helped establish that kernel-level multiplexing deserved hardware support.

5.2 Background: GPU Execution Model

The Hardware Foundation

Understanding kernel-level multiplexing requires understanding how GPUs execute code. This section provides necessary background on GPU architecture, CUDA streams, concurrent execution, and existing sharing mechanisms. Readers familiar with GPU programming may skim this section; those approaching from systems or ML backgrounds will find it essential context for the technical contributions that follow.

Modern GPUs are fundamentally different from CPUs in their approach to parallelism. Where CPUs optimize for single-thread performance through deep pipelines, branch prediction, and out-of-order execution, GPUs optimize for throughput through massive parallelism. An NVIDIA A100 contains **108 Streaming Multiprocessors (SMs)**, each with 64 CUDA cores and 4 Tensor Cores, for a total of 6,912 CUDA cores and 432 Tensor Cores. The H100 scales further to **132 SMs** with 16,896 CUDA cores and 528 Tensor Cores.

This architecture reflects the SIMT (Single Instruction, Multiple Threads) execution model—a direct descendant of Flynn's SIMD classification [129] and the vector processing pioneered by the Cray-1 [130]. Thousands of threads execute the same instruction on different data, achieving throughput impossible on serial processors. But this architecture also creates the utilization challenges described above: when threads have insufficient work or wait for memory, the massive parallelism becomes massive waste.

CUDA Streams and Concurrency

CUDA organizes GPU work into **streams**—ordered sequences of operations where each operation within a stream completes before the next begins. This abstraction serves as the fundamental building block for expressing parallelism: operations assigned to the same stream execute serially, ensuring that data dependencies are respected, while operations assigned to different streams may execute concurrently if sufficient resources are available. The programmer expresses parallelism intent through stream assignment; the GPU hardware scheduler determines actual execution order based on resource availability and internal heuristics.

The stream model provides a clean separation between correctness and performance. Correctness is guaranteed within streams—a kernel reading data produced by a preceding kernel in the same stream will always see the completed results. Performance, however, is opportunistic across streams. Modern GPUs support substantial concurrent execution: the A100 can theoretically execute **up to 128 concurrent kernels** from different streams [125]. But this concurrency is not guaranteed. The GPU scheduler makes runtime decisions based on available SMs, register file pressure, shared memory allocation, and a host of internal factors invisible to the application. Two kernels from different streams *might* run in parallel on disjoint SM sets, or they might serialize if either kernel demands resources that would conflict with concurrent execution.

This opportunistic scheduling lies at the heart of the multi-tenant GPU sharing problem. When User A and User B both launch kernels, neither can predict or control whether their work will execute concurrently or serially. User A's latency depends not only on their own kernel's resource requirements but on the opaque interactions between their kernels and User B's workload. The GPU provides no mechanism for User A to reserve resources, express priority, or even observe what resources are available. From the programmer's perspective, the GPU is a black box that accepts kernel launches and eventually returns results—the mapping from submissions to execution is hidden behind hardware scheduling decisions that optimize for aggregate throughput rather than per-tenant predictability.

Compounding this unpredictability is the overhead of kernel launches themselves. Each launch traverses a complex software stack: the application calls the CUDA runtime, which invokes the driver, which constructs command buffers and submits them to the hardware command processor for decoding and dispatch. This traversal incurs **2–20 microseconds** of latency in typical cases [125]. The overhead includes argument validation and copying to GPU-accessible memory, driver context management, command buffer construction with kernel parameters and launch configuration, and hardware command processor decoding. For large training kernels running for milliseconds, this overhead is negligible—a rounding error in total execution time. But inference involves many small kernels: layer normalizations completing in 10 microseconds, activation functions in 5 microseconds, residual additions in 3 microseconds. When kernel execution time approaches or falls below launch overhead, the GPU spends more time preparing to compute than actually computing.

CUDA Graphs address launch overhead by capturing an entire computation graph once and replaying it with minimal per-launch cost—**sub-microsecond replay** for previously instantiated graphs. But CUDA Graphs require static computation patterns: the graph topology, tensor addresses, and kernel configurations must be fixed at capture time. This requirement conflicts fundamentally with LLM serving, where batch sizes vary per request, sequence lengths differ across users, and continuous batching dynamically adds and removes requests from the active batch. The tension between launch efficiency and scheduling flexibility represents one of the core challenges that kernel-level multiplexing must address.

NVIDIA MPS (Multi-Process Service)

Multi-Process Service (MPS) represents NVIDIA's first systematic approach to GPU sharing across operating system processes. Without MPS, each process requires exclusive GPU access, and context switching between processes incurs **50–100 microseconds** of overhead—serializing what could be parallel work into expensive sequential time-slicing. MPS provides a unified GPU context through which multiple processes can submit work concurrently, reducing context switch overhead to sub-millisecond levels and enabling kernels from different processes to execute in parallel on the same GPU.

The architecture is straightforward: an MPS daemon interposes between client applications and the GPU driver, multiplexing their work submissions onto shared GPU resources within a single CUDA context. Current implementations support **up to 48 concurrent clients** on Volta and later GPUs. Despite improvements since its Kepler-era introduction—including per-client virtual address spaces and advisory SM limits via `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`—MPS provides no guaranteed resource isolation, no priority mechanism, and no fault isolation between clients sharing a context.

NVIDIA MIG (Multi-Instance GPU)

Multi-Instance GPU (MIG), introduced with the A100 architecture in 2020, takes the opposite approach from MPS: hardware partitioning that divides a physical GPU into up to **seven** isolated instances, each with dedicated SMs, HBM slices, L2 cache, and memory controllers. MIG provides true hardware isolation with only **~3% slowdown** [131], but its coarse, fixed profiles cannot express arbitrary allocations, reconfiguration requires draining all jobs and takes seconds, and instances cannot communicate via NVLink—making MIG unsuitable for fine-grained sharing or tensor-parallel workloads.

Related Academic Work

The limitations of MPS and MIG have motivated substantial academic research on GPU sharing. The field has progressed from cluster-level temporal sharing (Gandiva [58], 2018) through kernel-level spatial partitioning (GSLICE [132], REEF [133], KRISP [134], Paella [135]) to thread-block and TPC-level partitioning (Tally [136], LithOS [137], 2025). Software interposition systems (Orion [138], KRYPTON [139]) demonstrated production viability, while SGDRC [140] was the first to leverage NVIDIA's official SM masking interface. The trajectory is clear: each generation pushed to finer granularity with lower overhead, and NVIDIA's hardware support—MPS (2012) $\rightarrow$ MIG (2020) $\rightarrow$ Green Contexts (2023)—absorbed lessons from research systems that preceded it. The OoO JIT work described in Section 5.3 was among the earliest to explore kernel-level spatial coalescing; the subsequent growth of research reflects sustained interest in this layer of the stack.

5.3 Out-of-Order JIT Kernel Execution

The Software Approach

In OoO VLIW JIT compiler intercepts kernel launches at runtime, analyzes dependencies, and combines compatible kernels for concurrent execution. The work demonstrated a **7.7$\times$ opportunity gap** between naive execution and optimal spatial-temporal packing, with measured throughput improvements of **1.25–2.48$\times$** for multi-tenant workloads.

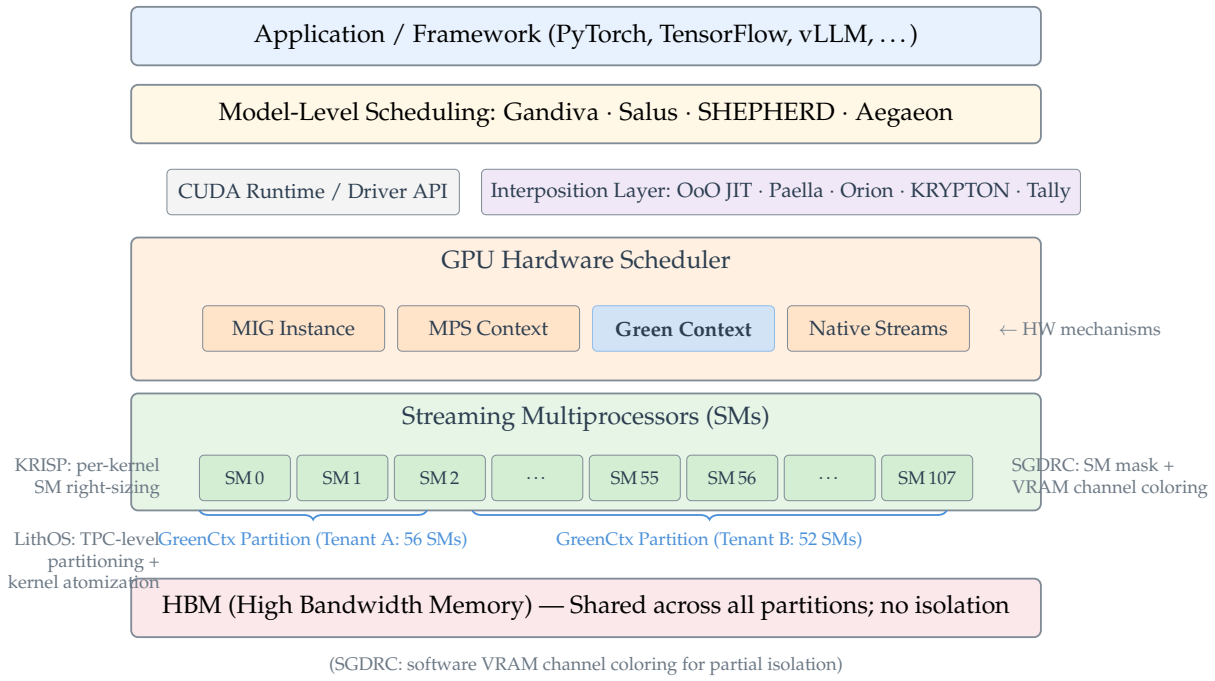

Figure 5.1:
GPU resource stack showing where each sharing system operates. The interposition layer (OoO JIT, Paella, Orion, KRYPTON, Tally) sits between the CUDA runtime and the hardware scheduler. Green Contexts provide hardware-enforced SM partitioning at a finer granularity than MIG.

The name reflects the technical approach: Out-of-Order (OoO) execution across independent kernel streams, inspired by Very Long Instruction Word (VLIW) processor design, implemented through Just-In-Time (JIT) compilation of kernel launch sequences.

At the time, this technique was motivated by the emergence of efficient batched and grouped GEMM implementations, which made it practical to multiplex many independent matrix multiplications. A batched GEMM executes a collection of independent GEMM operations in one call, usually with similar shapes, while a grouped GEMM generalizes this idea to sets of GEMMs with different dimensions or layouts. In modern terminology, this work can be viewed as a form of "horizontal fusion" for GEMM workloads: it dynamically batches independent just-in-time kernels so they can be scheduled and executed together more efficiently.

Motivation

The multi-tenant GPU serving problem crystallized the need for kernel-level multiplexing. Consider a common production scenario: a cloud GPU serving inference requests

from multiple users. User A submits an image classification request while User B submits a text generation request. Under the standard CUDA execution model, their kernels execute sequentially—User A's entire inference pass completes before User B's begins. Yet there is no fundamental reason for this serialization. The two users' computations are logically independent, operating on disjoint data with no shared state. If their kernels could execute concurrently on different subsets of the GPU's streaming multiprocessors, throughput would increase without affecting either user's latency.

This multi-tenant scenario has become the norm rather than the exception. Public cloud GPU instances command premium prices—$2–3/hour for an A100, twice or higher for an H100—yet many inference workloads cannot saturate a full GPU. A single BERT inference request might use 5% of an A100's compute capacity; a single GPT-2 generation pass might engage 15% of available SMs. Economic pressure therefore pushes cloud providers toward multi-tenancy, packing multiple users onto shared GPUs to amortize hardware costs. But achieving this packing efficiently requires coordination at a granularity that existing systems do not provide. Model-level schedulers like Ray Serve allocate entire GPUs to processes; MPS enables concurrent process execution but provides no control over resource allocation; MIG partitions hardware but at coarse, fixed granularities. None of these mechanisms can exploit the fine-grained temporal and spatial opportunities that exist at the kernel level.

Profiling production inference workloads revealed a striking pattern: the bottleneck was not kernel execution but kernel launch. The software overhead of traversing the CUDA stack—driver calls, argument validation, command buffer construction, hardware dispatch—serializes what the hardware could execute in parallel. Even when two kernels from different users have no data dependencies and could run concurrently on different SMs, the sequential launch process forces them into a serial pipeline. The overhead compounds across the many small kernels in a typical inference pass. ResNet-50 inference on NVIDIA V100 achieves **under 30% utilization** at interactive latencies; even with aggressively large batches, utilization struggles to reach **40%** of the 15.7 TFLOPS peak [38]. The Google TPU v1, despite being purpose-built for inference, achieved **just 8.2% utilization** for text processing workloads [128]. These numbers do not represent inherent limitations of the hardware—they represent the cost of a programming model that serializes inherently parallel work.

The central insight behind the OoO JIT approach was that this serialization could be reduced transparently. Modern GPUs support concurrent kernel execution—the A100 can run up to 128 kernels simultaneously if resources permit. The hardware is capable; the programming model simply fails to exploit that capability. A runtime layer that intercepts kernel launches, identifies independence relationships among pending kernels, and combines compatible kernels into concurrent batches could unlock spatial parallelism without requiring any application changes. The analysis and combination could happen just-in-time, adapting to the dynamic mix of workloads as users come and go. This idea—runtime independence analysis and kernel combining—became the foundation of the OoO VLIW JIT compiler.

Technical Approach

The OoO JIT draws its intellectual lineage from Very Long Instruction Word (VLIW) processor design. In VLIW architectures, the compiler—rather than the hardware—is responsible for identifying independent operations and packing them into wide instruction words that execute in parallel across multiple functional units. The hardware is simplified because the compiler guarantees that co-scheduled operations have no dependencies. The Itanium processor family and the TI C6000 DSP series demonstrated this approach at the instruction level; the OoO JIT extends it to the GPU kernel level. Independent kernels from different streams or users are identified and packed into combined launches that execute concurrently on different SM partitions. The critical difference from classical VLIW is that the "compiler" operates at runtime (just-in-time) rather than ahead-of-time, because the kernel mix is inherently dynamic—different users submit different workloads at unpredictable times, and the set of available kernels for combining changes continuously.

The system achieves transparency through interposition on the CUDA runtime using `LD_PRELOAD`, a mechanism that redirects shared library function calls without modifying application binaries. When an application issues a kernel launch, the OoO JIT intercepts the call before it reaches the CUDA driver. The interception captures the complete launch descriptor: kernel function pointer, grid and block dimensions, shared memory requirements, stream assignment, and all kernel arguments. From these descriptors, the system estimates the resource footprint of each kernel—registers per thread, shared memory per block, and expected SM occupancy. The captured kernel is then placed into a pending queue rather than being immediately dispatched. The system monitors this queue for combining opportunities: when multiple independent kernels are pending, or when a configurable timeout expires, the combiner constructs a batched launch.

The correctness of kernel combining rests on dependency analysis. Not all pending kernels can be safely co-executed. The OoO JIT performs conservative analysis along three dimensions, and a kernel pair must satisfy all three to be eligible for combination. First, the system checks *memory dependencies* by tracking all GPU memory allocations (intercepting `cudaMalloc` and `cudaFree`) and analyzing kernel arguments to identify pointer parameters that reference GPU memory. If kernel K_1 writes to a memory region that kernel K_2 reads—or vice versa—the two kernels have a true data dependency and must execute in their original order. The system maintains a shadow memory map that records which allocations are accessed by which kernels, enabling efficient conflict detection. Second, the system enforces *CUDA stream semantics*. Kernels assigned to the same stream carry an implicit ordering guarantee: K_1 must complete before K_2 begins. Only kernels from different streams are candidates for combining; within-stream ordering is always preserved. Third, the system checks for *resource conflicts*. If two kernels' combined requirements—registers, shared memory, thread blocks—exceed the GPU's capacity, colocating them would force the hardware to serialize their execution anyway, providing no benefit while introducing combining overhead. The system estimates per-kernel resource consumption from the launch descriptors and rejects combinations that would exceed

hardware limits.

When the dependency analysis identifies a set of mutually independent kernels—a maximal independent set in the dependency graph—the combiner constructs a spatially coalesced launch. Multiple kernel descriptors are packed into a single submission to the GPU's command processor, with SM partitioning directives that assign different kernels to different subsets of streaming multiprocessors. The GPU executes the coalesced kernels concurrently on their assigned partitions, and the system routes completion notifications back to the originating streams, preserving the illusion of sequential per-stream execution. The resulting "superkernels"—VLIW-inspired launch bundles containing multiple independent GPU programs—achieve spatial multiplexing that the standard CUDA launch model cannot express.

Implementation

The OoO JIT implementation forms a four-stage pipeline on the critical path of every kernel launch: interception (via `LD_PRELOAD`), dependency tracking, combining, and dispatch. The system's correctness guarantee is semantic equivalence: combined execution must produce identical output to sequential execution, enforced through conservative dependency analysis that favors safety over combining opportunities.

Evaluation

The system was evaluated on multi-tenant inference scenarios using NVIDIA V100 GPUs. Workloads included image classification (ResNet-18, ResNet-50), object detection (SSD), and recurrent networks (LSTM language models)—representative of the diverse model types that coexist in production serving environments.

The most important result was quantifying the opportunity: spatial coalescing of ResNet-18 convolution kernels achieved a **7.71× throughput improvement** over time-slicing baselines, demonstrating the enormous gap between current practice and what the hardware can deliver when kernels are properly colocated. For memory-bound recurrent workloads (RNN/LSTM matrix-vector multiplications), coalescing achieved **2.48× throughput improvement**, showing that even memory-intensive workloads benefit from spatial sharing because their compute idle time can be filled by co-tenants. Coalesced kernels outperformed space-only multiplexing via Hyper-Q by **3.23×**, demonstrating that the combined spatial-temporal packing of the VLIW approach captures opportunities that static spatial partitioning cannot. End-to-end auto-tuned collaborative execution achieved **1.25× throughput improvement** at maximum—lower than the microbenchmark gains, reflecting the overhead of runtime analysis and the conservative dependency checking that prioritizes correctness over performance.

Per-kernel overhead averaged **5–15 microseconds** for interception and dependency analysis. For inference kernels taking 10–100 microseconds of actual computation, this overhead is acceptable—the combining benefits exceed the interception costs. For very

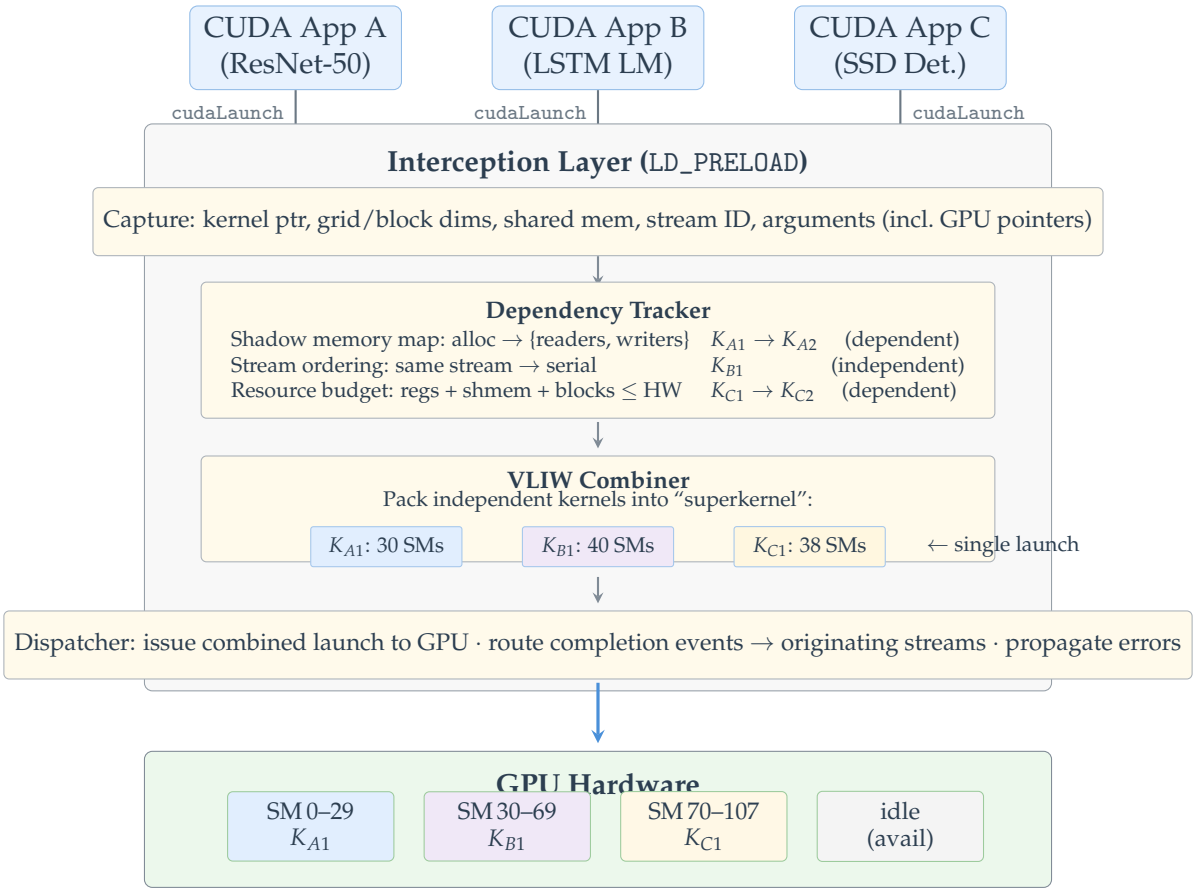

Figure 5.2: Architecture of the OoO VLIW JIT compiler. CUDA applications' kernel launches are intercepted via `LD_PRELOAD`, analyzed for dependencies, combined into "superkernels," and dispatched for concurrent execution on disjoint SM partitions.

Table 5.1: OoO JIT evaluation results across multi-tenant inference workloads on NVIDIA V100.

Workload	Improvement	Comparison
ResNet-18 convolution coalescing	**7.71**×	vs. time-slicing
Matrix-vector (RNN/LSTM)	**2.48**×	vs. sequential
Coalesced kernels	**3.23**×	vs. Hyper-Q (space-only)
Auto-tuned collaborative	**1.25**×	end-to-end

small kernels below 10 microseconds, however, the overhead becomes a significant fraction of execution time, and combining may not provide net benefit. This overhead floor represents a fundamental limitation of software interposition: every intercepted call must traverse additional code paths before reaching the GPU.

The benefit varied predictably with workload characteristics. Multi-tenant scenarios with heterogeneous models—different architectures, different resource profiles—provided the richest combining opportunities because their kernels had complementary resource demands. Memory-bound phases (common in decode) left compute resources idle that could be filled by co-tenants' compute-intensive kernels. Conversely, workloads dominated by few large kernels offered limited opportunities, as did homogeneous multi-tenant scenarios where all users ran the same model (producing kernels with similar resource profiles that competed rather than complemented).

Limitations and Lessons

The OoO JIT work achieved its primary goal—demonstrating that kernel-level multiplexing unlocks substantial utilization improvements—but the experience also revealed fundamental limitations of software-only approaches that informed the subsequent direction of this research.

Software-only multiplexing is inherently complex. The system must comprehensively understand CUDA's execution semantics—memory ordering, stream synchronization, error propagation, launch overhead—and maintain this understanding across driver versions that evolve without backward-compatibility guarantees. Edge cases proved particularly challenging: custom CUDA usage patterns that bypassed standard launch APIs, undocumented driver behaviors that changed between minor versions, and GPU architecture differences that affected resource limits and scheduling heuristics. Each edge case required investigation and often bespoke handling, creating a maintenance burden that grew with the system's scope.

The interposition approach was also fragile in ways that limited production viability. Because the system interposes on the CUDA runtime, any change in the runtime's internal behavior—argument buffer formats, synchronization protocols, error codes—could silently break combining correctness. Validating each CUDA driver update against the full space of combining scenarios was impractical, meaning that production deployment would require either pinning driver versions (sacrificing performance improvements and security patches) or maintaining a continuous integration pipeline against NVIDIA's release schedule.

A subtler limitation was isolation. Kernel combining improves throughput by colocating kernels on the same GPU, but colocated kernels share memory bandwidth, L2 cache, and memory controller resources. These shared resources create interference paths that the combining system cannot eliminate. A co-tenant's memory-intensive kernel could saturate HBM bandwidth, degrading a latency-sensitive user's performance even when the two kernels execute on disjoint SM partitions. The OoO JIT provided no mechanism for

memory bandwidth isolation—a limitation shared with MPS and, as later research also identified for Green Contexts [141].

Perhaps the most consequential lesson was what the work implied about the right division of labor between software and hardware. The OoO JIT reconstructed—through interposition and runtime analysis—information that the GPU hardware already possessed: which SMs are free, which kernels can colocate, and how to partition resources among concurrent workloads. This reconstruction was expensive, fragile, and necessarily approximate. The experience crystallized a clear conclusion: if the hardware provided multiplexing primitives directly—APIs to query resource availability, partition SMs, and enforce isolation—the system could be dramatically simpler, more robust, and more efficient. Rather than combining kernels in software with correctness risks, one could simply request isolated hardware partitions and launch kernels normally.

This conclusion motivated the investigation of hardware-supported approaches described in Section 5.4. When CUDA 12.4 exposed Green Contexts in the Driver API, providing the hardware primitives the OoO JIT experience had identified as necessary, the path forward became clear.

5.4 GreenContext: Hardware-Supported SM Partitioning

The Hardware Opportunity

The complexity of software-only kernel multiplexing motivated investigation of hardware support. CUDA 12.4 exposed *Green Contexts* in the Driver API—a mechanism for creating lightweight CUDA contexts with limited SM resources [125]. This hardware primitive enables cleaner implementations of kernel-level multiplexing: rather than combining kernels in software, Green Contexts assign fixed SM partitions to different workloads, achieving isolation and sharing with minimal software complexity.

The timing was not coincidental. Research systems like OoO JIT, Salus, and REEF had demonstrated the value of fine-grained GPU sharing. Industry demand for multi-tenant inference grew as GPU costs increased and utilization remained low. NVIDIA responded with hardware support that codified the patterns these research systems had pioneered.

CUDA Green Contexts

Green Contexts are lightweight CUDA contexts with explicit resource limitations—specifically, a bounded set of SMs on which kernels may execute. When creating a Green Context, the application queries available SM resources, partitions them into disjoint sets, and creates contexts bound to each partition. The CUDA Driver API provides functions to query total SM availability (`cuDevGetDevResource`), split resources into partitions (`cuDevSmResourceSplitByCount`), generate resource descriptors, and create contexts bound to those descriptors (`cuGreenCtxCreate`). Once created, kernels launched within

a Green Context can only execute on the allocated SMs—the hardware enforces this limitation with no runtime overhead on the kernel launch path.

The API has evolved rapidly since its introduction. CUDA 12.4 provided the initial Driver API implementation, requiring applications to work directly with low-level resource management primitives. CUDA 13.1 added Runtime API exposure for Green Contexts and static SM partitioning for MPS, improving the path for framework adoption while preserving the same core idea: explicit SM resource provisioning for latency-sensitive work [142].

Hardware constraints shape the partitioning granularity achievable on each GPU generation. Pascal-generation GPUs (compute capability 6.x) permit partitions as small as a single SM with no alignment requirements—maximum flexibility. Volta and Turing (7.x) impose a minimum of 2 SMs with partitions aligned to multiples of 2, reflecting the paired-SM design of those architectures. Ampere (8.x) and Hopper (9.x) require a minimum of 4 SMs with 2-SM alignment, and Hopper additionally reserves 2 SMs for Cooperative Groups and Dynamic Parallelism operations. These constraints reflect the underlying hardware scheduling granularity: the GPU's work distributor operates on SM groups rather than individual SMs, and the partitioning API exposes this structure rather than hiding it.

The implications for kernel-level multiplexing are profound. SM partitioning becomes a hardware feature with guaranteed enforcement rather than a software approximation with best-effort semantics. One context's kernels cannot consume another context's SMs—the hardware scheduler enforces isolation without any software interposition in the launch path. There is no software interposition on each kernel launch: standard launches work within a Green Context, avoiding the microseconds of interception latency that software approaches require. The GPU's hardware scheduler retains responsibility for optimizing kernel execution within each partition, applying the same scheduling intelligence it would apply to an exclusive-access scenario—the system benefits from hardware optimization without reimplementing it in software. And the programming model remains clean: applications create partitioned contexts during initialization, associate workloads with contexts through standard CUDA context management, and launch kernels using familiar APIs. The complexity is confined to the cold path of context creation, not the hot path of request serving.

Building on OoO JIT Insights

The OoO JIT work supported a fundamental premise: kernel-level multiplexing can substantially improve GPU utilization in favorable workload mixes. The **7.7× opportunity gap** between sequential execution and optimal spatial-temporal packing demonstrated that the headroom exists—the question was how to capture it practically. Green Contexts provide that practical path, operationalizing the insight with hardware support rather than software reconstruction.

The architectural simplification is dramatic. OoO JIT required continuous interposition on the hot path: intercepting every kernel launch, analyzing dependencies across all pending kernels from multiple tenants, identifying combining opportunities based on resource complementarity, constructing combined kernel launches with merged arguments and coalesced memory operations, dispatching to the GPU and routing results back to the correct originating streams, and handling errors while maintaining CUDA stream semantics—all while keeping overhead below the threshold where combining benefits would be negated. Every step in this pipeline introduced complexity, correctness risks, and latency. Green Contexts collapse this entire machinery to three operations: create partitioned contexts during initialization, associate incoming workloads with appropriate contexts based on tenant identity or priority, and launch kernels normally through standard CUDA APIs. The complexity moves from the hot path—executed on every kernel launch—to the cold path—executed once during context creation. In the measured hot path, the software interception overhead drops from 5–15 microseconds per kernel to no observable launch-path overhead beyond the execution-time effect of using fewer SMs.

The isolation properties also strengthen. OoO JIT's kernel combining improved throughput by colocating kernels on the same SM partitions, but colocated kernels necessarily share resources beyond SMs: memory bandwidth, L2 cache capacity, memory controller queues. A memory-intensive co-tenant kernel could saturate HBM bandwidth, degrading a latency-sensitive kernel's performance even when the two execute on disjoint SM sets. Green Contexts provide stronger spatial isolation: SMs are hardware-partitioned, and each partition receives dedicated scheduling resources from the GPU's work distributor. While memory bandwidth remains shared—Green Contexts do not partition memory controllers or HBM channels—the compute isolation removes one measured interference path for compute-bound workloads. A latency-sensitive workload in one partition cannot be delayed by a throughput-oriented workload monopolizing SM resources in another, enabling tighter SLO guarantees than software-only approaches could provide.

Design Considerations

Green Contexts raise fundamental allocation questions that have no universal answer—the right strategy depends on workload characteristics, SLO requirements, and operational constraints. The simplest approach is **static equal partitioning**: divide SMs equally among tenants, giving each an identical share of compute resources. This strategy is trivial to implement and reason about—each tenant knows exactly what resources they have, and there are no complex interactions to debug. But equal partitioning is inefficient when workloads are imbalanced. A small model serving lightweight requests receives the same allocation as a large model processing complex queries; the small model's SMs sit idle while the large model queues work waiting for its insufficient partition. The strategy optimizes for simplicity at the cost of utilization.

Proportional allocation addresses this inefficiency by sizing partitions according to expected load or model requirements. A model expected to handle 60% of traffic receives 60% of SMs; a model handling 10% receives 10%. This approach achieves better utilization when predictions are accurate, but introduces a prediction problem: how does the system know traffic distribution before requests arrive? Static proportional allocation works well when workload characteristics are stable and known—the same model mix serving the same traffic patterns day after day. It fails when traffic shifts, new models are deployed, or request patterns change unexpectedly. The system has no mechanism to adapt allocations to runtime reality.

A more sophisticated approach combines **minimum guarantees with a shared overflow pool**. Each tenant receives a guaranteed minimum SM allocation sufficient for their baseline load, plus access to a shared pool of SMs that absorbs traffic bursts. A tenant experiencing a spike can temporarily expand into the shared pool; when the spike subsides, those SMs return to shared availability. This strategy balances isolation—the guaranteed minimum ensures no tenant can be starved—with efficiency—the shared pool prevents dedicated resources from sitting idle during low-traffic periods. The tradeoff is implementation complexity: the system must track pool availability, manage contention when multiple tenants burst simultaneously, and handle the transition when a tenant's allocation changes. For inference serving with bursty traffic patterns and strict SLOs, this complexity is often worthwhile.

LLM inference adds phase-specific considerations that complicate allocation further. The prefill phase—processing input tokens in parallel—is compute-intensive and benefits from more SMs; parallelism across input positions can saturate available compute capacity. The decode phase—generating output tokens one at a time—is memory-bandwidth-bound and derives limited benefit from additional SMs; memory bandwidth becomes the bottleneck long before compute capacity is exhausted. An optimal allocation would dynamically shift SMs between phases: expand during prefill when compute is valuable, contract during decode when it is not, freeing SMs for other tenants' prefill phases. But context reconfiguration is expensive—Green Context creation takes ~**1.1 ms** on H200, with occasional spikes to 3–4 ms (measured in Section 5.4, Figure 5.3)—and for short sequences where prefill and decode alternate rapidly, the reconfiguration overhead may exceed the benefit. In practice, static allocation with phase-appropriate sizing often suffices: provision enough SMs to avoid bottlenecking prefill, accept that decode will not fully utilize them, and rely on multi-tenant sharing to fill the gaps with other tenants' work.

Dynamic SM reallocation remains valuable for longer-timescale adaptation. Green Contexts can be destroyed and recreated with different SM allocations, enabling a management layer to respond to load changes. The layer monitors utilization and queue depths per context, detects imbalances where some contexts underutilize their allocations while others queue work, and triggers reallocation when the imbalance exceeds a threshold. The reconfiguration overhead limits reallocation frequency to minutes or hours rather than per-request—this is not a mechanism for fine-grained load balancing

but for adapting to diurnal patterns, traffic shifts between model versions, or gradual changes in workload mix. For these coarser-grained adaptations, periodic reallocation captures most of the benefit without incurring continuous overhead.

Implementation

Integrating Green Contexts into an inference serving system requires modifications at two levels: initialization-time context setup and request-time routing. The initialization path uses the CUDA Driver API to query available SM resources, partition them according to the chosen allocation strategy, and create Green Context handles bound to each partition. A typical setup for a two-tenant deployment queries total SM availability, splits the pool into two partitions—perhaps a 60/40 split favouring the higher-traffic tenant—generates resource descriptors for each partition, and creates Green Context objects that kernels will execute within. This initialization occurs once at server startup and takes on the order of milliseconds, dominated by driver initialization rather than partitioning logic.

The runtime path integrates with the serving system's scheduler through context routing. The scheduler maintains a mapping from logical tenants—identified by user identity, model name, priority level, or request metadata—to Green Context handles. When a request arrives, the scheduler determines the appropriate context, sets it as current for the executing thread (via `cuCtxSetCurrent` or by binding streams to contexts), and invokes the model's inference path. Kernels launch through standard CUDA APIs—`cuLaunchKernel` or the «<»> syntax—with no modification to kernel code. The hardware automatically restricts execution to the context's allocated SMs. From the kernel's perspective, nothing has changed; from the system's perspective, SM bounds are enforced by hardware.

The scheduler also maintains per-context telemetry: queue depths indicating how many requests are waiting for each context, utilization metrics measuring how fully each partition's SMs are occupied, and latency distributions tracking request completion times. This telemetry feeds the reallocation policy, triggering partition adjustments when imbalances persist. A context consistently showing high queue depths while neighbours show low utilization indicates a partition that needs more SMs; the policy can destroy and recreate contexts with adjusted allocations to rebalance. The reallocation overhead—measured at $\sim$**1.1 ms** per create and $\sim$**15** μ**s** per destroy on H200 (Figure 5.3)—constrains reallocation to infrequent events (minutes to hours) rather than per-request decisions. The first creation in a fresh process takes $\sim$100 ms due to one-time driver initialization, but subsequent operations are fast.

The critical performance property is **no measured software overhead** on the kernel launch path. Once contexts are created and the routing table is established, request serving proceeds with standard CUDA operations. There is no interception, no dependency analysis, no software combining—just context selection followed by normal kernel launches. The complexity is entirely confined to initialization and infrequent reallocation,

leaving the hot path—executed thousands of times per second during serving—as fast as baseline CUDA execution.

Evaluation

The evaluation of Green Contexts on NVIDIA H200 (132 SMs, CUDA 12.x) establishes four focused results: the API is viable for initialization-time setup, SM partitioning removes measurable compute-bound interference in a raw Driver API microbenchmark, single-tenant vLLM still leaves utilization headroom, and the H200 kernel-combining opportunity differs sharply from the 2019 V100 result. Results are summarized in Figures 5.3–5.6; detailed protocols are in the experiment protocol (Part C).

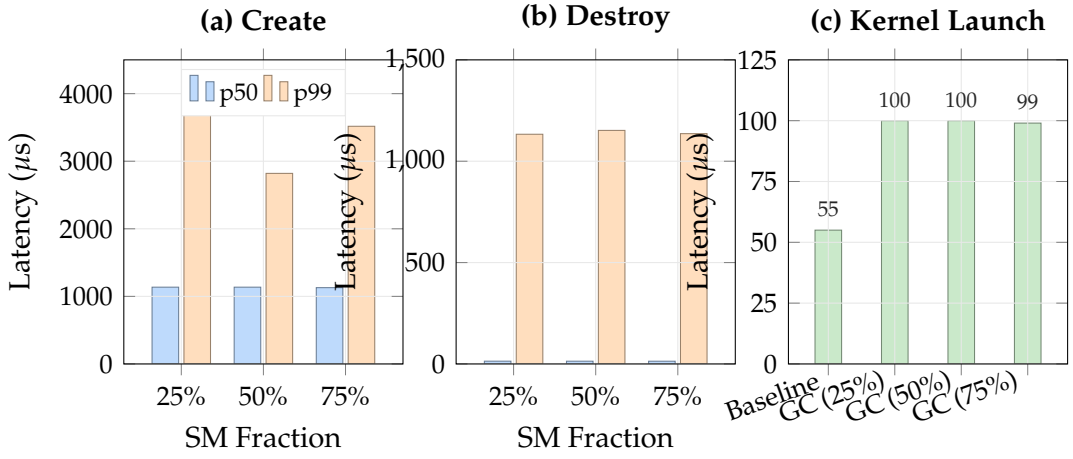

Figure 5.3: Green Context API overhead on H200 (132 SMs). (a) Create latency: ~1.1 ms (p50) regardless of SM partition size, with p99 spikes to 3–4 ms. (b) Destroy latency: ~14 μs (p50); occasional p99 spikes to ~1.1 ms. (c) Kernel launch: a GEMV(32×4096) takes 55 μs on 132 SMs (baseline) and ~100 μs on a Green Context partition—the difference reflects fewer SMs, not launch overhead. (100 trials per run, 3 runs per SM fraction.)

API overhead (Figure 5.3). Green Context creation takes **1.1 ms** (p50) regardless of SM partition size, with p99 spikes to 3–4 ms. Destruction is fast at **~15 μs**. The first creation in a fresh process incurs ~100 ms of one-time driver initialization. Kernel launches within a Green Context add no measurable overhead beyond the execution time difference from fewer SMs—a GEMV(32×4096) takes 55 μs on 132 SMs and 100 μs on 32 SMs, but the launch mechanism itself is identical. These numbers support using Green Context operations for initialization-time setup and infrequent reallocation (minutes-to-hours timescale), while the kernel launch hot path is unaffected in this microbenchmark.

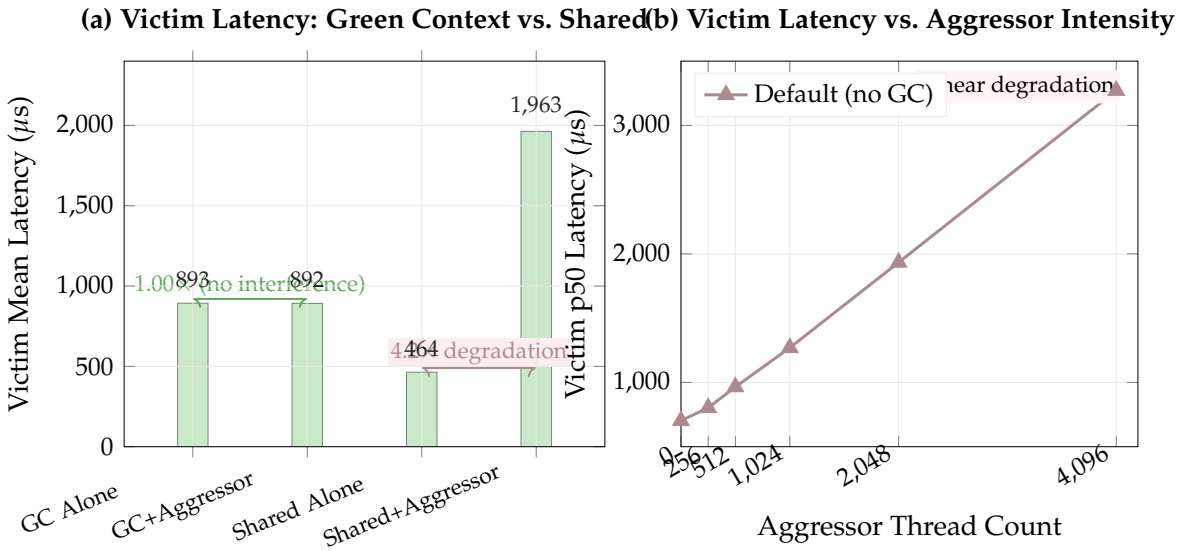

Figure 5.4:
Hardware SM isolation eliminates kernel-level interference. (a) With Green Contexts (64 victim SMs, 68 aggressor SMs), victim latency is identical with and without a concurrent aggressor: 893 μs vs. 892 μs (1.00×, $\sigma = 0.8$ μs). Without Green Contexts, the aggressor degrades victim latency by 4.2×: 464 μs $\rightarrow$ 1,963 μs ($\sigma = 8{,}068$ μs; p99 = 32 ms).
(b) Without Green Contexts, victim p50 latency degrades linearly as aggressor thread count increases from 0 to 4,096 (703 μs $\rightarrow$ 3,271 μs). Measured on H200 (132 SMs), compute-intensive kernels launched via CUDA Driver API directly within Green Contexts. (500 trials per configuration, 3 runs.)

Hardware SM isolation (Figure 5.4). The central result: Green Context SM partitioning removes measurable SM scheduling interference in this compute-bound raw Driver API microbenchmark. A compute-intensive victim kernel (132 blocks × 256 threads, 200K iterations) was measured with and without a concurrent aggressor kernel running in a separate context. Without Green Contexts (shared CUDA context), the aggressor degrades victim latency by **4.2**× (464 μs $\rightarrow$ 1,963 μs). With Green Contexts (64 victim SMs, 68 aggressor SMs), the victim shows **1.00**× **degradation** (893 μs $\rightarrow$ 893 μs)—the aggressor has no measurable effect in this setup. The victim's standard deviation drops from 8,068 μs (shared) to 0.8 μs (Green Context), meaning hardware partitioning removes the observed average interference and collapses variance for this workload pair. This result was obtained using raw CUDA Driver API kernel launches (`cuLaunchKernel`) directly within Green Contexts, bypassing framework-level context management; integrating Green Contexts with serving frameworks like PyTorch or vLLM requires framework-native support, as `cuCtxPushCurrent` alone does not redirect framework-managed operations.

The aggressor sweep (Figure 5.4b) further quantifies the interference that Green Con-

texts remove in this compute-bound setup. Without hardware partitioning, victim p50 latency degrades linearly with aggressor intensity: from 703 μs with no aggressor to 3,271 μs with 4,096 aggressor threads—a 4.7× degradation. At intermediate points, 256 aggressor threads cause a 14% increase; 1,024 threads cause an 81% increase. This linear degradation characteristic means that interference is hard to predict in multi-tenant environments where aggressor intensity varies over time—precisely the scenario that hardware SM partitioning is designed to reduce.

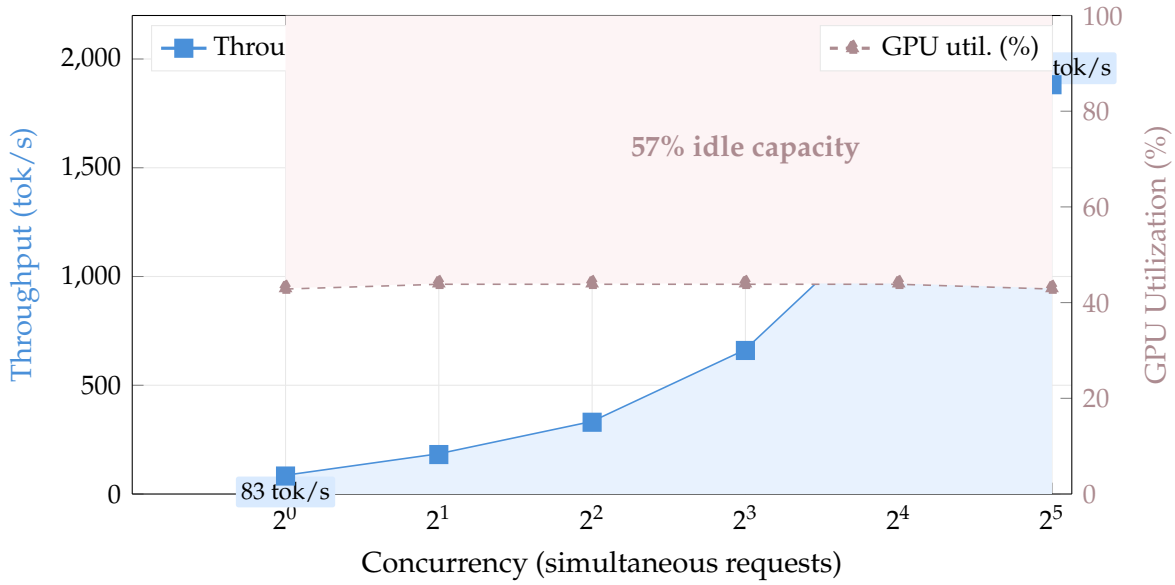

Figure 5.5: LLM inference utilization gap on H200 running Qwen-7B via vLLM. Throughput scales near-linearly from 83 tok/s ($c = 1$) to 1,882 tok/s ($c = 32$) through continuous batching (solid blue, left axis). Yet GPU utilization (dashed, right axis) remains flat at ~43% regardless of concurrency—57% of compute capacity sits idle even under heavy load. This directly confirms the utilization paradox: model-level optimizations maximize request throughput within a single tenant but cannot address the fundamental memory-boundedness of LLM decode. The idle capacity represents the opportunity that multi-tenant sharing via Green Contexts can exploit. (25 samples per concurrency level.)

Utilization gap (Figure 5.5). Profiling Qwen-7B inference on vLLM at concurrency levels 1–32 reveals that GPU utilization (`nvidia-smi`) remains flat at **~43% regardless of concurrency**. Throughput scales 23× from 83 tok/s ($c = 1$) to 1,882 tok/s ($c = 32$) through continuous batching, yet the reported utilization remains 57 percentage points below full-device utilization even under heavy load. This supports the utilization paradox described in Section 5.1: model-level optimizations maximize request throughput within

a single tenant, but do not fully address the memory-boundedness of LLM decode that leaves compute underutilized. The reported headroom motivates multi-tenant sharing—via Green Contexts or MPS—as the next integration target.

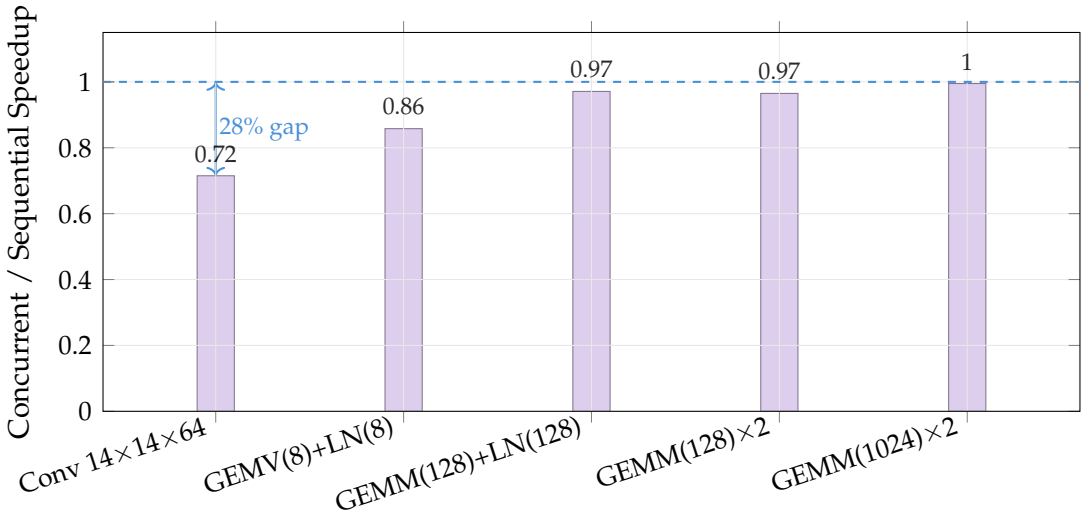

Figure 5.6: Evolution of the kernel-level opportunity gap. On H200 (2025), concurrent kernel execution on separate CUDA streams provides $\leq$1.0$\times$ speedup for all tested pairs: modern cuBLAS saturates all 132 SMs even for small kernels. Small convolutions show a 28% overhead from stream management; large GEMMs (1024$\times$1024) reach 0.995$\times$—effectively no gap. Contrast with V100 (2019), where Jain et al. [38] measured 1.25–7.71$\times$ speedup. The gap closure motivates the shift from software combining to hardware partitioning. (200 trials per pair, 3 runs; error bars omitted for clarity—CI widths $<$0.02.)

Opportunity gap evolution (Figure 5.6). The 7.7$\times$ concurrent kernel speedup demonstrated on V100 by the OoO JIT work does not appear in the tested H200 kernel pairs. Across five pairs spanning small convolutions to large matrix multiplications, concurrent execution on separate CUDA streams provides no benefit—and slight overhead (0.72–0.99$\times$) due to stream management costs. In these cases, modern kernels leave little spatial overlap for OoO-style combining to exploit. This finding strengthens rather than undermines the thesis argument: the opportunity has shifted from software kernel combining to hardware resource partitioning. OoO JIT's mechanism—intercepting and combining independent kernels—was effective when libraries left SMs idle. On modern hardware, the better-supported mechanism for this chapter's setting is SM partitioning via Green Contexts, which provides isolation without requiring kernel-level interposition.

Comparison with OoO JIT. The comparison with OoO JIT illuminates the tradeoffs between software and hardware approaches. OoO JIT incurs **5–15 microseconds per kernel** for interception and dependency analysis; in this benchmark, Green Contexts add no measured launch-path overhead once contexts are established. OoO JIT requires maintaining a complex interposition layer across CUDA driver versions; Green Contexts require only documented API calls. OoO JIT provides software-managed isolation that can be violated by bugs or edge cases; Green Contexts provide hardware-enforced SM bounds through the GPU scheduler. Where OoO JIT retains an advantage is flexibility: dynamic combining can adapt to workload changes on a per-kernel basis, while Green Context partitions are fixed until explicit reconfiguration. For inference serving—where workload mixes evolve over minutes to hours rather than microseconds—this flexibility is rarely valuable enough to justify the complexity cost.

Recent independent research reports a consistent limitation. A 2026 study on performance isolation in edge GPU systems [141] evaluated MPS, MIG, and Green Contexts across A100 and Jetson Orin platforms, finding Green Contexts promising for fine-grained SM allocation but limited by the absence of memory isolation. Green Contexts partition compute but not memory bandwidth; for memory-bound workloads, interference through shared HBM and memory controllers remains possible. Addressing this limitation—potentially through software memory bandwidth budgeting layered atop Green Context compute isolation, or through future hardware support for memory partitioning analogous to SM partitioning—remains an open direction.

Table 5.2: Summary of Green Context evaluation results on H200 (132 SMs, CUDA driver 590.48). Full protocols and reproducibility details are in the experiment protocol (Part C).

Experiment	Metric	Value	Comparison	Implication
C.1: API Overhead	Create p50	1.1 ms	—	Suitable for init-time setup
	Destroy p50	14 μs	—	Fast teardown
	Kernel launch (GC)	100 μs	55 μs baseline	Difference = fewer SMs, not overhead
C.2: SM Isolation	GC victim+aggressor	893 μs	893 μs alone	1.00×: no measured interference
	Shared victim+aggr.	1,963 μs	464 μs alone	4.2× degradation
	Victim σ (GC)	0.8 μs	8,068 μs shared	Variance sharply reduced
C.3: Utilization Gap	GPU util. at c=32	43%	—	57 pp reported headroom
	Throughput scaling	83 → 1,882 tok/s	23×	Batching helps, util stays flat
C.4: Opportunity Gap	Conv speedup	0.72×	V100: up to 7.7×	SM saturation on modern HW
	GEMM(1024) speedup	0.995×	—	No measured gain

5.5 Discussion

The preceding sections presented two concrete systems—OoO JIT and Green Contexts—that attack GPU underutilization at the kernel level. This section steps back from imple-

mentation details to examine three broader questions that these systems raise. First, what does the progression from software to hardware multiplexing reveal about how systems research drives hardware evolution? Second, how does kernel-level multiplexing integrate with the model-level optimizations described in Chapters 3 and 4 to form a complete efficiency stack? Third, how should practitioners choose among the multiplexing mechanisms now available, and what capabilities remain missing?

The Evolution Pattern

The progression from OoO JIT to Green Contexts illustrates a broader pattern in systems research: software prototypes demonstrate value, motivating hardware support that enables simpler, more robust production implementations. This pattern—software demonstrates, hardware enables—recurs throughout systems history and offers a lens for understanding the trajectory of GPU multiplexing research.

OoO JIT demonstrated that kernel-level multiplexing could unlock the **7.7× opportunity gap** between sequential execution and optimal packing on V100 hardware. The evidence established the value of the idea, but the software-only implementation was complex and fragile—interposing on every kernel launch, reconstructing dependency graphs, combining kernels while maintaining CUDA semantics across driver versions. On modern H200 hardware, this chapter's tested kernel pairs show $\leq$**1.0**× speedup from concurrent streams (Figure 5.6), suggesting that optimized modern kernels leave less spatial overlap for the OoO JIT mechanism to exploit in these cases. Green Contexts reduce the remaining complexity by providing hardware SM partitioning that achieves isolation without kernel-level interposition.

The historical parallels are instructive. Software RAID systems demonstrated in the 1980s that disk striping and mirroring could improve both performance and reliability; hardware RAID controllers then implemented these patterns efficiently, enabling mainstream adoption. Software virtualization—VMware's binary translation, early Xen's paravirtualization—proved that virtual machines were practical on commodity hardware; Intel VT-x and AMD-V then provided hardware extensions that made hypervisors efficient enough for production datacenters. Software network virtualization demonstrated that multi-tenant networking was achievable; SR-IOV and DPDK hardware offloads then delivered the performance necessary for cloud deployments. Even virtual memory followed this trajectory: software paging demonstrated the concept's value before hardware memory management units made it practical for operating systems to depend upon. In each case, the research contribution was demonstrating value and defining the right abstractions; hardware then implemented those abstractions efficiently. The OoO JIT work contributed to demonstrating that kernel-level GPU multiplexing deserved hardware support; Green Contexts represent that support arriving.

The tradeoff between complexity and flexibility remains real even with hardware support. OoO JIT's software combining could adapt to any workload mix—any kernel could potentially combine with any other, enabling dynamic optimization in response to run-

time conditions. Green Contexts offer less flexibility; partitions are fixed at context creation and require expensive reconfiguration to change. But for most inference workloads, this rigidity is acceptable. Workload patterns in production serving are relatively stable: the same models serve similar request distributions hour after hour, with gradual shifts over days or weeks rather than sudden changes between requests. Static partitioning sized for typical load handles these patterns well; the dynamic adaptation OoO JIT enabled was rarely necessary, while the complexity it required was always burdensome. When genuine flexibility is needed—highly variable workload mixes, rapid adaptation to changing patterns, research environments exploring new model combinations—software approaches may still be appropriate. But as hardware support improves with finer-grained partitioning and faster reconfiguration, the scope for software-only solutions steadily narrows.

Integration with Serving Systems

Kernel-level multiplexing complements rather than replaces the model-level optimizations described in earlier chapters. Chapter 4 showed how vLLM achieves near-zero memory waste through PagedAttention and 2–4× throughput improvement through continuous batching—yet GPU compute utilization remains at **20–40%** due to memory-bound decode phases, small interactive batches, and kernel launch overhead. These are exactly the gaps that kernel-level multiplexing can fill.

Consider multi-tenant vLLM deployment with Green Context partitioning. Two tenants share a single GPU, each receiving a Green Context with half the available SMs. Within each context, vLLM operates normally—PagedAttention manages KV cache memory, continuous batching maximizes request throughput, the iteration-level scheduler interleaves prefill and decode. But now when Tenant A's decode phase is memory-bound—compute sitting idle while waiting for KV cache reads from HBM—Tenant B's compute-intensive prefill phase can execute concurrently, using SMs that would otherwise be wasted. The complementary resource usage patterns of prefill (compute-intensive, memory-write) and decode (memory-read, compute-sparse) suggest that multi-tenant serving can improve on single-tenant utilization. Model-level optimization within each tenant and kernel-level partitioning across tenants address different bottlenecks; the intended integration is complementary rather than substitutive. The measurements in this chapter do not yet demonstrate that full integrated serving system; they motivate it by separating the kernel-level isolation result from the single-tenant utilization-gap result.

Deeper integration could exploit additional opportunities. LLM serving has predictable phases with different resource characteristics—prefill demands compute while decode demands memory bandwidth. A serving system aware of Green Contexts could dynamically request more SMs during prefill phases, releasing them during decode to be used by other tenants whose prefill phases are running. Priority contexts could dedicate SM partitions to latency-critical requests—premium users, time-sensitive applications—

guaranteeing resources regardless of background load while lower-priority requests share remaining capacity. Overflow handling could allow burst traffic to expand into a shared SM pool beyond guaranteed allocations, capturing efficiency during demand spikes while maintaining isolation during normal operation. These integrations require coordination between the serving scheduler, which understands request priorities and SLO requirements, and the SM allocator, which manages hardware resources. The interface between these components—what information to share, how frequently to communicate, how to make allocation decisions with minimal overhead—represents an area for future research.

Choosing a Multiplexing Mechanism

Not every deployment should reach for kernel-level techniques. **MPS** remains appropriate when processes cooperate and tail latency is secondary: it is ubiquitous, well documented, and introduces no SM-partitioning API surface. **MIG** fits strict **hardware isolation** with fixed profiles—ideal when compliance or billing requires hard walls between tenants, at the cost of inflexible partitioning. **Green Contexts** occupy a middle ground: hardware-enforced SM bounds with more flexibility than MIG profiles on some generations, but they still **share memory bandwidth**—memory-bound neighbours can disturb latency-sensitive decode. **OoO JIT** (and research systems in its family) remain relevant when workloads demand **dynamic** combining that hardware partitions cannot express, or when prototyping on hardware without modern partitioning APIs.

Multi-node serving changes the picture: kernel-level multiplexing optimizes *within* a GPU; tensor and pipeline parallelism (Chapter 4) dominate *across* GPUs. This dissertation's position is complementary: Chapter 4 maximizes efficiency **per GPU** for LLM inference; Chapter 5 fills utilization gaps **inside** that GPU when multi-tenant or multi-phase opportunities exist. Neither replaces cluster scheduling or network-aware placement, which remain necessary for global deployments and large artifacts.

Future Hardware Evolution

Each GPU generation introduces new sharing capabilities, with the trajectory clear: finer-grained partitioning, stronger isolation, lower reconfiguration overhead. NVIDIA's Blackwell architecture introduces **MLOPart (Memory Locality Optimization Partition)**, creating multiple distinct CUDA devices per physical GPU. On B200 and B300 systems, each GPU can present as two MLOPart devices with associated compute and memory resources—a step beyond MIG's fixed profiles toward more flexible hardware virtualization [142].

Current Green Contexts address compute isolation but not memory bandwidth isolation. Two contexts sharing a GPU still compete for HBM bandwidth through shared memory controllers—a limitation noted by recent research [141] and consistent with the evaluation of memory-intensive workloads. Future hardware may provide memory bandwidth

partitioning analogous to SM partitioning, dividing HBM channels among contexts to reduce this remaining interference path. The recent SGDRC work [140] demonstrates software approximations through VRAM channel colouring, suggesting that the hardware abstraction would be valuable.

Reconfiguration overhead remains a limiting factor. Current Green Context creation and destruction requires draining in-flight work and takes hundreds of microseconds to milliseconds—acceptable for infrequent adaptation but prohibitive for per-request or per-phase adjustment. Hardware support for hot reconfiguration—changing SM allocations without draining—would enable the dynamic adaptation that currently remains impractical, unlocking phase-aware allocation and fine-grained load balancing across tenants.

As hardware evolves, the systems research question shifts from "how do we achieve sharing?"—the question OoO JIT addressed through heroic software effort—to "how do we best exploit the sharing primitives hardware provides?"—the question Green Contexts begin to answer. This dissertation's contributions—demonstrating value through software prototypes, exploring hardware support through early adoption of Green Contexts—position future work to answer that question as hardware capabilities continue expanding.

5.6 Summary

This chapter presented the author's work on GPU kernel multiplexing—pushing efficiency below the model level to address the utilization gap that persists even with sophisticated serving systems. Even with vLLM's high memory efficiency and continuous batching, GPU utilization in inference can remain far below full device capacity. The sources of this underutilization—small batches constrained by latency requirements, memory-bound decode phases, kernel launch overhead, and on-chip state flushing between kernel invocations—leave the majority of GPU capacity idle even when serving systems are well-optimized at the model level.

The opportunity is substantial: on V100 hardware, a **7.7× gap** existed between naive sequential execution and optimal spatial-temporal packing of GPU kernels [38]. On modern H200 hardware, this chapter's kernel-combining microbenchmark measures $\leq$1.0× speedup across the tested pairs, but the **utilization gap persists**: reported GPU utilization stays at ∼43% even at 32 concurrent requests, leaving substantial headroom. The opportunity has shifted from kernel combining to resource partitioning—filling the utilization gap by sharing SMs across tenants rather than combining kernels within a tenant.

The author co-authored the OoO VLIW JIT compiler with Paras Jain, Ajay Jain, and collaborators at UC Berkeley, demonstrating a software-only approach to kernel-level multiplexing. By intercepting kernel launches, analyzing dependencies across concurrent tenants, and combining compatible kernels for concurrent execution, OoO JIT achieved **1.25–2.48× throughput improvements** on multi-tenant workloads. This work demonstrated the value of kernel-level multiplexing and quantified the opportunity—but also

revealed the complexity inherent in software-only solutions: interposition fragility across driver versions, correctness risks in dependency analysis, and per-kernel overhead that approached kernel execution time for small operations.

CUDA Green Contexts provide the hardware support that makes kernel-level multiplexing more practical. Rather than combining kernels in software, Green Contexts provision contexts with bounded SM resources. Measured on H200, a concurrent aggressor kernel degrades a victim by 4.2× in a shared context; with Green Context partitioning, the degradation drops to **1.00**× in the compute-bound raw Driver API microbenchmark (Figure 5.4). The victim's latency variance drops from 8,068 μs (shared) to 0.8 μs (Green Context), demonstrating that hardware SM partitioning removes the measured interference for this workload pair. The implementation is dramatically simpler than OoO JIT: Green Context creation takes ∼1.1 ms, and the benchmark observes no additional launch-path overhead beyond the execution-time effect of using fewer SMs (Figure 5.3). The guarantees are stronger: hardware enforcement rather than software approximation.

The progression from OoO JIT to Green Contexts illustrates the familiar pattern of software demonstrating value to motivate hardware support. The research contribution was showing that kernel-level multiplexing deserved attention; the hardware contribution was making it practical. This pattern—software prototype → hardware primitive → production system—positions future work to exploit increasingly sophisticated sharing mechanisms as GPU architectures evolve.

Kernel-level multiplexing complements rather than replaces the model-level optimizations described in earlier chapters. Multi-tenant vLLM with Green Context partitioning represents an integration opportunity: model-level optimizations (PagedAttention, continuous batching) within each tenant's partition, kernel-level isolation and utilization improvements across tenants. These two levels address different bottlenecks; combining them is the natural next step toward a fuller efficiency stack. The work presented in this chapter represents the frontier of the author's research on inference efficiency, pushing below the model boundary that Chapters 3 and 4 addressed to show that purpose-built abstractions can improve efficiency at every layer of the inference stack.

This chapter completed the dissertation's vertical descent through the inference stack, showing how purpose-built GPU abstractions—Out-of-Order JIT compilation and Green Context partitioning—can address utilization headroom that model-level optimizations leave on the table. Chapter 6 synthesizes this vertical arc and turns to future directions for the field.

Chapter 6

Conclusion & Future Directions

This dissertation has argued that **inference serving deserves recognition as its own area of systems research**. The evidence is empirical: across work spanning model composition (Clipper, InferLine, Ray Serve), LLM inference systems (vLLM, PagedAttention, Jenga, Pie), and GPU kernel multiplexing (OoO JIT, GreenContext), purpose-built abstractions for inference have delivered substantial improvements in the measured settings where general-purpose alternatives leave exploitable structure unused. The conceptual frame is the same one that made databases, operating systems, and networking durable systems fields: a persistent triangle among **workload**, **hardware**, and **abstraction**. When any corner changes, the system must change with it. Inference serving has now gone through several such shifts—from `/predict` endpoints, to prediction pipelines, to deployment graphs, to model-centric LLM engines—and each shift exposed new structure that generic stacks missed. The systems in this dissertation reinforce design principles that connect those shifts: expose workload structure, manage scarce accelerator resources explicitly, and make abstractions adoptable by practitioners. This chapter synthesizes the three primary technical contributions, assesses the state of the field, sketches future research directions, and reflects on what building and operating open-source systems taught about research methodology.

6.1 Summary of Contributions

The Central Insight

A single insight unifies the dissertation's technical work: **inference workloads exhibit exploitable structure at every layer of the stack, and general-purpose abstractions fail to capture that structure.** The useful mental model is a systems triangle: **workloads** define recurring patterns such as batchable classification requests, multi-stage prediction pipelines with end-to-end SLOs, autoregressive LLM sessions with growing KV state, and multi-tenant clusters with bursty demand; **hardware** defines constraints such as

CPU vector units, K80- and P100-era GPUs, A100/H100/B200-class accelerators, memory bandwidth, interconnect topology, and SM-level isolation mechanisms; and **abstractions** mediate between the two through interfaces such as `/predict`, pipelines, deployment graphs, paged KV caches, and kernel-scope partitions. The dissertation's contributions can be read as successive attempts to choose the right abstraction as both workloads and hardware changed.

That pattern appears differently at each layer. At the **model and pipeline layer**, composition patterns and end-to-end latency constraints create optimization opportunities that per-model tuning cannot see; InferLine's SLO decomposition and Ray Serve's deployment model both treat pipelines as first-class schedulable units (Chapter 3). At the **memory layer**, KV cache growth during autoregressive generation behaves like a dynamically expanding working set, and PagedAttention's virtual-memory abstraction matches that behavior in a way contiguous allocation cannot [30] (Chapter 4). At the **hardware layer**, kernel execution leaves temporal and spatial gaps—microsecond-scale idle windows and underused SMs—that kernel-level multiplexing can fill [38] (Chapter 5). General-purpose systems, designed for arbitrary computation, treat these layers as opaque; purpose-built systems expose the structure and optimize against it. The practical impact is measurable: **2–10**$\times$ throughput improvements in representative settings, large memory-efficiency gains for LLM serving, and utilization headroom that remains visible even after model-level batching.

Table 6.1 summarizes how the three contributions map abstractions to insights and outcomes.

Table 6.1: How the three primary contributions map abstractions to insights and outcomes. Each row states primary contributions; limitations appear in the corresponding chapters (e.g., Green Contexts do not partition memory bandwidth; Ray Serve trades automation for operational transparency).

Contribution	Key Abstraction	Central Insight	Impact
Model composition (Ch. 3)	Pipeline as unit	End-to-end SLOs require pipeline-level optimization, not isolated per-model tuning	Ray Serve in production; InferLine's decomposition ideas in later pipeline-serving research
vLLM & LLM serving (Ch. 4)	Paged KV cache / memory-first scheduling	The KV cache is the binding resource; treat it like memory systems, not like batch jobs	Broad adoption of vLLM and PagedAttention-class designs in open-source and industry
GPU kernel multiplexing (Ch. 5)	SM-level partitioning & kernel-scope scheduling	Exploitable structure exists below the model; software prototypes and hardware primitives (Green Contexts) converge	Demonstrated throughput gains; path to hardware-backed isolation for multi-tenant inference

Model Composition

Clipper established the vocabulary of model serving—caching, batching, model selection—showing that serving could be treated as a systems problem rather than an ad hoc wrapper around training frameworks [25]. The author contributed to the Clipper open-source project. InferLine formalized **latency-aware provisioning and SLO decomposition** for multi-stage pipelines: end-to-end latency targets decompose into per-stage budgets, and the system searches configurations that meet SLOs at minimum cost [28]. The author is a co-author on this work. Ray Serve translated these ideas into **production infrastructure**: a framework-level programming model for composing models and business logic, integrated with Ray's distributed runtime [37]. The author started and maintained Ray Serve at Anyscale, designing core abstractions.

The principle is to treat serving pipelines as **compositional, schedulable units**. End-to-end SLOs apply to complete request paths, so optimization must consider the full graph of stages rather than isolated models. The principle is also practical: users adopt systems that preserve familiar interfaces while changing what happens underneath. Ray Serve became a core component of the Ray ecosystem, deployed in settings from startups to large enterprises. InferLine's techniques influenced subsequent academic and industrial work on pipeline-aware serving. The dissertation does not claim credit for Clipper's original design or for every Ray Serve feature; it traces a coherent arc from vocabulary to formalization to deployable system.

vLLM and Memory-First LLM Serving

vLLM built PagedAttention into a production serving stack: continuous batching, efficient memory management for variable-length sequences, and an API surface that matches how users already call LLMs [30]. The author joined the project after publication and led it as Project Lead, guiding vLLM through rapid open-source growth and governance maturation while the ecosystem standardized around continuous batching and PagedAttention-class memory management. Follow-on research contributions associated with the author's tenure include Jenga for heterogeneous LLM architectures [85] and Pie for NVLink-aware memory management.

The principle is that LLM serving is memory-first. The KV cache—not raw FLOPs—is often the resource that limits batch size and throughput. Systems that treat KV cache management as a first-class memory management problem achieve results that compute-centric schedulers miss. The same end-to-end view appears here as a memory problem: optimize for the resource that actually constrains latency and throughput, not a convenient proxy. vLLM became a widely used open-source LLM serving system, referenced in production deployments and integrated with cloud and on-prem stacks. PyTorch Foundation hosting and sustained release velocity illustrate institutionalization beyond a single lab [143]. The dissertation attributes PagedAttention's algorithmic invention to the cited SOSP paper and collaborators; it describes the author's role as project leadership and sys-

tems integration, not sole invention of the algorithm. Controlled experiments on a single H200 GPU support these design choices: PagedAttention achieves 99.5% KV cache utilization versus 18–26% for static allocation (3.9–5.6× more concurrent requests), chunked prefill bounds decode interference by 3.5× during long-context prefill, and prefix caching delivers 10–16× TTFT speedup at production-relevant prefix lengths (Section 4.7).

GPU Kernel Multiplexing

The OoO VLIW JIT compiler demonstrated **kernel-level multiplexing** through software interposition: combining compatible kernels from multiple tenants for concurrent execution, exposing a large **opportunity gap** between sequential launches and optimal packing [38]. GreenContext work builds on CUDA **Green Contexts**—hardware-supported SM partitioning—to achieve isolation and sharing with less runtime complexity than continuous interposition. The author co-authored OoO JIT with Paras Jain, Ajay Jain, and collaborators; GreenContext-related evaluation reflects ongoing systems research at the hardware boundary.

The principle is that **exploitable structure exists below the model boundary**. Even within a single model's forward pass, kernels, memory phases, and SM occupancy create sharing opportunities that request-level or model-level schedulers cannot see. Kernel-level multiplexing remains a frontier: hardware interfaces, interference, and integration with frameworks like vLLM are still evolving. The dissertation positions this contribution as establishing the problem and demonstrating solutions—software-first, then hardware-backed—not as a finalized production stack for every datacenter.

6.2 The State of Inference Systems

What Has Been Solved

Several problems that were open a decade ago now have **strong baseline solutions**. Continuous batching—scheduling at iteration granularity so that short requests need not wait for long ones—has moved from research prototypes [29] into mainstream stacks; vLLM's scheduler and related systems such as TensorRT-LLM and TGI implement variants of this idea at production quality. PagedAttention and related block-based allocators show that **high KV-cache utilization** is achievable for standard transformer inference with dynamic sequence lengths [30]. The memory wall is not "solved" in the sense of unlimited context for free, but the pre-allocation waste of early systems is largely understood and avoidable. Basic serving infrastructure has also matured: organizations can deploy LLM applications without writing a custom server from scratch, and open-source systems, cloud managed endpoints, and vendor runtimes now provide a menu of options with explicit tradeoffs.

What Remains Hard

Efficiency at extreme scale. Global deployments with strict tail-latency SLOs, multi-region routing, and fault isolation still stress every layer of the stack. Consider a chatbot serving users across North America, Europe, and Asia: each region requires GPU capacity, but demand peaks shift with time zones—overprovisioning everywhere wastes capital; underprovisioning risks SLO violations during surges. Fault isolation compounds the challenge: a single bad GPU in a tensor-parallel group can stall an entire replica, and cascading retries amplify tail latency across regions. The techniques in this dissertation improve per-node efficiency; they do not eliminate the need for **careful capacity planning, cross-region load balancing, and end-to-end observability** that treats the global deployment as one system.

Multi-modal and omni-modal inference. Vision-language and audio-language models mix **different compute and memory patterns** in one request path. A vision-language model such as LLaVA processes a high-resolution image through a vision encoder (compute-heavy, fixed-length), projects it into the language model's embedding space, then generates text autoregressively (memory-bandwidth-heavy, variable-length). These stages have different optimal batch sizes, different memory footprints, and different scaling characteristics—yet current schedulers treat the entire forward pass as a single unit. Audio-language models add streaming constraints: partial audio arrives continuously while text generation must begin before the full utterance completes. Scheduling, memory abstractions, and cost models designed for text-only transformers require fundamental extension to handle these heterogeneous pipelines.

Long context. Context lengths in the **hundreds of thousands to millions of tokens**—whether for retrieval-augmented generation over large document corpora, multi-turn agent sessions, or full-codebase analysis—reintroduce memory and attention bottlenecks faster than raw hardware capacity grows [65, 64]. A single 1M-token context at FP16 precision requires approximately 30GB of KV cache for a 70B-parameter model, consuming nearly an entire A100's memory for one request. Techniques such as KV cache quantization, sparse attention, and hierarchical caching reduce the constant factors, but the fundamental quadratic-in-length attention cost and linear-in-length memory cost remain binding constraints. Efficient long-context serving—maintaining low latency while amortizing the enormous prefill cost across multiple decode steps—remains an active and high-stakes research area.

Cost optimization. Inference can dominate ML budgets because it is an ongoing operating cost rather than a one-time training expense. Better GPU utilization alone does not guarantee **lowest total cost**: a serving cluster with 90% GPU utilization but suboptimal region placement may pay more in cross-region egress than it saves in compute. Data movement, storage of model artifacts and KV caches, networking between disaggregated

components, and redundant computation across availability zones can collectively dominate bills. Inference systems therefore need **system-wide cost awareness** that jointly optimizes compute, memory, network, and storage costs rather than treating GPU utilization as a sufficient proxy.

Transferable Lessons from the Core Chapters

The core chapters reinforce themes that recur across inference systems:

- **Abstractions that hide complexity while exposing structure**—Ray Serve's deployment graph, PagedAttention's virtual KV memory, and Green Context partitioning each hide low-level machinery while preserving the structure needed for optimization.
- **End-to-end metrics over proxies**—inference systems should optimize for user-perceived latency, SLO compliance, and task success, not leaderboard throughput alone.
- **System-wide cost awareness**—compute, memory, network, and storage costs interact; local GPU utilization is necessary but not sufficient.
- **Adaptation beats static configuration**—continuous batching, preemption, chunked prefill, and SM partitioning all respond to workload dynamics that fixed knobs miss.

These lessons are not afterthoughts; they are part of the dissertation's argument that inference serving is a **systems discipline** with transferable design methodology.

6.3 Future Directions

The following directions are neither exhaustive nor independent; production systems will combine them (e.g., disaggregated **and** heterogeneous **and** multi-modal). They are useful **research lenses** because each stresses different parts of the stack. They also reflect the same workload-hardware-abstraction triangle: future inference systems will separate phases when the hardware sweet spots diverge, configure themselves when static knobs fail, and treat the GPU itself as a distributed resource when kernel-level structure matters.

Disaggregated Serving

Prefill (compute-heavy, parallel over prompt tokens) and decode (memory-bandwidth-heavy, sequential tokens) have different hardware sweet spots. Packing both phases onto identical GPUs leaves one dimension underutilized; systems such as DistServe [33] and Splitwise [98] formalize phase splitting and goodput optimization under SLOs rather than raw tokens per second alone. The open questions are system-wide rather than local:

how to serialize, compress, and pipeline KV state between prefill and decode workers, potentially across datacenter-scale deployments, without moving network and storage costs onto the critical path; how to tolerate stragglers when independently scaled phases make tail latency a distributed-systems problem; how to right-size each pool under diurnal traffic without expensive cold starts; and how to build cost models that include cross-AZ and cross-region bandwidth, since Chapter 4's single-cluster evaluation does not cover network-aware scheduling.

Adaptive and Self-Configuring Serving

The number of serving knobs has outgrown manual tuning. Batch sizes, prefill chunk sizes, KV cache block sizes, routing policies, autoscaling thresholds, speculative decoding parameters, quantization choices, and placement policies all interact with workload mix and hardware generation. A configuration that works for chat on one H100 cluster may be wrong for coding agents on B200s or for low-latency routing across regions. The research challenge is to make these systems adaptive without making them unstable: online controllers must learn from live telemetry without oscillation or SLO regressions; safe-exploration mechanisms must test new configurations under production traffic without exposing users to unacceptable tail latency; and objective functions must optimize goodput, cost, energy, and user-perceived quality jointly rather than treating tokens per second as the only target.

Heterogeneous and Specialized Hardware

NVIDIA data-center GPUs are not the only deployment target: TPUs, AMD GPUs, Intel Gaudi, AWS Trainium/Inferentia, and edge SoCs each expose different memory hierarchies, collectives, and kernel libraries. vLLM's multi-backend trajectory (Chapter 4) shows that **portability** is engineering-intensive because abstractions must hide vendor differences without hiding performance. Future systems need unified schedulers that map the same logical batch to heterogeneous devices while preserving numerical parity where required, cost-aware routing that can choose cheaper but slower device paths for best-effort traffic, and compilation pipelines that preserve dynamic shapes across IRs originally designed for static graphs.

GPU as a Distributed System

Modern accelerators increasingly look like small distributed systems: many SMs, multiple memory levels, high-bandwidth interconnects, copy engines, and specialized units execute overlapping phases of work. MegaKernel-style execution, horizontal fusion across operators, and kernel-level multiplexing all point in the same direction: the serving runtime must reason below the model boundary, and Chapter 5's OoO JIT and GreenContext

results are early examples of that shift. The open problems are to choose fusion boundaries without giving up runtime scheduling freedom, to share SMs, caches, and memory bandwidth without timing interference or unpredictable p99 latency, and to expose GPU-internal placement decisions without forcing application authors to become kernel-scheduling experts.

6.4 Reflections on Building Systems

The Tension Between Research and Production

Research rewards novelty; production rewards reliability. The dissertation's trajectory includes both research prototypes and systems that organizations deploy. The productive pattern is not "research first, production second" in isolation, but **iterative translation**: prototypes clarify ideas; production hardens them; the next research cycle incorporates what production revealed.

Open Source as Research Methodology

Open-source release is not merely dissemination. It is a **method for stress-testing claims**: if an abstraction does not help real users, adoption and contributions stall. vLLM's growth and external contributions, along with Ray Serve's production use, provide evidence that these serving abstractions met a widespread need. Benchmarks matter; **adoption and maintenance** are additional evidence. This is also where a product mindset becomes part of systems research rather than a distraction from it. Good abstractions are not only fast; they are understandable, installable, debuggable, and compatible with the workflows users already have. The path from Clipper's `/predict` interface to Ray Serve's deployment graph to vLLM's OpenAI-compatible API reflects the same lesson: the right interface lowers adoption barriers while preserving enough structure for the system to optimize. Solving the abstract M×N compatibility problem—many models across many accelerators—is not sufficient by itself. Impact requires practical and novel solutions grounded in deep workload understanding, sustained maintenance, and a community small enough that users, contributors, researchers, and hardware vendors repeatedly meet around the same bottlenecks.

Retrospective: What Would Be Done Differently

Every long research trajectory involves missteps, and three retrospective priorities stand out. First, production hardening should shape system architecture from the beginning: prototypes that never reach users teach less than prototypes that someone deploys, and **shipping** forces discipline around APIs, migrations, and backward compatibility. Second, code is necessary but insufficient; documentation, contributor onboarding, and gover-

nance determine whether a project survives leadership transitions, and vLLM's PyTorch Foundation milestone is an example of **institutionalizing** sustainability. Third, inference systems sit between **ML, hardware, and software engineering**, so earlier collaboration with ML researchers on workload models and with hardware partners on emerging primitives would have accelerated some design decisions.

6.5 Closing Remarks

The Field Is Maturing but Still Young

The period from roughly 2020 to 2025 transformed LLM serving from a niche deployment concern into **critical infrastructure**. Yet compared against databases, operating systems, or networking, inference serving is **young**. The canonical textbooks, reference architectures, and settled tradeoffs are still being written. That is an opportunity for researchers entering the field.

Open Source and Community

Ray Serve, vLLM, and related projects succeeded because **communities** formed around them—users filing issues, vendors integrating, researchers extending. The dissertation's technical contributions are inseparable from that social context. **Governance and maintenance** are as much infrastructure as schedulers and allocators. Open source made the systems legible to the field: users could compare performance claims against their own workloads, vendors could add support for new hardware, and researchers could build follow-on ideas without rebuilding the stack from scratch. That social feedback loop is one reason inference serving matured quickly once LLM workloads made the bottleneck obvious.

Final Thought

Ten years ago, inference could be treated as a post-hoc wrapping exercise around a trained model. Today, inference cost shapes hardware roadmaps, product decisions, and research agendas, and this dissertation has argued that inference serving deserves recognition as its own systems area. Its contributions span **model-level abstractions** (Chapter 3), **memory-first LLM serving** (Chapter 4), and **kernel-level multiplexing** (Chapter 5), but the central insight is the same across all three: **inference workloads expose structure at every layer**, and purpose-built abstractions capture that structure at pipeline, memory, and SM granularity in ways general-purpose stacks miss. The triangle will keep moving: workloads will shift toward agents and multimodal interaction, hardware will shift toward heterogeneous accelerators and richer partitioning primitives, and abstractions will need to shift from static serving endpoints toward adaptive systems that configure them-

selves. Model architectures, hardware, and applications will change; the methodology of finding the right abstraction will not. A decade from now, the specific systems named here may be superseded; the **discipline**—evidence-backed serving stacks, honest limitation analysis, and interfaces that practitioners can adopt—should remain a benchmark against which new work is judged.

This dissertation has argued that inference serving is its own systems discipline—one that rewards purpose-built abstractions at every layer of the stack. From pipeline orchestration to paged memory to SM partitioning, the methodology is consistent: understand the workload, respect the hardware, build the right abstraction, and test it in production. The systems will evolve; the discipline endures.

Glossary of Key Terms

This glossary collects the key terms used throughout this dissertation. Terms are grouped by the chapter in which they first appear and listed in order of occurrence.

Chapter 1: Introduction

Inference serving (vs. training) Running trained ML models to produce predictions for incoming requests, as opposed to training models on data. *Ch. 1, also Ch. 2*

SLO (Service Level Objective) A latency target, typically expressed as a percentile (e.g., p99 latency < 100 ms means 99% of requests must complete within 100 ms) [35]. *Ch. 1, also Chs. 2, 3*

Throughput Requests processed per unit time (requests/second) or, for LLMs, tokens generated per unit time (tokens/second). *Ch. 1, also Ch. 2*

Latency Time from request arrival to response completion; for LLMs, often decomposed into TTFT (time to first token) and ITL (inter-token latency). *Ch. 1, also Ch. 2*

LLM (Large Language Model) Neural network models with billions of parameters trained on text data, capable of generating coherent text responses. *Ch. 1*

KV cache Key-value tensors cached during autoregressive generation to avoid recomputing attention over previous tokens [30]. *Ch. 1, also Chs. 2, 4*

Continuous batching Iteration-level scheduling where requests can join and leave batches dynamically, rather than waiting for all requests in a batch to complete [29]. *Ch. 1, also Ch. 2*

PagedAttention Virtual memory abstraction for KV caches that eliminates memory fragmentation by storing KV cache in non-contiguous blocks [30]. *Ch. 1, also Chs. 2, 4*

FlashAttention IO-aware attention algorithm that avoids materializing the full attention matrix [31]. *Ch. 1*

Chapter 2: Background

Model server Software layer that orchestrates inference execution, handling batching, scheduling, and memory management. *Ch. 2*

Static batching Processing requests in fixed batches that complete together—all requests wait for the slowest. *Ch. 2, also Ch. 4*

Chapter 3: Model Composition

Model composition Serving multiple models on shared infrastructure while meeting diverse SLOs. *Ch. 3*

Statistical multiplexing Exploiting non-aligned peaks across workloads to improve utilization. *Ch. 3*

SLO decomposition Allocating end-to-end latency budget across pipeline stages to minimize cost. *Ch. 3*

Adaptive batching Dynamically adjusting batch size based on load and latency targets. *Ch. 3*

Model selection Routing queries to model variants based on accuracy–latency requirements. *Ch. 3*

Deployment A model or processing function with specified resource allocation (Ray Serve). *Ch. 3*

Deployment graph A DAG of deployments representing a multi-model pipeline (Ray Serve). *Ch. 3*

Profiler-planner-controller Architecture pattern for SLO-aware pipeline optimization (InferLine). *Ch. 3*

Squishy bin packing Bin packing variant where item sizes vary with batching decisions (Nexus). *Ch. 3*

Power of Two Choices Load balancing algorithm that samples two replicas and routes to the less loaded one; achieves $O(\log \log n)$ maximum queue length (Ray Serve). *Ch. 3*

Target ongoing requests Queue-depth autoscaling metric; desired concurrent requests per replica (Ray Serve). *Ch. 3*

GCS (Global Control Store) Ray's distributed metadata store; enables fault tolerance through state checkpointing. *Ch. 3*

Chapter 4: vLLM

Autoregressive generation Generating text one token at a time, sampling each token from a distribution conditioned on every preceding token—both the input prompt and all previously generated output. *Ch. 4*

Arithmetic intensity The ratio of FLOPS to bytes of memory traffic. Low during LLM decode, placing operations in the memory-bandwidth-limited regime on the roofline model. *Ch. 4*

Prefill phase Processing the entire input prompt in a single forward pass, populating the KV cache. Compute-intensive and resembles traditional inference. *Ch. 4, also Ch. 6*

Decode phase Each forward pass produces one new token and updates the KV cache, repeating until an end-of-sequence token or length limit. Memory-bandwidth-bound. *Ch. 4, also Ch. 6*

TTFT (Time to First Token) Time from request arrival until the first output token is generated; dominated by prefill latency. *Ch. 4, also Chs. 1, 6*

ITL (Inter-Token Latency) Time between consecutive output tokens during decode; determines perceived streaming smoothness. *Ch. 4, also Chs. 1, 6*

Copy-on-Write (COW) Memory sharing technique where parallel sequences reference the same physical KV cache blocks for shared prefixes; a block is duplicated only when a sequence diverges. *Ch. 4*

Chunked prefill Breaking long prefills into smaller chunks (typically 512 tokens), interleaving decode iterations between chunks to prevent head-of-line blocking from long prompts. *Ch. 4*

Prefix caching Automatic detection and reuse of shared token prefixes across requests. Hashes token prefixes at block granularity; matching blocks skip prefill entirely. *Ch. 4*

Speculative decoding Breaking the sequential decode bottleneck by using a small draft model to propose candidate tokens, then verifying them in a single target-model forward pass [32]. Theoretically lossless via rejection sampling. *Ch. 4, also Ch. 6*

Quantization Reducing numerical precision of model weights or KV cache (e.g., FP16 to INT8 or INT4) to shrink memory footprint and increase effective memory bandwidth. *Ch. 4*

Tensor parallelism Distributing a model by sharding each layer's weight matrices across GPUs, so every GPU participates in every layer's computation; partial results are aggregated via all-reduce. *Ch. 4*

Jenga PagedAttention extension for hybrid LLM architectures (e.g., models mixing full attention, sliding-window attention, GQA, and SSM layers). Uses a two-level memory allocator with per-layer caching policies [85]. *Ch. 4*

Pie NVLink-aware memory management that treats CPU memory as a fast secondary tier on Grace Hopper hardware (900 GB/s NVLink vs. 64 GB/s PCIe), expanding the effective KV cache pool by roughly 8×. *Ch. 4*

Chapter 5: GPU Kernel Multiplexing

Streaming Multiprocessor (SM) The fundamental compute unit of NVIDIA GPUs. Each SM contains multiple CUDA cores and Tensor Cores and executes thread blocks independently. The A100 has 108 SMs; the H100 has 132. *Ch. 5*

CUDA Stream A sequence of GPU operations that execute in order. Operations in different streams can execute concurrently if resources permit. *Ch. 5*

Kernel Launch Overhead The software latency between initiating a kernel launch and the kernel beginning execution on the GPU. Typically 2–20 microseconds; reduced to sub-microsecond with CUDA Graphs. *Ch. 5*

MPS (Multi-Process Service) NVIDIA software enabling GPU sharing across OS processes. Provides concurrent kernel execution but no resource isolation guarantees. *Ch. 5*

MIG (Multi-Instance GPU) Hardware partitioning on A100/H100 GPUs creating isolated instances with dedicated SMs, memory, and cache. Limited to 7 instances with fixed profiles. *Ch. 5*

Green Context Lightweight CUDA context with hardware-enforced SM limitations, introduced in CUDA 12.4. Enables fine-grained GPU partitioning without the rigidity of MIG. *Ch. 5*

Spatial Multiplexing Partitioning GPU resources (SMs, memory) among concurrent workloads so each has dedicated resources. *Ch. 5*

Temporal Multiplexing Time-slicing GPU access among workloads, with each getting exclusive access for a time quantum. *Ch. 5*

Opportunity Gap The difference between current utilization and theoretical maximum. For kernel-level multiplexing, the 7.7× gap represents headroom for improvement through better resource sharing. *Ch. 5*

Chapter 6: Conclusion & Future Directions

Disaggregated serving Separating prefill and decode phases onto different hardware pools, each sized for its dominant bottleneck (compute for prefill, memory bandwidth for decode) [33]. *Ch. 6*

Goodput SLO-compliant throughput: the fraction of requests that meet latency targets, as distinct from raw tokens per second. *Ch. 6*

References

[1] Edsger W. Dijkstra. “The structure of the “THE”-multiprogramming system”. In: *Communications of the ACM* 11.5 (1968), pp. 341–346. DOI: 10.1145/363095.363143. URL: https://doi.org/10.1145/363095.363143.

[2] Dennis M. Ritchie and Ken Thompson. “The UNIX Time-Sharing System”. In: *Communications of the ACM* 17.7 (1974), pp. 365–375. DOI: 10.1145/361011.361061. URL: https://doi.org/10.1145/361011.361061.

[3] Peter J. Denning. “Virtual Memory”. In: *ACM Computing Surveys* 2.3 (1970), pp. 153–189. DOI: 10.1145/356571.356573. URL: https://doi.org/10.1145/356571.356573.

[4] Edgar F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Communications of the ACM* 13.6 (1970), pp. 377–387. DOI: 10.1145/362384.362685. URL: https://doi.org/10.1145/362384.362685.

[5] Morton M. Astrahan et al. “System R: Relational Approach to Database Management”. In: *ACM Transactions on Database Systems* 1.2 (1976), pp. 97–137. DOI: 10.1145/320455.320457. URL: https://doi.org/10.1145/320455.320457.

[6] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. “Access Path Selection in a Relational Database Management System”. In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. 1979, pp. 23–34. DOI: 10.1145/582095.582099. URL: https://doi.org/10.1145/582095.582099.

[7] Jim Gray. “The Transaction Concept: Virtues and Limitations”. In: *Proceedings of the Seventh International Conference on Very Large Data Bases*. Invited paper. Cannes, France: IEEE Computer Society, 1981, pp. 144–154. URL: https://www.sigmod.org/publications/dblp/db/conf/vldb/Gray81.html.

[8] Vinton G. Cerf and Robert E. Kahn. “A Protocol for Packet Network Intercommunication”. In: *IEEE Transactions on Communications* 22.5 (1974), pp. 637–648. DOI: 10.1109/TCOM.1974.1092259. URL: https://doi.org/10.1109/TCOM.1974.1092259.

[9] Jon Postel. *Transmission Control Protocol*. RFC 793. Sept. 1981. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/rfc/rfc793.

[10] Jerome H. Saltzer, David P. Reed, and David D. Clark. "End-to-End Arguments in System Design". In: *ACM Transactions on Computer Systems* 2.4 (1984), pp. 277–288. DOI: 10.1145/357401.357402. URL: https://doi.org/10.1145/357401.357402.

[11] Samuel J. Leffler, William N. Joy, and Robert S. Fabry. *4.2BSD Networking Implementation Notes (Revised July, 1983)*. Tech. rep. UCB/CSD-83-146. EECS Department, University of California, Berkeley, July 1983. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1983/5453.html.

[12] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. "A Fast File System for UNIX". In: *ACM Transactions on Computer Systems* 2.3 (1984), pp. 181–197. DOI: 10.1145/989.990. URL: https://doi.org/10.1145/989.990.

[13] Mendel Rosenblum and John K. Ousterhout. "The Design and Implementation of a Log-Structured File System". In: *ACM Transactions on Computer Systems* 10.1 (1992), pp. 26–52. DOI: 10.1145/146941.146943. URL: https://doi.org/10.1145/146941.146943.

[14] Michael Stonebraker and Lawrence A. Rowe. "The Design of POSTGRES". In: *ACM SIGMOD Record* 15.2 (1986), pp. 340–355. DOI: 10.1145/16856.16888. URL: https://doi.org/10.1145/16856.16888.

[15] David Patterson, Joseph Gonzalez, Urs Hölzle, Quoc Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. "The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink". In: *Computer* 55.7 (2022), pp. 18–28. DOI: 10.1109/MC.2022.3148714. URL: https://doi.org/10.1109/MC.2022.3148714.

[16] Bryan Bamford. *AI Inference Costs in 2025: The $255B Market's Energy Crisis and Path to Sustainable Scaling*. Tensormesh Blog. Dec. 2025. URL: https://www.tensormesh.ai/blog-posts/ai-inference-costs-2025-energy-crisis.

[17] Nestor Maslej, Loredana Fattorini, Raymond Perrault, Yolanda Gil, Vanessa Parli, et al. *Artificial Intelligence Index Report 2025*. Tech. rep. AI Index Steering Committee, Institute for Human-Centered AI, Stanford University, Apr. 2025. DOI: 10.48550/arXiv.2504.07139. URL: https://hai.stanford.edu/ai-index/2025-ai-index-report.

[18] Ed Zitron. *Exclusive: Here's How Much OpenAI Spends On Inference and Its Revenue Share With Microsoft*. Where's Your Ed At. Leaked-document report; covered by TechCrunch and The Register. Nov. 2025. URL: https://www.wheresyoured.at/oai_docs/.

[19] Kim Hazelwood et al. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629. DOI: 10.1109/HPCA.2018.00059. URL: https://doi.org/10.1109/HPCA.2018.00059.

[20] Jongsoo Park et al. "Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications". In: *arXiv preprint arXiv:1811.09886* (2018). DOI: 10.48550/arXiv.1811.09886. eprint: 1811.09886. URL: https://arxiv.org/abs/1811.09886.

[21] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. "Hidden Technical Debt in Machine Learning Systems". In: *Advances in Neural Information Processing Systems*. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper_files/paper/2015/hash/86df7dcfd896fcaf2674f757a2463eba-Abstract.html.

[22] Erick Brethenoux and Anthony Mullen. *Survey Analysis: The Most Successful AI Implementations Require Discipline, not Ph.D.s*. Gartner Research. Published August 26, 2022; based on the 2021 Gartner AI in Organizations Survey. Aug. 2022. URL: https://www.gartner.com/en/documents/4018048.

[23] Yoshiaki Inoue. "Queueing Analysis of GPU-Based Inference Servers with Dynamic Batching: A Closed-Form Characterization". In: *Performance Evaluation* 147 (2021), p. 102183. DOI: 10.1016/j.peva.2020.102183. URL: https://doi.org/10.1016/j.peva.2020.102183.

[24] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. "TensorFlow-Serving: Flexible, High-Performance ML Serving". In: *NIPS Workshop on ML Systems*. 2017. DOI: 10.48550/arXiv.1712.06139. URL: https://arxiv.org/abs/1712.06139.

[25] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. "Clipper: A Low-Latency Online Prediction Serving System". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Mar. 2017, pp. 613–627. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw.

[26] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. "Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis". In: *Proceedings of SOSP*. 2019, pp. 322–337. DOI: 10.1145/3341301.3359658. URL: https://doi.org/10.1145/3341301.3359658.

[27] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 443–462. URL: https://www.usenix.org/conference/osdi20/presentation/gujarati.

[28] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph E. Gonzalez, and Alexey Tumanov. "InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines". In: *ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery, 2020, pp. 477–491. DOI: 10.1145/3419111.3421285. URL: https://doi.org/10.1145/3419111.3421285.

[29] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. "Orca: A Distributed Serving System for Transformer-Based Generative Models". In: *Proceedings of OSDI*. 2022.

[30] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of SOSP*. 2023. DOI: 10.1145/3600006.3613165. URL: https://doi.org/10.1145/3600006.3613165.

[31] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2022.

[32] Yaniv Leviathan, Matan Kalman, and Yossi Matias. "Fast Inference from Transformers via Speculative Decoding". In: *Proceedings of ICML*. 2023.

[33] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. "DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving". In: *Proceedings of OSDI*. 2024.

[34] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. "Machine Learning Operations (MLOps): Overview, Definition, and Architecture". In: *IEEE Access* 11 (2023), pp. 31866–31879. DOI: 10.1109/ACCESS.2023.3262138. URL: https://doi.org/10.1109/ACCESS.2023.3262138.

[35] Jeffrey Dean and Luiz André Barroso. "The Tail at Scale". In: *Communications of the ACM* 56.2 (2013), pp. 74–80. DOI: 10.1145/2408776.2408794. URL: https://doi.org/10.1145/2408776.2408794.

[36] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, July 2019, pp. 947–960. URL: https://www.usenix.org/conference/atc19/presentation/jeon.

[37] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Oct. 2018, pp. 561–577. URL: https://www.usenix.org/conference/osdi18/presentation/moritz.

[38] Paras Jain, Xiangxi Mo, Ajay Jain, Alexey Tumanov, Joseph E. Gonzalez, and Ion Stoica. "The OoO VLIW JIT Compiler for GPU Inference". In: *arXiv preprint arXiv:1901.10008* (2019).

[39] Thomas G. Dietterich. "Ensemble Methods in Machine Learning". In: *Multiple Classifier Systems*. Vol. 1857. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 1–15. DOI: 10.1007/3-540-45014-9_1. URL: https://doi.org/10.1007/3-540-45014-9_1.

[40] NVIDIA. *FasterTransformer*. GitHub. GitHub repository. 2021. URL: https://github.com/NVIDIA/FasterTransformer (visited on 05/12/2026).

[41] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. "Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 2021, pp. 1–18. URL: https://www.usenix.org/conference/osdi21/presentation/qiao.

[42] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[43] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[44] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: https://github.com/jax-ml/jax.

[45] NVIDIA. *NVIDIA TensorRT*. NVIDIA Developer. 2019. URL: https://developer.nvidia.com/tensorrt (visited on 05/12/2026).

[46] NVIDIA. *Triton Inference Server Release 20.03*. NVIDIA Deep Learning Triton Inference Server Documentation. 2020. URL: https://docs.nvidia.com/deeplearning/triton-inference-server/archives/triton-inference-server-2450/release-notes/rel_20-03.html.

[47] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. "MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, July 2019, pp. 1049–1062. URL: https://www.usenix.org/conference/atc19/presentation/zhang-chengliang.

[48] Rico Sennrich, Barry Haddow, and Alexandra Birch. "Neural Machine Translation of Rare Words with Subword Units". In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2016, pp. 1715–1725. DOI: 10.18653/v1/P16-1162.

[49] Taku Kudo and John Richardson. "SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing". In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 2018, pp. 66–71. DOI: 10.18653/v1/D18-2012.

[50] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[51] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12.85 (2011), pp. 2825–2830. URL: https://www.jmlr.org/papers/v12/pedregosa11a.html.

[52] Alex Guazzelli, Michael Zeller, Wen-Ching Lin, and Graham Williams. "PMML: An Open Standard for Sharing Models". In: *The R Journal* 1.1 (2009), pp. 60–65. DOI: 10.32614/RJ-2009-010.

[53] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[54] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. "Scaling Distributed Machine Learning with the Parameter Server". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[55] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[56] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. "BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 972–986. DOI: 10.1109/SC41405.2020.00073. URL: https://doi.org/10.1109/SC41405.2020.00073.

[57] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. "PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Oct. 2018, pp. 611–626. URL: https://www.usenix.org/conference/osdi18/presentation/lee.

[58] Wencong Xiao et al. "Gandiva: Introspective Cluster Scheduling for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Oct. 2018, pp. 595–610. URL: https://www.usenix.org/conference/osdi18/presentation/xiao.

[59] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. "Tiresias: A GPU Cluster Manager for Distributed Deep Learning". In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Feb. 2019, pp. 485–500. URL: https://www.usenix.org/conference/nsdi19/presentation/gu.

[60] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. "INFaaS: Automated Model-less Inference Serving". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 397–411. URL: https://www.usenix.org/conference/atc21/presentation/romero.

[61] Zhuohan Li et al. "AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving". In: *Proceedings of OSDI*. 2023.

[62] Tri Dao. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning". In: *The Twelfth International Conference on Learning Representations*. 2024. arXiv: 2307.08691. URL: https://openreview.net/forum?id=mZn2Xyh9Ec.

[63] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. "Accelerating Large Language Model Decoding with Speculative Sampling". In: *arXiv preprint arXiv:2302.01318* (2023).

[64] Zhenyu Zhang et al. "H2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023. DOI: 10.48550/arXiv.2306.14048. eprint: 2306.14048. URL: https://papers.nips.cc/paper_files/paper/2023/hash/6ceefa7b15572587b78ecfcebb2827f8-Abstract-Conference.html.

[65] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. "KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 37. 2024, pp. 1270–1303. DOI: 10.52202/079017-0040.

[66] Gartner. *Gartner Says AI-Optimized IaaS Is Poised to Become the Next Growth Engine for AI Infrastructure*. Gartner Press Release. Oct. 2025. URL: https://www.gartner.com/en/newsroom/press-releases/2025-10-15-gartner-says-artificial-intelligence-optimized-iaas-is-poised-to-become-the-next-growth-engine-for-artificial-intelligence-infrastructure.

[67] Sam Altman. *Three Observations*. Blog post. Characterizes inference cost decline as approximately 10x per year. Feb. 2025.

[68] Guido Appenzeller. *Welcome to LLMflation: LLM Inference Cost Is Going Down Fast*. a16z. Documents 1000x inference cost decline over three years (2021–2024). Nov. 2024. URL: https://a16z.com/llmflation-llm-inference-cost/.

[69] DeepSeek-AI. *DeepSeek-V3 Technical Report*. arXiv preprint arXiv:2412.19437. Dec. 2024. DOI: 10.48550/arXiv.2412.19437. arXiv: 2412.19437. URL: https://arxiv.org/abs/2412.19437.

[70] Gartner. *Gartner Predicts That by 2030, Performing Inference on an LLM with 1 Trillion Parameters Will Cost GenAI Providers Over 90 Percent Less Than in 2025*. Gartner Press Release. Published March 25, 2026. Mar. 2026. URL: https://www.gartner.com/en/newsroom/press-releases/2026-03-25-gartner-predicts-that-by-2030-performing-inference-on-an-llm-with-1-trillion-parameters-will-cost-genai-providers-over-90-percent-less-than-in-2025.

[71] Kotikalapudi Sriram and Ward Whitt. "Characterizing Superposition Arrival Processes in Packet Multiplexers for Voice and Data". In: *IEEE Journal on Selected Areas in Communications* 4.6 (1986), pp. 833–846. DOI: 10.1109/JSAC.1986.1146402.

[72] Simon Mo, Edward Oakes, and Michael Galarnyk. *Serving ML Models in Production: Common Patterns*. Anyscale Blog. 2021. URL: https://www.anyscale.com/blog/serving-ml-models-in-production-common-patterns.

[73] AWS. *Amazon SageMaker Inference Pipelines*. AWS Documentation. 2023.

[74] Akshay Malik, Edward Oakes, and Phi Nguyen. *Ray Serve: Tackling the Cost and Complexity of Serving AI in Production*. Anyscale Blog. 2023. URL: https://www.anyscale.com/blog/tackling-the-cost-and-complexity-of-serving-ai-in-production-ray-serve.

[75] NVIDIA. *NVIDIA Triton Inference Server*. NVIDIA Documentation. 2023. URL: https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html (visited on 05/12/2026).

[76] Saeid Ghafouri, Kamran Razavi, Mehran Salmani, Alireza Sanaee, Tania Lorido-Botran, Lin Wang, Joseph Doyle, and Pooyan Jamshidi. "IPA: Inference Pipeline Adaptation to Achieve High Accuracy and Cost-Efficiency". In: *Journal of Systems Research* 4.1 (2024). DOI: 10.5070/SR34163500. URL: https://www.jsys.org/bibliography/2024-04-15-ghafouriSolution.html.

[77] Sohaib Ahmad, Hui Guan, and Ramesh K. Sitaraman. "Loki: A System for Serving ML Inference Pipelines with Hardware and Accuracy Scaling". In: *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*. 2024, pp. 267–280. DOI: 10.1145/3625549.3658688. URL: https://arxiv.org/abs/2407.03583.

[78] Michael Mitzenmacher. "The Power of Two Choices in Randomized Load Balancing". PhD thesis. University of California, Berkeley, 1996.

[79] Anyscale. *Ray Summit 2024: Breaking Through the AI Complexity Wall*. Anyscale Blog. 2024. URL: https://www.anyscale.com/blog/ray-summit-2024-recap.

[80] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. "MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving". In: *Proceedings of the 41st International Conference on Machine Learning*. Vol. 235. Proceedings of Machine Learning Research. PMLR, 2024, pp. 11905–11917. URL: https://proceedings.mlr.press/v235/duan24a.html.

[81] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[82] Hugging Face. *Text Generation Inference*. GitHub. 2022. URL: https://github.com/huggingface/text-generation-inference (visited on 05/04/2026).

[83] Cade Daniel, Chen Shen, Eric Liang, and Richard Liaw. *How continuous batching enables 23x throughput in LLM inference while reducing p50 latency*. Anyscale Blog. June 2023. URL: https://www.anyscale.com/blog/continuous-batching-llm-inference.

[84] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. "FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-Precision". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2024.

[85] Chen Zhang et al. "Jenga: Effective Memory Management for Serving LLM with Heterogeneity". In: *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. ACM, 2025, pp. 446–461. DOI: 10.1145/3731569.3764823. URL: https://doi.org/10.1145/3731569.3764823.

[86] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, et al. "Mistral 7B". In: *arXiv preprint arXiv:2310.06825* (2023). DOI: 10.48550/arXiv.2310.06825. eprint: 2310.06825. URL: https://arxiv.org/abs/2310.06825.

[87] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints". In: *Proceedings of EMNLP*. 2023.

[88] vLLM Project. *vLLM Documentation*. vLLM Documentation. 2026. URL: https://docs.vllm.ai/en/v0.20.0/ (visited on 05/04/2026).

[89] vLLM Project. *Speculative Decoding*. vLLM Documentation. 2026. URL: https://docs.vllm.ai/en/v0.20.1/features/speculative_decoding/ (visited on 05/04/2026).

[90] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. "EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty". In: *Proceedings of the 41st International Conference on Machine Learning*. Vol. 235. Proceedings of Machine Learning Research. PMLR, 2024, pp. 28935–28948. arXiv: 2401.15077. URL: https://proceedings.mlr.press/v235/li24bt.html.

[91] Amazon and NVIDIA Team. *P-EAGLE: Faster LLM Inference with Parallel Speculative Decoding in vLLM*. vLLM Blog. 2026. URL: https://vllm.ai/blog/p-eagle (visited on 05/04/2026).

[92] Paulius Micikevicius et al. "FP8 Formats for Deep Learning". In: *arXiv preprint arXiv:2209.05433* (2022). DOI: 10.48550/arXiv.2209.05433. eprint: 2209.05433. URL: https://arxiv.org/abs/2209.05433.

[93] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. "AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration". In: *Proceedings of Machine Learning and Systems*. Vol. 6. 2024. arXiv: 2306.00978. URL: https://proceedings.mlsys.org/paper_files/paper/2024/hash/42a452cbafa9dd64e9ba4aa95cc1ef21-Abstract-Conference.html.

[94] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. "GPTQ: Accurate Post-Training Quantization for Generative Pre-Trained Transformers". In: *Proceedings of ICLR*. 2023. arXiv: 2210.17323. URL: https://openreview.net/forum?id=tcbBPnfwxS.

[95] Jonas Kübler, Eldar Kurtić, Lucas Wilkinson, Matthew Bonanni, Michael Goin, Alexandre Marques, and Kailash Budhathoki. *The State of FP8 KV-Cache and Attention Quantization in vLLM*. vLLM Blog. Apr. 2026. URL: https://vllm.ai/blog/fp8-kvcache (visited on 05/04/2026).

[96] Zihao Ye et al. "FlashInfer: Efficient and Customizable Attention Engine for LLM Inference Serving". In: *Proceedings of Machine Learning and Systems*. 2025. DOI: 10.48550/arXiv.2501.01005. eprint: 2501.01005. URL: https://arxiv.org/abs/2501.01005.

[97] vLLM Project. *Attention Backend Feature Support*. vLLM Documentation. 2026. URL: https://docs.vllm.ai/en/v0.20.0/design/attention_backends/ (visited on 05/04/2026).

[98] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. "Splitwise: Efficient Generative LLM Inference Using Phase Splitting". In: *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 2024, pp. 118–132. DOI: 10.1109/ISCA59077.2024.00019. URL: https://doi.org/10.1109/ISCA59077.2024.00019.

[99] vLLM Project. *Disaggregated Prefilling (experimental)*. vLLM Documentation. 2026. URL: https://docs.vllm.ai/en/v0.20.1/features/disagg_prefill/ (visited on 05/04/2026).

[100] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. *vLLM: Easy, Fast, and Cheap LLM Serving with PagedAttention*. vLLM Blog. Blog post. June 2023. URL: https://blog.vllm.ai/2023/06/20/vllm.html.

[101] PyTorch Foundation. *PyTorch Foundation Expands to Umbrella Foundation and Welcomes vLLM and DeepSpeed Projects*. PyTorch Blog. 2025. URL: https://pytorch.org/blog/press-release-pytorch-foundation-expands-welcomes-projects-vllm-deepspeed/ (visited on 05/04/2026).

[102] vLLM Project. *vLLM Repository Metadata*. GitHub repository metadata. 2026. URL: https://github.com/vllm-project/vllm (visited on 05/04/2026).

[103] vLLM Team. *Announcing Llama 3.1 Support in vLLM*. vLLM Blog. 2024. URL: https://vllm.ai/blog/llama31.

[104] vLLM Team. *vLLM's Open Governance and Performance Roadmap*. vLLM Blog. 2024. URL: https://vllm.ai/blog/lfai-perf.

[105] vLLM Team. *vLLM v0.6.0: 2.7x Throughput Improvement and 5x Latency Reduction*. vLLM Blog. 2024. URL: https://vllm.ai/blog/perf-update.

[106] Red Hat. *Red Hat Announces Definitive Agreement to Acquire Neural Magic*. Red Hat Press Release. 2024. URL: https://www.redhat.com/en/about/press-releases/red-hat-acquire-neural-magic.

[107] Anupam Singh and Karun Channa. *Running AI Inference at Scale in the Hybrid Cloud*. Roblox Engineering Blog. Sept. 2024. URL: https://about.roblox.com/newsroom/2024/09/running-ai-inference-at-scale-in-the-hybrid-cloud/.

[108] vLLM Team. *vLLM V1: A Major Upgrade to vLLM's Core Architecture*. vLLM Blog. 2025. URL: https://vllm.ai/blog/v1-alpha-release (visited on 05/04/2026).

[109] Robert Shaw and Thameem Abbas Ibrahim Bathusha. *Performance boosts in vLLM 0.8.1: Switching to the V1 engine*. Red Hat Developer. 2025. URL: https://developers.redhat.com/articles/2025/04/28/performance-boosts-vllm-081-switching-v1-engine (visited on 05/04/2026).

[110] Woosuk Kwon. *[RFC]: Deprecating vLLM V0*. GitHub Issue #18571. 2025. URL: https://github.com/vllm-project/vllm/issues/18571 (visited on 05/04/2026).

[111] Marina Temkin. *Inference Startup Inferact Lands $150M to Commercialize vLLM*. TechCrunch. 2026. URL: https://techcrunch.com/2026/01/22/inference-startup-inferact-lands-150m-to-commercialize-vllm/ (visited on 05/04/2026).

[112] Noam Shazeer. "Fast Transformer Decoding: One Write-Head is All You Need". In: *arXiv preprint arXiv:1911.02150* (2019).

[113] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, et al. "Mixtral of Experts". In: *arXiv preprint arXiv:2401.04088* (2024). DOI: 10.48550/arXiv.2401.04088. eprint: 2401.04088. URL: https://arxiv.org/abs/2401.04088.

[114] Barak Lenz, Opher Lieber, Alan Arazi, et al. "Jamba: Hybrid Transformer-Mamba Language Models". In: *International Conference on Learning Representations*. 2025. URL: https://proceedings.iclr.cc/paper_files/paper/2025/hash/a9ed43fa31dc8b4a7d7a673d713dcb5f-Abstract-Conference.html.

[115] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. "Visual Instruction Tuning". In: *Advances in Neural Information Processing Systems*. Vol. 36. 2023. arXiv: 2304.08485. URL: https://papers.nips.cc/paper_files/paper/2023/hash/6dcf277ea32ce3288914faf369fe6de0-Abstract-Conference.html.

[116] Daya Guo, Dejian Yang, Haowei Zhang, et al. "DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning". In: *Nature* 645 (2025), pp. 633–638. DOI: 10.1038/s41586-025-09422-z. URL: https://www.nature.com/articles/s41586-025-09422-z.

[117] DeepSeek-AI. "DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model". In: *arXiv preprint arXiv:2405.04434* (2024). DOI: 10.48550/arXiv.2405.04434. eprint: 2405.04434. URL: https://arxiv.org/abs/2405.04434.

[118] vLLM Team. *SemiAnalysis InferenceMAX: vLLM and NVIDIA Accelerate Blackwell Inference*. vLLM Blog. 2025. URL: https://vllm.ai/blog/blackwell-inferencemax.

[119] Hugging Face. *TGI v3 overview*. Hugging Face Documentation. 2024. URL: https://huggingface.co/docs/text-generation-inference/en/conceptual/chunking (visited on 05/04/2026).

[120] James Park, Adam Zhao, Bing Yin, Charlie Taylor, Michael Frankovich, Parthasarathy Govindarajen, Nicolas Trown, Yang Zhou, and Faqin Zhong. *How Amazon Scaled Rufus by Building Multi-node Inference Using AWS Trainium Chips and vLLM*. AWS Machine Learning Blog. Aug. 2025. URL: https://aws.amazon.com/blogs/machine-learning/how-amazon-scaled-rufus-by-building-multi-node-inference-using-aws-trainium-chips-and-vllm/ (visited on 05/04/2026).

[121] Yingjiao Zhai, Satyam Kumar, Sundara Raman Ramachandran, Chuanrui Zhu, Chanh Nguyen, Farhan Toddywala, Chunnan Yao, Caleb Johnson, and Qing Lan. *How We Leveraged vLLM to Power Our GenAI Applications at LinkedIn*. LinkedIn Engineering Blog. Aug. 2025. URL: https://www.linkedin.com/blog/engineering/ai/how-we-leveraged-vllm-to-power-our-genai-applications (visited on 05/04/2026).

[122] Lianmin Zheng et al. "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2023. URL: https://proceedings.neurips.cc/paper_files/paper/2023/hash/91f18a1287b398d378ef22505bf41832-Abstract-Datasets_and_Benchmarks.html.

[123] vLLM Team. *vLLM: Easy, Fast, and Cheap LLM Serving with PagedAttention*. vLLM Blog. 2023. URL: https://vllm.ai/blog/vllm (visited on 05/04/2026).

[124] Pol G. Recasens, Ferran Agullo, Yue Zhu, Chen Wang, Eun Kyung Lee, Olivier Tardieu, Jordi Torres, and Josep Ll. Berral. "Mind the Memory Gap: Unveiling GPU Bottlenecks in Large-Batch LLM Inference". In: *Proceedings of the 2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*. 2025, pp. 277–287. DOI: 10.1109/CLOUD67622.2025.00036.

[125] NVIDIA. *CUDA C++ Programming Guide*. NVIDIA Developer Documentation, CUDA Toolkit 12.4.0 archive. 2024. URL: https://docs.nvidia.com/cuda/archive/12.4.0/cuda-c-programming-guide/index.html.

[126] Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. "Benchmarking and Dissecting the Nvidia Hopper GPU Architecture". In: *arXiv preprint arXiv:2402.13499* (2024). DOI: 10.48550/arXiv.2402.13499. URL: https://arxiv.org/abs/2402.13499.

[127] Triton contributors. *Triton: a language and compiler for writing highly efficient custom Deep-Learning primitives*. GitHub repository. 2024. URL: https://github.com/triton-lang/triton.

[128] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of ISCA*. 2017.

[129] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071. URL: https://doi.org/10.1109/TC.1972.5009071.

[130] Richard M. Russell. "The CRAY-1 computer system". In: *Communications of the ACM* 21.1 (1978), pp. 63–72. DOI: 10.1145/359327.359336. URL: https://doi.org/10.1145/359327.359336.

[131] NVIDIA. *MIG User Guide*. NVIDIA Multi-Instance GPU User Guide. 2024. URL: https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[132] Aditya Dhakal, Sameer G. Kulkarni, and K. K. Ramakrishnan. "GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. ACM, 2020, pp. 492–506. DOI: 10.1145/3419111.3421284.

[133] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. "Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, 2022, pp. 539–558. URL: https://www.usenix.org/conference/osdi22/presentation/han.

[134] Marcus Chow, Ali Jahanshahi, and Daniel Wong. "KRISP: Enabling Kernel-wise RIght-sizing for Spatial Partitioned GPU Inference Servers". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 624–637. DOI: 10.1109/HPCA56546.2023.10071121.

[135] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. "Paella: Low-latency Model Serving with Software-defined GPU Scheduling". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. ACM, 2023, pp. 595–610. DOI: 10.1145/3600006.3613163.

[136] Wei Zhao, Anand Jayarajan, and Gennady Pekhimenko. "Tally: Non-Intrusive Performance Isolation for Concurrent Deep Learning Workloads". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ACM, 2025, pp. 1052–1068. DOI: 10.1145/3669940.3707282.

[137] Patrick H. Coppock, Brian Zhang, Eliot H. Solomon, Vasilis Kypriotis, Leon Yang, Bikash Sharma, Dan Schatzberg, Todd C. Mowry, and Dimitrios Skarlatos. "LithOS: An Operating System for Efficient Machine Learning on GPUs". In: *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. ACM, 2025, pp. 1–17. DOI: 10.1145/3731569.3764818.

[138] Foteini Strati, Xianzhe Ma, and Ana Klimovic. "Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications". In: *Proceedings of the Nineteenth European Conference on Computer Systems*. ACM, 2024, pp. 1075–1092. DOI: 10.1145/3627703.3629578.

[139] Shulai Zhang, Ao Xu, Quan Chen, Han Zhao, Weihao Cui, Zhen Wang, Yan Li, Limin Xiao, and Minyi Guo. "Efficient Performance-Aware GPU Sharing with Compatibility and Isolation through Kernel Space Interception". In: *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. USENIX Association, 2025, pp. 1003–1019. URL: https://www.usenix.org/conference/atc25/presentation/zhang-shulai.

[140] Yongkang Zhang, Haoxuan Yu, Chenxia Han, Cheng Wang, Baotong Lu, Yunzhe Li, Zhifeng Jiang, Yang Li, Xiaowen Chu, and Huaicheng Li. "SGDRC: Software-Defined Dynamic Resource Control for Concurrent DNN Inference on NVIDIA GPUs". In: *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. ACM, 2025, pp. 267–281. DOI: 10.1145/3710848.3710863.

[141] Juan José Martín, José Flich, and Carles Hernández. "Peformance Isolation for Inference Processes in Edge GPU Systems". In: *arXiv preprint arXiv:2601.07600* (2026). DOI: 10.48550/arXiv.2601.07600. URL: https://arxiv.org/abs/2601.07600.

[142] Jonathan Bentz and Tony Scudiero. *NVIDIA CUDA 13.1 Powers Next-Gen GPU Programming with NVIDIA CUDA Tile and Performance Gains*. NVIDIA Technical Blog. Dec. 2025. URL: https://developer.nvidia.com/blog/nvidia-cuda-13-1-powers-next-gen-gpu-programming-with-nvidia-cuda-tile-and-performance-gains/.

[143] PyTorch Foundation. *PyTorch Foundation Welcomes vLLM as a Hosted Project*. PyTorch Blog. 2025. URL: https://pytorch.org/blog/pytorch-foundation-welcomes-vllm/.