Toward Intent-Aware, Adaptive Cloud Systems

A dissertation

submitted by

Abdullah Bin Faisal

In partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Science

TUFTS UNIVERSITY

August 2025

ADVISOR: Prof. Fahad R. Dogar

Toward Intent-Aware, Adaptive Cloud Systems

Abdullah Bin Faisal

ADVISOR: Prof. Fahad R. Dogar

Modern cloud systems must support an increasingly diverse set of workloads, from latency-sensitive web services and large-scale machine learning training to emerging prompt-driven applications powered by large language models (LLMs). These workloads vary widely in structure, goals, and resource demands, yet existing infrastructures force users to translate high-level intents (e.g., predictable latency, cost efficiency, fairness) into low-level mechanisms such as scheduling policies, GPU allocations, or caching strategies. This gap results in brittle fixed policies, opaque trade-offs, and interfaces that remain mechanism-oriented rather than goal-oriented.

This thesis proposes a design philosophy of intent-aware, adaptive cloud systems: infrastructure that (i) exposes intuitive, high-level interfaces for expressing goals, (ii) employs intelligence (e.g., learning) to adapt their behavior under diverse workloads and user preferences, and (iii) supports trade-off navigation through bi-directional interfaces. By combining expressiveness and adaptivity, such systems can align resource management decisions more directly with user expectations while remaining robust to workload variability.

The dissertation demonstrates this philosophy across three systems. 2D is a learning-based network scheduler that offers robust tail performance in face of changing cloud workloads (i.e., flow size distributions) by composing simple scheduling primitives (serialization and multiplexing) in a principled way. PCS is a GPU job scheduler that elevates predictability as a first-class objective, providing reliable job completion time estimates and a bi-directional interface that lets cloud operators explore trade-offs between predictability, and traditional scheduling objectives (e.g.,

fairness). LLMProxy is a proxy for prompt-driven applications that aims to reduce LLM-inference cost by accepting user-preferences via a `service_type` abstraction and mapping them to low-level decisions such as model selection, context shaping, and caching.

Together, these systems illustrate how intent-aware, adaptive infrastructures can deliver robust, flexible, and user-aligned performance across domains ranging from data center networking to GPU cluster scheduling and LLM inference.

To Mama, for teaching me how to use the computer mouse,

To Baba, for showing me what hard work looks like.

Acknowledgments

Alhamdulillah, all thanks are due to Allah for granting me the strength and perseverance to complete this journey.

I am deeply grateful to my advisor, Dr. Fahad Dogar, for believing in me whenever I doubted myself, and for giving me the time and space to pursue the problems I cared about. His encouragement to "go deeper" and advice to "write as if your mother were reading it" have profoundly shaped how I think and work. Thank you for creating opportunities for me, and for your reminders to push hard and embrace the intensity this journey demands. I am equally thankful to his family for treating me and my own family as part of theirs.

I thank the members of my dissertation committee for their thoughtful feedback, which helped improve this thesis. In particular, I am grateful to Dr. Raja Sambasivan for always being a kind and reassuring voice whenever I have been too harsh on myself.

I have had the privilege of having wonderful lab mates and colleagues, past and present. All of them have brought me joy and fulfillment in their own ways: Noah, for suggesting hikes I still aspire to measure up to; Rukhshan, for informally co-founding a dozen hilarious startup ideas with me and being a source of creative energy; Hiba, for the camaraderie and the eternal "hi vs. salam" debate; Musa, for our early conversations about impactful research; Sana, for pulling me into long bike rides and broadening my taste beyond desi food; and Tooba, for sharing the chaos of starting a PhD.

Special thanks to Mohsin, Osama, and Shahrukh, who have been like elder brothers: Mohsin for helping me settle in, teaching me to cook more than fried eggs, and for countless whiteboard discussions that went nowhere and everywhere

at once. My only complaint is that you graduated on time (too early). Osama, for exemplifying discipline and for being my go-to for every admin and system setup — I owe my CloudLab skills to you. Shahrukh bhai, for his advice on the PhD journey, especially on the post "PhD honeymoon" phase.

The Tufts community and the CS department have been a constant source of support, especially Jenny Mooney, Sarah Richmond, and Sandra Schulenburg, who always helped with logistics and last-minute requests. I am also thankful to have been selected for and be the recipient of the Dean's Fellowship awards and the Loevner Fellowship.

I owe much to my undergraduate advisors and mentors: Dr. Zartash Uzmi, for being a father figure throughout these years and inspiring me through his dedication; Dr. Ihsan Qazi, for his optimism and positivity that lifted me at low points and for first sparking my interest in cloud computing; Dr. Tariq Jadoon, for getting me excited about queueing and scheduling systems; Dr. Momin Uppal, for showing me what excellent teaching looks like; and Ali Hannan, my high school debates coach, for teaching me not just to orate, but to orate well.

I am extremely grateful to my friends outside of work. My childhood friends — Harris, Unzah, and Moaz — for being constant anchors and systems of support. Having you in my life is something I am deeply proud of. To Ammar, who quickly became like family. To Gufran, for being a rock throughout these years — our spontaneous hangouts, bike rides, and camping trips have been the cornerstones of my PhD journey. I am glad you reached out to me on Facebook before deciding on whether to come to Tufts.

Osama and Maryam have made imagining life outside of Boston really hard. Thank you for being my family away from home: from food to childcare to everyday help, you made life in Boston not only manageable but joyful. Maryam baji's cooking sustained me in my early years more times than I can count.

To my family, whose prayers, encouragement, and love have carried me: my grandparents, for their wisdom and tenderness; Mama and Baba, for their endless duas, support, and reminders of responsibility; my sisters, for keeping me guessing whether I'll get tough love or gentle love — but love all the same; and my parents-in-law, for always reminding me to prioritize my work. To my son Raahim, who

made sure I never wasted time "relaxing" — his daily reminder, "Baba, you go lab", was the soundtrack I loved listening to during my final stretch. And to my daughter Leenah, for her perfectly timed arrival just months before I finished — very clever of her to show up right at the end.

Finally, my deepest gratitude to my wife, Fizzah. Her love and support made this journey soul-fulfilling. I am grateful for the countless meals she cooked, the graphs she made for my papers, and the many ways she lifted me up. I am incredibly fortunate to have you by my side and so glad that we went through this together. Thank you Jo.

Abdullah Bin Faisal

TUFTS UNIVERSITY

August 2025

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Modern cloud systems must support an increasingly diverse set of workloads, ranging from latency-sensitive web services and large-scale machine learning training jobs to emerging prompt-driven applications powered by large language models (LLMs). These workloads differ widely in their structure, performance goals, and resource demands: some require predictable latency, others need cost efficiency etc. As a result, *resource management* — the process of allocating scarce and typically shared resources (e.g., GPUs, network bandwidth) — has become a central and increasingly complex systems challenge.

Users — including application developers, cloud operators, tenants, and end-users — care about high-level outcomes, which are often competing. An end-user wants their search results or chat responses to appear instantly. A cloud tenant running an ML training pipeline wants their job to complete within a predictable time window. A cloud operator managing shared infrastructure aims to ensure fairness across competing workloads while meeting service-level objectives. While these goals vary, they all require making efficient use of underlying resources — compute, memory, network bandwidth, and more — in ways that reflect users' intent.

Achieving these outcomes, however, requires configuring low-level system mechanisms such as scheduling algorithms, resource selection policies, and caching

strategies. For example, to reduce LLM inference cost, a developer might truncate prompts or choose a cheaper model. A cloud operator might apply fair sharing to allocate resources, which works well for some workloads but poorly for others [203]. An ML practitioner (i.e., cloud tenant) might over-provision GPUs for their model training job without knowing if it improves runtime [170, 211]. In each case, users must translate high-level goals into low-level configurations; a process that is opaque, requires deep expertise, and often application-specific.

This gap between *what* users want and *how* systems operate manifests in several ways:

- **Brittleness of Fixed Policies.** Many systems use fixed policies designed to optimize for a specific application workload or performance metric. For instance, when it comes to resource allocation, a FIFO scheduler [193] reduces tail latency under light-tailed workloads, where most jobs are of comparable size and extreme outliers are rare, but performs poorly under heavy-tailed workloads, where a small fraction of very large jobs dominate overall demand. Similarly, pre-emptive resource scheduling policies promote goals such as fairness and lower average job completion times, but often come at the cost of reduced predictability in individual jobs' execution times [90]. Recent research formalizes this limitation: no single non-learning resource allocation strategy is robust across all workloads and objectives [203]. In practice, this means operators must frequently retune configurations or fall back on ad hoc fixes. As workload diversity and system complexity grow, this brittle, manual tuning process becomes a bottleneck for scalability and reliability.

- **Competing Objectives and Trade-offs.** Resource management often involves navigating inherent trade-offs between competing goals: ensuring high resource utilization might hurt average latency [86]; improving predictability in a user's job completion time may require conservative scheduling that sacrifices throughput [117]; using a higher-quality LLM can improve accuracy but at a greater cost [47]. These trade-offs shift with workload characteristics, user preferences,

and deployment context. Yet most systems optimize for a single fixed goal (e.g., minimizing mean latency), leaving users to grapple with trade-offs they neither control nor understand.

- **Mechanism-Oriented Interfaces.** Even when systems expose configuration options, they often take the form of mechanism-oriented interfaces — knobs like setting queue priorities [33], or number of GPUs to use for a job [89] — rather than intent-oriented abstractions. These interfaces assume that users can translate their high-level goals (e.g., "maximizing quality of an LLM response while minimizing costs") into appropriate parameter settings. But in practice, this translation can be highly workload-specific and typically requires deep systems expertise. As a result, systems remain difficult to control and adapt poorly to new applications or changing conditions.

Together, these issues point to a deeper problem: today's systems expect users to manage complexity manually, rather than providing tools that adapt intelligently on the user's behalf. This dissertation explores how to address this problem by designing systems that are *smarter*: they understand and learn workload characteristics, and *goal-aligned*: they allow users to express what they care about, and make low-level decisions accordingly.

The key insight is that modern systems must be both adaptive and expressive: they must adapt to workload dynamics, system state, and user preferences, while also exposing interfaces that allow users to specify high-level goals rather than low-level mechanisms. This requires rethinking how we design such systems; not as static policy engines, but as learning and decision-making frameworks that operate across layers, from infrastructure to application.

1.2 Intent-Aware, Adaptive Systems

These insights highlight a recurring challenge in modern cloud systems: users care about high-level goals, but systems expose low-level mechanisms. Addressing this

gap requires a shift in how we design systems. This dissertation is grounded in a set of cross-cutting principles that guide the design of such systems:

Expose Intent, Not Mechanisms. Rather than relying on users to configure system internals, systems should expose interfaces that let users express what they want, such as minimizing cost, or navigating fairness-predictable latency trade-offs. These high-level goals serve as the basis for aligning system behavior with user expectations.

Use Learning to Bridge the Gap. Mapping intent to system behavior requires learning. Fixed policies fail to generalize across diverse workloads and goals. Systems must observe workload patterns, user preferences, and performance feedback to *learn* how to utilize low-level systems techniques that reflect the specified objectives.

Identifying the Right Control Primitives. To support adaptability, systems must be built around control primitives that are expressive enough to capture different optimization goals, yet structured enough to be tuned or learned. These include scheduling strategies that expose various trade-offs, and runtime controls that adjust caching or model selection behavior for LLM-inference workloads.

Support Trade-Off Navigation through Bi-Directional Interfaces. Real-world goals are rarely single-dimensional. Systems must surface and manage trade-offs; for example, those between cost and accuracy, or predictability and fairness. This requires bi-directional interfaces that not only accept user goals but also convey the consequences of different choices, helping users make informed trade-offs.

We now show how these principles are applied across three systems developed in this dissertation:

- **2D** is a learning-based network scheduler that adapts tail latency performance to changing flow size distributions by composing simple scheduling primitives —

Principle	2D (Bandwidth Scheduling)	PCS (GPU Job Scheduling)	LLMProxy (Prompt Serving)
Capturing Intent	Optimize for tail latency implicitly	Cloud operators specify preferences (e.g., fairness vs. predictability)	Users choose from high-level service types
Using Learning/Intelligence	Learning flow-size distributions	Simulation-based search strategy to learn different trade-offs	Low-cost LLMs to make decisions (e.g., which model to use)
Control Primitives	Serialization and Multiplexing	Weighted Fair Queues (WFQ)	Cache, Context Mngr, Model Selector
Bi-directional Trade-Off Resolution	N/A; Fully automatic	Pareto frontier of trade-offs to choose from	Returned Metadata (e.g., amount of context used) to explain behavior

Table 1.1: Application of design principles across systems developed in this dissertation.

serialization and multiplexing — based on learned workload characteristics. *This work appeared at ACM CoNext'18*

- **PCS** is the first GPU job scheduler that provides job completion time predictions to users. It exposes a bi-directional (Pareto frontier) interface to cloud operators, helping them navigate trade-offs between reliable predictions and traditional scheduling objectives (e.g., fairness). It achieves its goals by bridging preemptive and non-preemptive scheduling techniques by learning how to configure Weighted Fair Queues (WFQs) based on workload dynamics and cloud operators' preferences. *This work appeared at USENIX OSDI'24*

- **LLMProxy** is a prompt-serving infrastructure that reduces LLM inference cost through intelligent model selection, context shaping, and caching. It captures users' cost-saving intent via a lightweight `service_type` abstraction that steers low-level decisions such as model choice and context length. This enables users to realize different cost-quality trade-offs.

Table 1.1 summarizes how these principles are applied to these three systems. We now provide additional details.

1.2.1 2D: A Learning-Based Network Scheduler for Robust Tail Performance

Modern cloud applications are highly sensitive to network performance, especially tail latency. Prior proposals (e.g., pFabric [29], PDQ [104], Orchestra [53]) have argued to go beyond TCP and shown improved response times by reordering network

flows based on static scheduling policies like Shortest-Job-First (SJF) or FIFO. While effective in narrow regimes, these policies lack robustness: each performs well only for specific workload characteristics and optimization metrics. For example, FIFO is optimal for light-tailed workloads but suffers from head-of-line blocking under heavy-tailed conditions.

2D addresses this challenge by introducing a *learning-based*, workload-adaptive scheduler that delivers robust performance across diverse workloads and objectives. The key idea is to separate scheduling decisions across timescales: long-term adaptation to workload distribution and short-term enforcement of efficient flow ordering. To do so, 2D combines two classic scheduling primitives — *serialization* (e.g., FIFO within flow classes) and *multiplexing* (Fair-Sharing across classes) — in a composable, data-driven way.

At a high level, 2D learns the workload's flow size distribution and partitions flows into size-based classes. It then allocates bandwidth across classes (multiplexing) in a way that avoids HOL blocking, while serializing flows within each class to optimize completion time. These decisions are made at different granularities: class thresholds and bandwidth allocations are updated at long timescales (seconds or minutes), while flow ordering within a class is enforced per-arrival.

2D's mechanism is designed for practical deployment: it requires no switch changes and supports distributed enforcement. Bandwidth allocation and flow classification are done at end-hosts, while a lightweight centralized sequencer enforces per-class FIFO ordering.

We have implemented 2D in C as an application-layer transport protocol, with a centralized module that continuously learns workload characteristics and updates scheduling thresholds. In testbed experiments on a 10Gbps Emulab cluster using realistic cloud application workloads (e.g., data mining, web search), 2D outperforms both static scheduling baselines (e.g., FIFO, PS) and prior smarter heuristics (e.g., Aalo [52], Baraat [66], PIAS [33]). It delivers robust performance across diverse workloads, achieving up to 10× lower tail (p90) completion times.

1.2.2 PCS: A Predictability-Centric GPU Scheduler

As AI workloads continue to scale, GPU clusters have become central to machine learning development. Yet, in shared, multi-tenant clusters, job completion times are notoriously hard to execute in a predictable manner due to complex scheduling behavior; particularly the use of aggressive preemption based policies (e.g., fair-sharing, shortest-job-first). While preemption helps optimize for performance and fairness, it undermines predictability by introducing variability that depends on unknown future job arrivals. Conversely, non-preemptive policies like FIFO are predictable but often inefficient and unfair.

PCS addresses this trade-off by designing a scheduling framework that places *predictability as a first-class objective*, alongside fairness and performance. The core insight behind PCS is that bounded preemption, as provided by *Weighted Fair Queuing* (WFQ), can bridge the gap between rigid non-preemptive policies and highly dynamic preemptive schedulers. By tuning WFQs (e.g., number of queues, GPU shares across queues etc.), PCS can span the space of different scheduling configurations.

Rather than returning a single opaque scheduling configuration, PCS introduces a *bi-directional* interface that allows cloud operators to specify their high-level preferences (e.g., average job completion time vs. average prediction error) and explore different configurations, and hence trade-offs. This enables informed, goal-aligned decision-making based on the operator's relative preferences.

Internally, PCS relies on two simulation-aided components: (1) At longer timescales, it employs a *simulation-based search strategy* to efficiently explore the trade-off space induced by different WFQ configurations. This process identifies Pareto-optimal configurations by efficiently varying the number of queues, job-to-queue mappings, and queue weights. (2) At runtime, PCS uses a lightweight, simulation-based predictor to project individual job completion times under the chosen WFQ configuration and current cluster state (e.g., existing jobs and their demands). This provides the per-job predictability needed for cluster users.

PCS is implemented in Python as a modular scheduling framework built on

Ray [144], and evaluated through both small-scale testbed experiments and large-scale simulations. The evaluation uses realistic ML training workloads derived from Microsoft's shared GPU cluster traces [114]. PCS reduces job completion time prediction error by 50–800% compared to performance-optimized schedulers, while remaining within 1.1–3.5× of their performance and fairness metrics.

1.2.3 LLMProxy: A Proxy-Based Framework for Cost-Aware Prompt Serving

As applications increasingly rely on prompt-based interactions with LLMs, the cost and complexity of serving these prompts have become key challenges. Unlike traditional web requests, LLM queries involve decisions around model choice, context construction, and caching, each of which directly affects both cost and quality. However, current infrastructure leaves these decisions to the application, requiring developers to manage low-level optimizations.

LLMProxy introduces a general-purpose, application-facing proxy for intent-aware and cost-efficient prompt serving. The core idea is to identify common LLM-serving decisions: *model selection*, *context shaping*, and *caching*, and compose them behind a unified, delegation-based interface. Instead of manually configuring optimization knobs, applications simply specify a high-level `service_type` (e.g., `cheap`, `balanced`, `max_quality`) to express their preferences. LLMProxy then dynamically selects the most cost-effective execution plan based on these preferences.

LLMProxy also incorporates a *bi-directional interface* that supports transparency and iterative control. It returns metadata detailing how each request was handled, including the selected model, cache hit/miss status, and context length, enabling users to understand the trade-offs made. If needed, applications can respond by adjusting their `service_type` and regenerating the prompt.

Internally, LLMProxy uses *low-cost LLMs* to drive decision-making across three pluggable components. The model adapter invokes an internal model to choose which LLM to use, including combining multiple LLMs. The context manager uses a local model to determine which parts of the input are most relevant, enabling

dynamic context reduction. The semantic cache employs a lightweight model to decide both when to serve a cached response and how to construct it from stored content.

LLMProxy has been deployed in two real-world settings. First, it powers a WhatsApp-based Q&A service targeting users in developing regions, where cost sensitivity is critical. Over 12+ months, the system has served 14.7K+ queries from more than 100 users. Second, LLMProxy was integrated into three university courses, supporting students as they built LLM-powered applications under strict budget constraints. Post-course surveys indicated that the proxy was intuitive to use and easy to integrate into student workflows. In addition to these deployments, microbenchmarks show that LLMProxy's optimizations (i.e., smart model selection, context shaping, and caching) can reduce LLM-serving costs by up to 30%.

1.3 Thesis Statement

This dissertation demonstrates that systems which admit high-level user goals and learn to adapt their behavior offer a more flexible and robust alternative to traditional, fixed-policy infrastructure.

In particular, this thesis states that:

We can build efficient and adaptive systems by: (1) exposing intuitive, high-level interfaces for users to express their objectives/intents/trade-offs, and (2) using intelligence (e.g., learning-based techniques) to adapt system behavior based on workload characteristics, user goals, or operational constraints.

By aligning system behavior with user intent through intelligence, we can design systems that are both responsive to evolving needs and robust under diverse workloads. This approach complements domain heuristics with learning and adaptation, enabling systems to flexibly align with user goals and workload dynamics. Rather than relying solely on fixed strategies or rigid abstractions, it uses learning-based techniques to make more informed decisions, offering flexibility and responsiveness without sacrificing control. The systems presented in this disser-

tation demonstrate how this philosophy can be applied across diverse domains — including network scheduling (2D), GPU job orchestration (PCS), and LLM-based prompt serving (LLMProxy) — illustrating both the generality and effectiveness of this approach.

Scope. Our focus is on cloud and AI infrastructure systems that must adapt to diverse workloads and evolving user goals. The proposed design philosophy, centered on intent-aware interfaces and learning-based adaptation, is applied to systems concerned with resource management, performance predictability, and cost efficiency. The work spans domains including network scheduling, GPU job orchestration, and LLM inference, where users can express preferences that systems can interpret and learn from.

This work does not aim to build general-purpose AI agents (e.g., AlphaEvolve [156]) or natural language-based interfaces, though the abstractions presented could support such extensions in future work. While the techniques focus on intelligent adaptation, they are grounded in domain-specific control primitives that balance flexibility with ease of deployment.

Assumptions. This dissertation makes several assumptions to bound the problem space:

Workload observability. Resource demands and performance signals (e.g., network flow sizes, job runtimes, or query characteristics) are assumed to be observable or reasonably predictable using existing techniques. For example, prior work provides methods for classifying short versus long flows [66, 29], and for forecasting training job completion times [136, 170].

Non-adversarial setting. Users are assumed to behave cooperatively, expressing preferences or intents honestly; the systems are not designed to withstand adversarial manipulation of inputs.

Non-Goals. These assumptions narrow the space of problems we address. Complementing them, this dissertation explicitly does not attempt to:

Develop new learning algorithms. The work leverages existing learning or forecasting techniques (e.g., regression) for demand and workload estimation but does not propose novel learning models.

Redesign underlying mechanisms. The TCP/IP stack, GPU kernel execution, and LLM serving frameworks are treated as given; the contributions lie in how they are orchestrated and adapted.

Optimize for all objectives simultaneously. Each system narrows in on particular trade-offs (e.g., predictability in scheduling, cost in inference) rather than attempting to optimize for every possible metric. Although the frameworks and ideas introduced can be generalized to objectives and metrics not considered by the proposed systems (e.g., energy efficiency).

Provide general-purpose agency. While abstractions here may support broader agentic workflows, the focus is not on designing autonomous AI agents or natural language interfaces.

1.4 Contributions

Overall, this thesis consists of the following contributions:

Proposed Ideas/Systems

- We advocate for a design philosophy centered around intent-aware, adaptive systems and articulate core design principles, including exposing high-level intent, identifying adaptable control primitives, applying learning-based techniques in a principled way, and supporting bi-directional exploration of trade-offs.
- In Chapter 2 we propose a learning-based bandwidth scheduler for cloud systems, aimed at offering robust tail-latency across different application workloads. We show how a simple learning-based scheme combined with principled use of basic scheduling primitives can outperform existing strategies
- In Chapter 3 we propose a novel resource management objective: offering reliable

job completion time predictions and design a system that balances prediction accuracy with traditional scheduling objectives (e.g., fairness)

- In Chapter 4, we introduce a prompt-serving proxy that supports cost-saving optimizations, including smart model selection, context pruning, and caching, and present the design of a lightweight, serverless architecture that exposes a high-level, bidirectional API for delegating control over prompt handling while preserving transparency.

Software Artifacts The systems developed in this dissertation are implemented as functional prototypes. All artifacts are either publicly available or deployed in production-like environments:

- 2D is implemented as a modular, flow-based network scheduler requiring no changes to switches. It is open-sourced and available at: `github.com/abdullahfsm/multiclass-scheduler`
- PCS is implemented within the Ray distributed framework [144] and supports exploration of WFQ-based scheduling policies. The artifact has been awarded all three badges (Functional, Reusable, Available) by the OSDI'24 artifact evaluation committee and is available at: `github.com/abdullahfsm/PCS/tree/osdi2024-artifact`. The prototype enables reproduction of key results and interactive exploration of Pareto-optimal trade-offs for custom workloads
- LLMProxy is deployed in production using a serverless AWS infrastructure, integrating Lambda functions with persistent storage layers. It powers a WhatsApp-based Q&A service with rich features (e.g., leaderboards, follow-up prompts). The software has also been made available for use in university classroom settings via a RESTful API

Evaluation/Deployment Insights Each system in this dissertation is evaluated using either real deployments, controlled testbeds, or simulation-driven studies. To-

gether, these evaluations highlight how intent-aware, adaptive systems can generalize across workloads, reduce cost and unpredictability, and better align with user goals:

- We evaluate the 2D prototype on three cloud application workloads on a 10G testbed on Cloudlab [3]. Our results show that 2D demonstrates robust tail latency performance across these workloads, outperforming other strategies by up to 4×. We also evaluate 2D in scenarios where the workloads i) are co-located, and ii) change over time, shedding light on 2D's agility to adapt to such scenarios.

- We evaluate the PCS prototype for a 16 GPU testbed on Cloudlab, on an AutoML style workload consisting of ML hyperparameter tuning tasks. We also run larger scale experiments on a simulator, replaying realistic ML training traces (Philly [114]). Collectively, our results show that PCS can increase prediction reliability by 50–800% while operating within 1.1–3.5× of performance- or fairness-optimal baselines.

- We share our deployment experience supporting a Q&A WhatsApp service and three courses across two semesters. As part of our study, we conducted surveys which indicated that over 75% of users found the proxy easy to use, and its features useful. We also evaluate our system on traces derived from the WhatsApp service logs, and we show that the proxy achieves 30% cost savings compared to default strategies used by most production LLM serving systems today.

1.5 Related Work

A growing body of research has explored how to make systems more adaptive and responsive to user or workload requirements. Broadly, these efforts fall along three complementary axes: (i) high-level interfaces that expose control over low-level system objectives, (ii) intelligent low-level control that uses learning-based techniques or optimization to fine-tune policies, and (iii) workload adaptivity that learns from workload dynamics to remain robust across diverse scenarios. This thesis draws inspiration from all three directions but focuses on *combining* their strengths: exposing

intent through expressive interfaces while adapting low-level mechanisms beneath them.

Interfaces for Expressing Objectives. A first line of work develops higher-level abstractions or domain-specific languages (DSLs) for resource management. Cluster and GPU schedulers such as Carbyne [86], Gandiva [210], and Gavel [149] provide ways for cloud operators to express fairness, efficiency, or priority goals. NUMFabric [147] abstracts away bandwidth allocation decisions using a network utility maximization framework. Programmable networks have similarly embraced domain-specific interfaces, most notably with P4 [39] and follow-up work, which enables operators to specify packet-processing pipelines declaratively. More recent work like Cilantro [38] allows cloud operators to specify different resource allocation objectives as utility functions. These systems highlight the value of surfacing objectives explicitly, but the burden is still on the human to specify a fixed goal or to manually compose multiple objectives into a single utility function; the system itself rarely adapts its interpretation of objectives as conditions evolve, and in many cases, the interfaces themselves require substantial expertise (e.g., encoding preferences as utility functions [147, 38] or optimization equations [149]).

Intelligent Low-Level Control. A second research thrust has focused on embedding intelligence into low-level control policies. In congestion control, examples include Remy [207], and PCC [70], which use offline or online training to discover transport algorithms. In cluster scheduling, Decima [139] applied deep reinforcement learning to DAG scheduling, while SelfTune [121] continuously adapts cluster policies via feedback-driven learning. Beyond scheduling, researchers have applied learning to optimize caching policies [116], storage systems [94], and system configuration for CDNs [150]. Collectively, these systems showcase the benefits of using learning to go beyond human-designed heuristics, demonstrating that adaptive, data-driven control can achieve efficiencies difficult to capture with static policies.

Workload Adaptivity. A third body of work emphasizes adaptivity across variable and uncertain workloads. In networking, AuTo [46] applied reinforcement learning to optimize flow completion times for workloads, while coflow schedulers like Baraat [66] use limited form of workload learning to distinguish between short and long flows. In GPU cluster scheduling, systems like Themis [136], Pollux [170], and Optimus [165] learn job characteristics (e.g., an AutoML job's structure) to reduce average job completion time or improve fairness. These systems highlight the importance of learning from workload structure rather than relying on static assumptions. Yet, most of these systems target a fixed system-chosen metric (e.g., minimizing tail latency) rather than being able to handle diverse goals and user intents.

Distinguishing Themes. Together, these three strands — interfaces, intelligent low-level control, and workload adaptivity — represent the state of the art in resource management for networks, clusters, and emerging AI platforms. The distinguishing thread in this thesis is not only to build on these advances but also to bridge them: providing interfaces that capture user or operator intent in a simple way, while simultaneously employing adaptive mechanisms that learn and respond under the hood. This combination allows the system to deliver outcomes that are *both* aligned with what stakeholders want and robust to workload and variability.

1.6 Dissertation Organization

The remainder of this dissertation is organized as follows.

- **Chapter 2** presents 2D, a learning-based scheduler for data center networks that adapts to workload dynamics and achieves tail-robust performance by composing simple scheduling primitives and separating policy from enforcement.
- **Chapter 3** describes PCS, a GPU job scheduler for shared ML clusters. PCS introduces a high-level interface for exploring trade-offs among predictability, fairness, and latency, and uses a simulation-driven learning strategy to map user-specified objectives to low-level scheduling behavior.

- **Chapter 4** introduces LLMProxy, a proxy for prompt-driven applications that uses an LLM-powered agent to interpret user intent and configure downstream behaviors such as model selection and caching. This system demonstrates how intent-awareness can enable flexible, evolvable LLM infrastructure.
- **Chapter 5** concludes with a summary of the dissertation's contributions and the broader implications of the proposed design philosophy for future systems.

Chapter 2

2D: Learning-based Network Flow Scheduling

2.1 Overview

Existing bandwidth scheduling systems for data center networks optimize for a specific workload and performance metric. In this chapter, we present 2D, a new scheduler that offers *robustness* across performance metrics and changing workloads – a ground existing scheduling policies are unable to cover. 2D combines basic scheduling building blocks of multiplexing and serialization in a principled way, ensuring tail optimal performance across workloads while also improving the average (and lower percentiles) completion times.

To implement 2D for flow-level scheduling in a distributed setting, we break-up the scheduling decision into two parts: coarse time-scale decisions based on workload and load changes are made by a centralized controller while per-flow serialization decisions are made in a distributed fashion, involving the end-points and sequencer(s). Our testbed experiments show that, for realistic cloud workloads, 2D provides consistent gains at the tail and average flow completion times compared to basic scheduling techniques (e.g., FIFO and processor sharing) as well as heuristic-based schedulers (e.g., Aalo and Baraat).

2.2 Motivation

Our work is motivated by the observation that typical cloud workloads have high variability (§2.2.1), which causes existing scheduling techniques to significantly underperform (§2.2.2), motivating the need for a learning-based adaptive scheduling policy (§2.2.3).

2.2.1 Workload Variability

We loosely define a workload to correspond to the network footprint of a cloud application (e.g., Web-Search, Cache) or a job from a tenant in a compute cluster (e.g., Sorting, Word Count, Model Training, etc). Prior work shows that common cloud workloads can have high variability – both *within* a workload as well as *across* workloads [66, 120, 59].

Within-Workload Variability. Most popular network workloads (e.g., web search [28], distributed data mining [87], etc) follow a heavy-tailed flow size distribution: they have a mix of short and long flows. While taking a static view of these workloads is useful, it hides the *temporal* variations that can occur within such a workload. For example, at short time scales, not even a single large flow may be present in the system, and the workload may appear light-tailed during that duration.

To illustrate this phenomenon, we look at the fraction of small flows that have to contend with a large flow (we call this "contention"), for a heavy-tailed flow size distribution. We analyze the data-mining workload [87] – well-known for its heavy tailed flow size distribution – and apply a simple TCP-like scheme to allocate bandwidth to these flows. We define short flows as those smaller than 100KB, while flows larger than 1MB are classified as long flows. We calculate the contention as we vary the network.

Table 2.1 shows that, under medium network load (50%), almost 45% of the small flows never encounter a large flow during their life-span. Even for very high loads (e.g., 90%), a non-negligible fraction of flows never encounter a large flow. This points out that, at short time scales, even heavy-tailed distributions

Load	20%	50%	80%	90%
Contention	21.2%	54.7%	82.7%	93.0%

Table 2.1: Contention: fraction of short flows that contend with a large flow(s) as a function of the system load for data mining workload [87]. Despite the heavy-tailed nature of the workload, a significant number of short flows do not encounter a single large flow, especially at lower loads, indicating within-workload variability (§2.2.1).

may exhibit behavior similar to a light-tailed distribution, highlighting what we call within-workload variability.

Across-Workload Variation. Typical clusters run multiple applications or jobs; this applies to both compute clusters, where jobs from tenants may arrive in an online fashion [120, 59], as well as user-facing clusters, such as Facebook's "heterogeneous" clusters that run multiple applications [176]. These jobs may run concurrently or sequentially, depending on their arrival and departures times, as well as the cluster scheduling policy.

From the perspective of network flow scheduling, it is the online nature of job arrivals and departures, combined with their multiplexing, which makes the aggregate network workload highly variable. For example, a light-tailed workload running alone initially may later be multiplexed with a heavy-tailed workload, causing the aggregate workload to shift from being light-tailed to heavy-tailed. Similarly, two light-tailed workloads – with very different mean sizes – may become a heavy-tailed workload at the aggregate level, if they are multiplexed together.

Both within-workload variation and across-workload variation cause problems for existing scheduling techniques, as we discuss next.

2.2.2 Limitations of Existing Scheduling Policies

For our analysis, we consider three well-known scheduling policies: PS, SRPT, and FIFO. Our choice is informed by three factors. First, these policies have been extensively studied in the scheduling theory, with strong analytical results that we use for our discussion [30, 178, 138, 40, 188, 201]. Second, these policies are used by several recent scheduling proposals (e.g., PDQ [104], pFabric [29], Orchestra [53]). Third,

Performance Objective	Light-tailed Workloads	Heavy-tailed Workloads
AFCT	SRPT	SRPT
Tail FCT	FIFO	PS∼=SRPT

Table 2.2: Optimal or near optimal scheduling policies for different workloads (heavy vs. light-tailed) and objective functions (AFCT vs. Tail FCT). No single policy is optimal across workloads and performance metrics (§2.2.2), highlighting the need for an adaptive scheduling policy (§2.2.3).

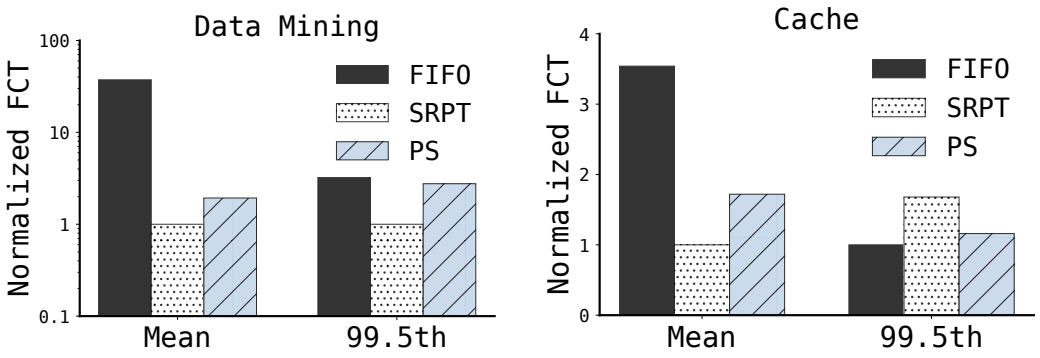
several recent heuristic-based scheduling techniques (e.g., Baraat [66], Aalo [52]), which we later cover in our evaluation, also use these policies as a building block.

The above scheduling policies suffer from two key limitations. First, they typically optimize for a *fixed workload distribution*, and their performance could suffer significantly if used for a different workload [52]. In practical settings, workloads could have variability, as we described in the previous section. Second, they optimize for a *fixed performance metric*, such as the average or tail completion times. For demanding cloud applications, it is important to reduce both the average as well as tail completion times.

These limitations often result in fundamental trade-offs that the system has to make. As highlighted in Table 2.2, no single scheduling policy is optimal across different workloads and performance metrics. In fact, an optimal strategy for a certain workload or metric performs poorly (or is the least desirable) for a different workload or metric. For example, SRPT optimizes AFCT for both heavy and light-tailed workloads but performs poorly at the tail for light tailed workloads. Similarly, FIFO is tail optimal for light-tailed workloads but performs poorly for heavy-tailed workloads.

While the above limitations are known, we *quantify* these trade-offs for realistic workloads. We simulate a single bottleneck link scenario and consider two popular workloads used in prior studies: workload from Facebook's Cache application which is light-tailed [176], and the data mining workload (VL2 [87]) which is heavy-tailed. Our key observations are as follows:

- **No Single Optimal Policy Within a Workload.** For the Cache workload,



(a) Heavy-tailed Data Mining workload

(b) Light-tailed Cache workload

Figure 2.1: Performance of scheduling policies for the heavy-tailed data mining and light-tailed Cache ([176]) workloads. SRPT achieves the lowest average FCTs in all cases but performs poorly at the tail for the Cache workload. FCTs are normalized with respect to the best scheme for each workload and metric.

FIFO performs the best at the tail but is up to 3.5× worse than the best performing scheme at the mean (Fig. 2.1a).

- **No Single Tail Optimal Policy Across Workloads.** FIFO – the tail optimal policy for Cache – performs the worst at the tail for the heavy-tailed workload (data mining), with up to 3× increase in completion time compared to the best performing policy (Fig. 2.1b).

2.2.3 Need for Adaptive Scheduling Policies

The above analysis shows the limitations of existing scheduling policies, leading to the question: *is it possible to have a scheduling policy that is robust across workloads and metrics?* Weirman and Zwart [203] show that there is no non-learning scheduling policy that provides tail optimal performance across workloads. To put simply, if workload information is unknown then no optimal policy exists. This highlights the importance of *learning* information about the workload.

Weirman et al. [148] show that if we have "limited" form of learning, as in limited processor sharing (LPS), a scheduling policy that adapts the level of multiplexing based on load can provide "tail robust" performance. The authors term this as a "limited" form of learning because the only information available to the

scheduler is the system load. While LPS seems like a promising approach, it has not yet been implemented and evaluated in the context of network flow scheduling. More importantly, if the scheduler can learn even more about the traffic workload (not just the load), there is potential room to do even better than LPS.

We argue that in today's networks it is feasible to learn more than just the load: using flow statistics collected by network elements, we can easily learn the workload *distribution*. Several recent SDN-based systems record flow statistics at end-points (or switches), collect it at a centralized controller, and use this information for various purposes, such as traffic engineering [27, 173]. In fact, collecting similar workload information is feasible for most cloud systems that make scheduling decisions, such as a cluster manager (e.g., Mesos [102]) or a storage controller [191]. Further, systems that do not explicitly track workload can use service times – which is typically recorded by systems for performance debugging – as a proxy for workload information.

To summarize, learning workload distribution is often feasible in today's cloud environments, and can potentially be used to design adaptive scheduling policies.

2.3 2D Scheduling Policy

In this section, we describe the 2D[1] scheduling policy, under a generic setting where jobs arrive at a bottleneck server. The realization of 2D for network flow scheduling under practical conditions, where the notion of a job is a network flow, is presented in §2.4.

2.3.1 2D Objective

The *primary* objective of 2D is to offer robust tail performance across workloads. Specifically, for a given workload, it should perform at least as well as the known optimal scheme (FIFO or PS) at the tail percentiles (e.g., p90, p95, p99). For example, 2D should perform at least as well as FIFO for a light-tailed workload and

[1] 2D refers to the two scheduling primitives (dimensions) used: serialization and multiplexing

for a heavy-tailed workload, it should perform at least as well as PS.

A *secondary* objective for 2D is to minimize the lower percentiles and average completion times as well. Specifically, we want to be as close to SRPT as possible, given that it is optimal for minimizing average job completion times.

Putting these objectives together, 2D aims to be optimal at the tail while not significantly compromising on performance at the lower percentiles.

2.3.2 2D Design Principles

Our earlier analysis of scheduling policies (§2.2) highlights a tension between two scheduling primitives: *serialization* (offered by FIFO and SRPT) and *multiplexing* (offered by PS). Multiplexing is effective at avoiding the *head-of-line* (HOL) blocking problem (small jobs waiting behind a large job) in FIFO and the *starvation* problem (large job starved by a continuous arrival of small jobs) in SRPT. However, multiplexing causes completion times of all similar sized jobs arriving in a burst to increase. This leads us to two design principles that inform us on how to use serialization and multiplexing:

❶ Multiplexing should only occur between jobs of varied sizes (unlike PS which does multiplexing across all jobs).

❷ Serialization should only be done within a group of similar sized jobs.

Principle ❶ is crucial for tail optimality for heavy-tailed workloads (benefit of PS) while ❷ is similarly important for light-tailed workloads as well as for meeting our secondary objective of minimizing the lower percentiles.

2.3.3 2D Overview

Inspired by the benefits of serialization and multiplexing, we present 2D which embodies the two design principles to achieve our desired objectives. At a high level, 2D maps jobs into classes using thresholds calculated in an online fashion. These thresholds are computed such that job size variation within each class is low.

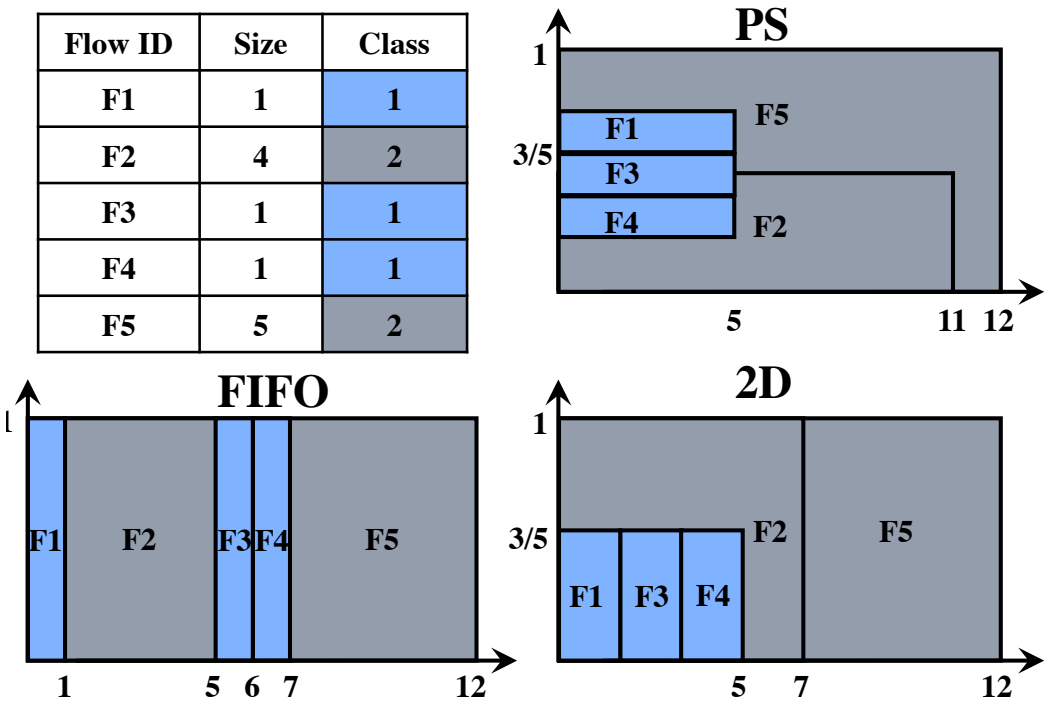


Figure 2.2: A simple toy example showing benefits of serialization and multiplexing. Flows F1-F5 all arrive at the same time ($t = 0$). In the example, F1,F3, and F4 are "small" flows and belong to class 1 while F2 and F5 are "large" flows and belong to class 2. FIFO suffers from HOL blocking. PS stretches completion time of short jobs to 5 units. 2D improves average completion time of short jobs and is no worse than PS for all flows.

Intra-class scheduling. Once similar sized jobs have been grouped into classes, 2D serves jobs within each class in FIFO order (serialization). The choice of FIFO – compared to SRPT or other scheduling techniques – is motivated by two reasons. First, for similar sized jobs, which is what we expect to find within a single class, FIFO's performance *optimality*, in the mean and tail regimes, has been proved in scheduling theory [97]. Second, it is *efficient* to implement, as we demonstrate in 2D's mechanism (§2.4): it does not require any preemption, which can have high flow switching overhead [104], and can support various optimizations, such as batching.

Inter-class bandwidth sharing. Resource capacity across classes is shared in a weighted fashion (multiplexing). These weights are chosen such that each class receives rate equal to the aggregate rate it would get under a PS discipline. This is to ensure tail robust performance across workloads as we explain in §2.3.5.

Algorithm 1 Segment Job Sizes Based on Variability

```
1: procedure ComputeThresholds(JobList F)
2:     F ← sort(F)
3:     T ← {}                                          ▷ List of thresholds
4:     segmentStart ← 0
5:     N ← length(F)
6:     while segmentStart < N do
7:         for i ← segmentStart + 1 to N do
8:             segment ← F[segmentStart : i]
9:             C² ← Var(segment)/Mean(segment)²
10:            if C² ≥ 1.0 then
11:                T.append(F[i − 1])                  ▷ Mark segment boundary
12:                segmentStart ← i                    ▷ Start new segment
13:                break
14:            else if i == N then
15:                T.append(∞)                         ▷ Final threshold
16:                return T
17:    return T
```

Figure 2.2 shows the working of 2D compared to FIFO and PS. We have jobs of two different sizes (small and large) arriving in an online fashion. With FIFO, we observe the head-of-line blocking problem while with PS, we observe unnecessary multiplexing between jobs of similar sizes. 2D divides them into two classes: multiplexing across classes avoids the head-of-line blocking problem while serialization within each class avoids unnecessary multiplexing.

The above example shows that even for a fixed workload, 2D can provide additional benefits at the lower percentiles while being at least good as the optimal policy at the tail. If the workload changes, 2D adapts accordingly – for example, if the workload changes to all small jobs (or all large jobs) in the above example, they will be mapped to a single class and will be scheduled in a FIFO order, which is the optimal policy for a light-tailed workload.

While the utility of simultaneously leveraging serialization and multiplexing is evident from the above example, there are two critical decisions that the 2D scheduler needs to take: i) *how to pick class thresholds*? (§2.3.4) and ii) *how to assign rates across classes*? (§2.3.5).

2.3.4 Class Thresholds

The benefits of 2D depend on accurately splitting jobs into classes, such that intra-class size variation is low. Getting this right is important: if we use too many classes then each job would be mapped to its own class, and 2D would degenerate to PS; if we choose too few classes, 2D's performance would be similar to FIFO. In addition to the number of classes, it is equally important to pick the right thresholds which determine the mapping of jobs to classes. Inaccurate thresholds can cause similar sized jobs to be mapped to different classes, resulting in unnecessary multiplexing; or even worse, it could make jobs varied in sizes to share the same class, leading to potential head-of-line blocking. Thus, the class threshold problem has two variables that need to be picked smartly: i) number of classes to make and ii) splitting thresholds.

The class threshold algorithm (Algorithm 1) divides a workload, potentially with high variability, into smaller chunks (classes) whose individual variation is low. We choose the Squared Coefficient of Variation (C^2) as our metric to quantify whether a class has high or low variability [204].

Algorithm. Overall, our thresholding algorithm is designed to slice a given distribution into k sections such that $C_k^2 < 1$ (Step 7 in Algorithm 1). Note that the algorithm automatically identifies the number of classes it must create in order to sufficiently reduce C_k^2 to at most 1.0. With this low variability in job sizes, jobs within each class can be safely scheduled in a FIFO fashion.

Choosing C^2 to group jobs. Choosing this metric is appropriate because, for distributions with a C^2 value of ≤ 1.0, FIFO has been shown to be optimal in minimizing tail FCTs and has good performance in the average case ([97, 96]). This approach guarantees that intra-class job size distribution will have a C^2 to be at most 1.0 and thus jobs within each class can be scheduled in a FIFO fashion, irrespective of difference in their absolute sizes, without worrying about the HOL blocking problem.

While we have not evaluated other clustering algorithms (e.g., k-means [205]),

we speculate that, for some workloads, they may also perform similar to our principled C^2 approach, but for workloads with no explicit modes (e.g., very long tails), they may perform less well. Additionally, some of the alternate approaches require specifying the number of clusters (classes) upfront, which vary from one workload distribution to another, whereas our C^2 approach automatically figures out the right number of classes.

Convergence. We have validated this thresholding algorithm on network traffic workloads sampled from realistic data center applications (§2.6). Our evaluation shows that typically a sample size of ∼3000 jobs is sufficient to capture the underlying characteristics of a given distribution, leading to the right number of classes and thresholds.

2.3.5 Class Rates

Like class thresholds, computing the right rates is critical for 2D – it is essential for meeting the tail optimality objective of 2D. We note that 2D is work-conserving, so the rates should be viewed as *weights*: if one class cannot use up its rate, it is divided among the remaining classes using the normalized residual weights.

2D follows the PS principle in assigning rate to a given class – each class is assigned a rate proportional to the number of jobs in that class relative to the total number of jobs in the system:

$$Rate_k = N_k(t)/N(t) \tag{2.1}$$

If there is just one class, $Rate_k$ is equal to the full system rate. When there are two or more classes – a typical scenario in heavy-tailed workloads – each class gets a rate proportional to the number of jobs in that class (similar to PS), thus ensuring that the tail completion time is determined by the PS regime. The example in Figure 2.2 also highlights this aspect: the last job in each class of 2D finishes at exactly the same time as it would finish in PS.

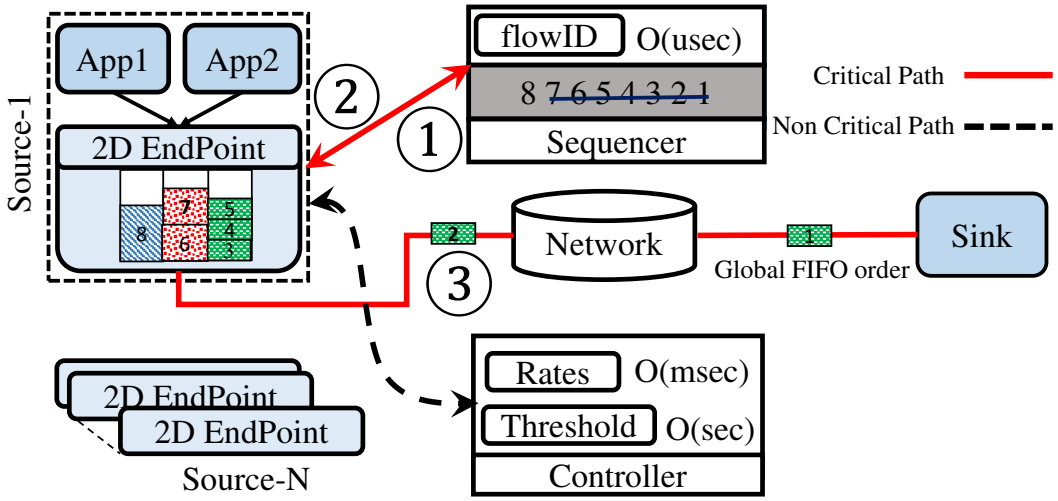


Figure 2.3: 2D's overview showing three network elements involved: i) End points schedule flows locally using assigned thresholds ii) the sequencer ensures global FIFO ordering by assigning unique flow IDs iii) the controller periodically fetches flow size information and recomputes class rates and thresholds.

To provide this strict guarantee, the 2D scheduler maintains a virtual PS scheduler which emulates a PS schedule, enabling us to use the corresponding numbers (N_k and N) from the PS schedule while calculating the rate in Equation 2.1. 2D's principled way of allocating rates, based on PS, is also different from heuristic solutions that target a different goal, such as Aalo [52], which prioritizes short jobs, thereby assigning a strictly higher rate to classes with smaller jobs. Following Aalo's rate allocation would increase the tail completion times of those jobs that fall in the lower classes (i.e., longer jobs) as they get a lower rate, which violates our primary objective of being tail optimal.

2.4 2D Mechanism

We now describe how the 2D scheduling policy, described in the previous section, can be realized for network flow scheduling, taking into account practical challenges of a data center setting.

2.4.1 Feasibility of Using Existing Mechanisms

Given a number of recent proposals for data center scheduling, a natural question to ask is whether we can use an existing mechanism to support 2D. To answer this question, we divide existing solutions into three broad categories, depending on *who* does the arbitration (or scheduling):

- **Switch-based Arbitration.** Each switch makes the scheduling decision for its links; it provides feedback to the sources, who adjust accordingly. Examples in this category include software switch based implementations, like PDQ [104] and Baraat [66], as well as recent proposals that use P4 to implement simple arbitration logic [179]. A key challenge for these solutions is to support rich scheduling policies while operating at line rate.

- **Centralized Arbitration.** The arbitration decision is made by a (logically) centralized server, as in FastPass [168] and PASE [145]. While these solutions can support rich scheduling policies, they face scalability challenges as well as cause additional latency for short flows by involving the server on the critical path of flows.

- **End-host based Arbitration.** Solutions like pHost [78] make the end-hosts responsible for making the scheduling decisions. While this avoids the problems associated with the above two approaches, the scheduling decision is typically sub-optimal because of the limited view of the end-points.

Each of the above categories has its own strengths, which we exploit in 2D's *hybrid* mechanism design.

2.4.2 2D Mechanism Overview

Our design is based on the insight that 2D makes multiple decisions at different timescales – by decoupling these decisions, we can design a simple, highly customized solution for our needs. Specifically, threshold and rate computation is only required at coarser timescales – on the order of seconds or even longer – and can be offloaded to

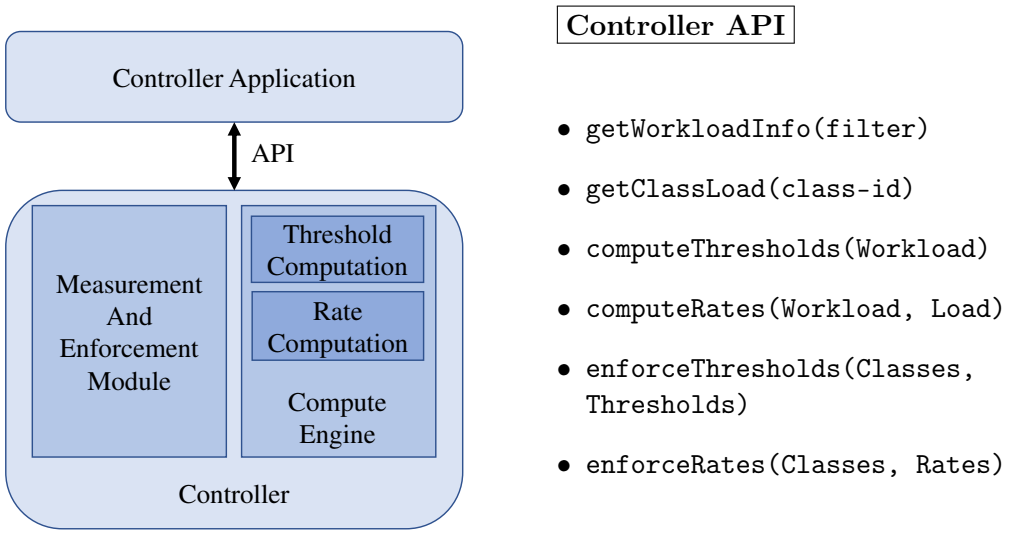
a centralized controller, while flow serialization needs to be done on the critical path of flow arrivals and departures (on the order of μs), and is the most demanding from a mechanism's perspective. By decoupling flow serialization from other components, we can focus on making it scalable and efficient, leveraging several FIFO specific optimizations, instead of aiming to support general purpose arbitration as is the case with existing mechanisms.

Figure 2.3 shows the high level working of 2D's mechanism, illustrating the decoupling of 2D's scheduling decisions. A controller gathers flow statistics and load information from end-points; it calculates class thresholds and rates, as described earlier in §2.3, and communicates these decisions to the end-points (§2.4.3). Unlike centralized arbitration solutions like FastPass [168] and PASE [145], all the controller functionality is *off* the critical path. On the critical path of flows is 2D's *transport*: it maps flows to the appropriate classes and enforces FIFO ordering within a class; this includes a local FIFO ordering for its own flows as well as a global FIFO ordering with the help of *sequencer(s)* (§2.4.4). The transport also uses multiple *optimizations* to make the serialization process scalable and efficient for workloads involving short flows (§2.4.5).

2.4.3 2D Controller

The 2D controller needs to compute and enforce class thresholds and rates. We have designed a modular controller architecture, with clear separation between a generic compute module and a network specific measurement and enforcement (ME) module. These modules are used by a light-weight 2D scheduling application, which decides *when* and *how* to use the underlying modules.

Figure 2.4a shows the controller architecture, highlighting the two key modules and their interaction with the scheduling applications. Using a high-level API (Fig 2.4b), a 2D scheduling application uses the underlying modules in the following way: It queries the ME module using `getWorkloadInfo()` and `getClassLoad()` calls. Workload and load information is returned in a unit-free format, supporting a generic, high-level API. The application computes class thresholds and rates, using



(a) Controller Design

(b) Controller API

Figure 2.4: 2D Controller comprises of two key components: i) Measurement and Enforcement module (ME), and ii) Compute Engine (CE). ME is responsible for gathering flow size statistics and 2D policy enforcement. CE runs the threshold (§2.3.4) and rate computation algorithms (§2.3.5).

the `computeThresholds()` and `computeRates()` calls, respectively. Finally, the application enforces thresholds, using the `enforceThresholds()`, and rates, using the `enforceRates()` call.

There are three important points to note in the above interactions. First, the application controls how these modules are used, in terms of timing and workload granularity. For example, in 2D, class thresholds are calculated on the order of seconds or minutes while the rates can be calculated more frequently. Second, the compute modules are generic: they can be reused across different workload types (e.g., flows, coflows, disk IO workload, etc). Third, the implementation of the ME engine is highly specific to the environment – in our case network flows – but the interface is high-level and can support other implementations, which are customized for a particular environment. While OpenFlow [142] is a natural choice for an ME module for network flows, for ease of prototyping, we have implemented a light-weight, customized protocol between end-points and the controller (§2.5.2).

2.4.4 2D Transport

2D transport sits in between applications and traditional transports (e.g., TCP). Its role is to enforce a FIFO ordering of flows within each class, ensuring that only the flow whose turn has come is handed over to the underlying transport while other flows wait in the queue. It assumes that flow size information is available at flow arrival – either explicitly provided by the application, using an extended socket API [206], or guessed using heuristics, such as using the occupancy of sender buffer as a proxy for flow size [195, 57]. Based on the flow size and class thresholds provided by the controller, each flow is mapped to its appropriate class.

Within each class, a FIFO order is enforced. A local FIFO order is straightforward to implement as the transport knows the arrival time of each of its local flows. The global FIFO ordering, however, involves *coordination*. We use sequencer(s) to implement coordination amongst end-points, enabling a loose, global FIFO ordering for flows.

On a flow's arrival, the transport gets a globally unique flow-id from a sequencer, which generates these ids using a monotonically increasing counter [66]. For improved scalability and performance, we can also use multiple sequencers, each one generating a different set of ids. While this provides a "loose" FIFO ordering of flows, the performance penalty is minimal, and can be further reduced through coordination between the sequencers [216]. The sequencer logic is simple – we have prototyped it on a commodity server, but recent work has shown how similar functionality can be supported at line rates using P4 switches or RDMA-based networks [127, 118].

The 2D transport uses the flow-id to determine *when* to schedule that flow; the only other information that is required is the latest flow-id to finish, so end-points know whose turn is next. There are multiple ways to get this information, including a *centralized* approach where the sequencer keeps track of finished flows, and end-hosts probe the sequencer periodically (similar to PDQ's probing[104]); a *decentralized* solution where each end-host estimates a suitable wait time based on the gap between its successive flow-ids; and a *hybrid* approach where a flow that is

about to finish broadcasts its id to the other end-points, allowing the next-in-line flow (based on the global ordering) to start. Our prototype supports the decentralized and hybrid modes – while their performance is comparable in our experimental setup, the latter is more friendly to the batching optimization, which is useful for scenarios involving short flows (§2.4.5).

Finally, once a flow's turn arrives, it is handed over to the underlying transport for transmission on the network. While the rate achieved by the flow depends on the class rate, we expect the underlying transport to be work-conserving: it should have the capability to use any spare capacity that may be available. For ease of implementation, our prototype uses TCP as the underlying transport but there is potential to use more aggressive transports that can start directly at the assigned class rate (similar to PDQ [104] and PASE[145]).

2.4.5 Optimizations for Short Flows

Our target workloads include applications with small flows – on the order of few KBs or even less. For such workloads, the 2D transport's basic design could be inefficient: short flows, especially those lasting less than an RTT, will incur the overhead of getting a flow-id as well as flow switching. This could hurt performance. We address this using three well-known optimizations: *batching*, *local-sequencing*, and *multiplexing*.

In batching, multiple successive flows arriving at an end-point can use the *same* flow-id, essentially transforming themselves into a larger flow. For example, suppose an end-point has retrieved a flow-id for a flow; while that flow is waiting for its turn, if more flows arrive in the same class, they can use the same flow-id, forming a batch. As a result, end-hosts need not go to the sequencer on every flow arrival. Batching is effective in our settings because each class has similar sized flows, so strict ordering of flows is less important, and loose ordering of flows provides acceptable performance. Further, batching is more effective when the load is high as there are more opportunities to batch flows that are waiting to be scheduled.

In local-sequencing, flows that are extremely short (e.g., lasting less than 1

RTT) do not use the global FIFO sequencing; instead, they only follow a local FIFO order, so the flow at the head of the queue is scheduled without waiting for its turn according to the global FIFO order.

Finally, we have multiplexing of really short flows: we send them simultaneously, rather than one at a time, in order to saturate the rate assigned to the class these flows belong to. In some cases they may have unique flow-ids while in others they may belong to a batch and share a single flow-id. This improves work conservation in scenarios where burst of short flows arrive. This decision is purely local and applied only to classes with small flows: those that last less than one RTT. For the workloads we evaluate, this optimization is only applied to the top class.

2.5 2D Prototype

We have implemented a prototype to evaluate 2D and other schemes.

2.5.1 2D Transport

We have implemented a 2D transport in C, and integrated it with an in-memory data-parallel network traffic generator. The traffic generator supports request-response style interaction between clients and servers, and has been used in several prior studies (e.g., ClickNP [125], MQ-ECN [34], FUSO [45], etc). It is is multi-threaded, supports generating different types of workloads, and uses TCP as its underlying transport. We add 2D transport support on the client, just above TCP. Finally, we have also implemented a light-weight module inside the transport that interacts with the sequencers and keeps track of UDP broadcast messages (for flows that are finished).

2.5.2 Controller

Our controller is implemented in Python; it supports the key modules described earlier. The ME module has specific support for gathering flow sizes and load information from end-points, as well as for enforcing class thresholds and rates at the

end-points. At the end-points, an ME agent is responsible for interacting with the 2D transport: this includes getting flow size information as well as load from the transport, and returning the class thresholds to the transport. To enforce per-class rates, the agent leverages Linux's support for hierarchical token buckets (HTB), which enables rate control based on the DSCP value assigned to each flow [175]. The transport sets the DSCP value according to the flow's class.

2.5.3 Other Scheduling Techniques

In addition to the baseline TCP (PS) and 2D, we have also implemented other scheduling techniques in our prototype. Our prototype supports basic scheduling techniques (PS, FIFO, Non-preemptive Shortest Job First (NP-SJF)) as well as state-of-the-art heuristic schedulers: PIAS [33], Baraat [66] and Aalo [52]. While both baraat and Aalo have been evaluated for co-flows/tasks, their underlying scheduling heuristics (i.e., FIFO-LM, D-CLAS) are applicable to flows as well. Implementing all these techniques was relatively straightforward as they leveraged 2D's support for FIFO and PS.

- **FIFO**. To support FIFO, we use 2D with just one class.
- **Baraat (FIFO-LM)**. Based on the workload distribution, we pre-establish the number of classes Baraat will require, using a threshold of p90 of the workload distribution to differentiate between small and heavy flows, as instructed by the authors. Subsequently, when a flow arrives, it is either dispatched to the class with small flows (if it is a small flow) or one of the empty classes.
- **Aalo (D-CLAS)**. Similar to Baraat, we pre-establish a fixed number of (exponentially spaced) classes to use. For a given workload, the thresholds and rates for each class are chosen based on Aalo's D-CLAS algorithm.
- **PIAS**. To support PIAS, we implemented strict priority scheduling across classes and disabled both local and global FIFO flow ordering. Following the workload distribution, we created eight priority classes and configured their thresholds based

on the authors' implementation[2].

2.6 Evaluation

We evaluate the performance of 2D under realistic data center workloads [87, 28, 176], through various experiments on the Emulab testbed [4]. The key goals of our evaluation are to: i) validate that 2D is tail optimal across different application workloads, ii) provides opportunistic gains at lower percentiles whenever possible, and iii) can track and react to *changes* in workloads.

Our results show that:

- 2D is tail-robust while lowering FCTs at lower percentiles for both heavy and light-tailed workloads, in comparison to tail-optimal scheduling policies (§2.6.2).
- Systematic threshold selection and rate division is important. One-size-fit-all approaches (e.g., Aalo [52]) and other heuristics (e.g., Baraat [66]) can result in performance loss by up to 4× (§2.6.2).
- 2D's learning mechanism ensures that suitable thresholds and class rates are picked, ensuring robust performance in the presence of multiple different workloads (§2.6.3).

2.6.1 Methodology

Experimental setup. We use the Emulab testbed [4] for all our experiments. We setup a 21 node cluster, with 18 clients, 1 server, 1 sequencer and 1 controller. All nodes are Emulab d430 machines with network interfaces supporting 10 Gbps.

Benchmark workloads. We use three different realistic workloads, derived from patterns observed in production data centers, and a synthetic uniform distribution (∼U[1MB, 15MB]). The first workload is from a web search application containing a mix of short and long flows (DCTCP [28]). It is a relatively high variance distribution

[2]Our PIAS evaluation is limited to the data mining and web search workloads due to the lack of threshold information for other workloads.

with a $C^2 \sim= 5$. The second flow size distribution is from a data mining cluster, which is extremely skewed: 80% of the flows are less than 10KB while the largest 2% of the flows can be as large as 1GB (VL2 [87]). It has a $C^2 \sim= 45$. The third workload is taken from Facebook's Cache cluster[3], which follows a light-tailed distribution with a $C^2 \sim= 3$ [176]. Flows arrive according to a Poisson process. Unless otherwise specified, our setup runs at 80% load w.r.t. the bottleneck link (server link).

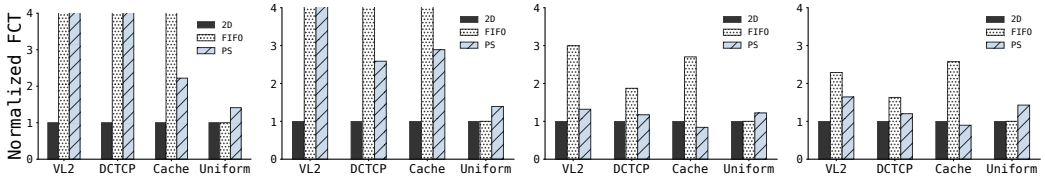
Metrics. Our primary metric is Flow Completion Time (FCT) at higher percentiles (p90, p99, p99.5), and the median (p50). We choose this array of metrics to show that 2D tries to perform as well as the tail-optimal scheduling policy for a given workload, if not better, while also showing FCT improvements at lower percentiles wherever possible. Our reported gains are the average of five runs and the results for each run are within 10% of the reported average.

Schemes. We compare 2D with three scheduling disciplines: FIFO, PS and NP-SJF; and three scheduling heuristics: Baraat, Aalo and PIAS. The first comparison point shows whether 2D is able to attain equal or better tail completion times compared to disciplines for which scheduling literature makes optimality claims. The second comparison point is to show the versatility of 2D across workloads compared to Aalo's D-CLAS scheduler, which is aggressive in creating classes, Baraat's FIFO-LM, which multiplexes similar sized long flows, and the loss in performance due to strict prioritization under PIAS. The precise implementation details to realize these schedulers have been mentioned in §2.5.3.

2.6.2 2D's Overall Performance

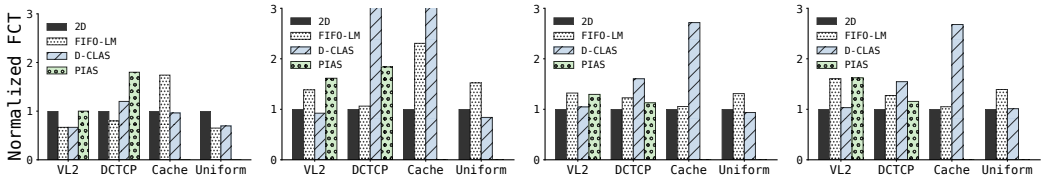
We evaluate 2D across diverse application workloads and performance metrics, comparing it against foundational scheduling techniques (FIFO, PS, and NP-SJF; Fig. 2.5) as well as heuristics (Aalo, Baraat, and PIAS; Fig. 2.6).

[3]We scale the distribution by a factor of 100 to ensure client applications do not become bottlenecked due to extremely small inter-arrival times.



(a) Median (b) p90 (c) p99 (d) p99.5

Figure 2.5: FCTs across different workloads and percentiles showing 2D in comparison with scheduling disciplines. Note that these results are normalized by 2D's FCTs.

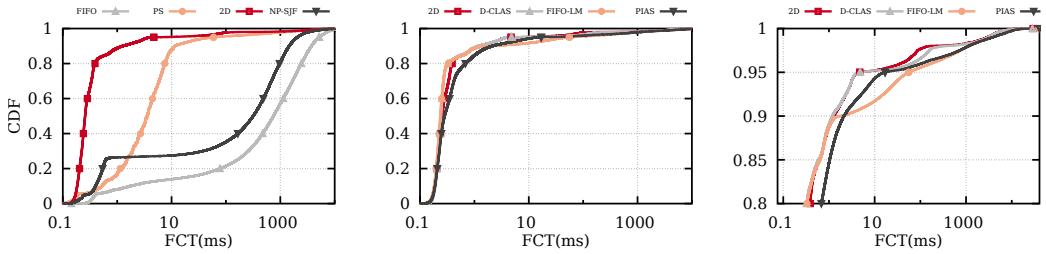


(a) Median (b) p90 (c) p99 (d) p99.5

Figure 2.6: FCTs across different workloads and percentiles showing 2D in comparison with scheduling heuristics. Note that these results are normalized by 2D's FCTs.

2D vs. well known scheduling policies. For tail FCTs (p99.5), 2D, even in the worst-case, remains within 9% of PS and FIFO, while improving other completion times by up to 2.6× (p90), 1.7× (mean) and 12× (median) at its best. We observe that when the workload has low variance (e.g., Uniform and Cache), 2D keeps multiplexing low during intervals when similar sized flows exist in the system, while PS multiplexes aggressively, leading to an increase in mean and tail FCTs. 2D is able to intelligently bifurcate workloads into regions of low variance, and applies serialization within each class. During epochs when multiplexing is required, the chosen thresholds in 2D ensure that disparate flows are mapped to *different* classes.

To elucidate this further, we zoom into the heavy-tailed data mining workload (Fig. 2.7a) for which PS is tail-optimal. We see that 2D meets its promise of being tail-robust while showing improvement over PS. By separating the largest flows from the smaller ones and serializing flows within each class, per-flow latency is significantly reduced. NP-SJF shows better performance than PS for lower percentiles; however since it doesn't support preemption, short flows can still get stuck



(a) 2D vs scheduling policies (b) 2D vs scheduling heuristics (c) Zoomed in CDF. 2D vs scheduling heuristics

Figure 2.7: CDF of FCTs under the (heavy-tailed) data mining workload.

behind large ones, so beyond p20, its performance degrades. FIFO suffers from HOL and performs poorly at the tail and lower percentiles.

2D vs. Contemporary Schedulers. We now compare 2D's performance with heuristic based flow schedulers that also attempt to use multiplexing and serialization. We compare their performance against 2D on two accounts: i) robustness for tail FCTs across different workloads and ii) relative performance improvement at lower percentiles. We assume knowledge of flow size information for Aalo's D-CLAS scheduler, PIAS's MLFQ scheduling policy, and Baraat's FIFO-LM.

2D improves FCTs by 3.6×, 10.3×, and 2.7× over Aalo when the workload has light-tailed characteristics (Cache) at the mean, p90, and p99, respectively. D-CLAS works well when the workload is heavy-tailed; however, its priority based rate division across classes forces it to perform poorly at higher percentiles for workloads containing several medium to long flows, which are given a lower rate, elongating their FCTs. This observation also holds for the web search (labelled as "DCTCP") workload, which contains a significant fraction of medium to long flows whose completion times are stretched under a priority based rate allocation strategy.

2D improves over Baraat by up to 1.5× at the tail (p90, p99.9) under the data mining workload, by avoiding multiplexing between long flows (Fig. 2.7c). FIFO-LM shows improvement for lower percentiles as 90% of the flows are very short and thus, are serialized; however, for workloads with low variance, the threshold set to p90 of the flow-size distribution proves to be an incorrect heuristic, causing several long

flows to be multiplexed resulting in tail (p90) FCTs to be 2.3× those achieved by 2D.

Even though PIAS is optimized for workloads with high variation, at higher percentile, for both data mining and web search workload, 2D improves FCTs by 1.6× and 1.8× at p90 over PIAS. This is because of the strict prioritization mechanism of PIAS, which is detrimental for medium to long flows, as is evident for the web search workload, which has several medium to long flows. We hypothesize that the gains of 2D over PIAS may even be more for Cache and Uniform workloads, as they are less heavy than data mining and web search. Overall, we observe that these scheduling heuristics have varied performance, unlike the robust performance offered by 2D.

To further understand this behavior, we analyze the performance of Baraat and Aalo under the Cache workload (Figure 2.8). At lower percentiles, Aalo achieves comparable completion times – due to its priority based rate division across the created classes – at the expense of long flows, which drive up the tail FCTs by 10× (2.7×) at p90 (p99), so much so that Aalo's D-CLAS scheduler performs worse than PS by a factor of 2.17× at p95 (an outcome we cannot afford with 2D!). Similarly, the multiplexing threshold chosen proves to be too conservative for Baraat's FIFO-LM scheduler, resulting in serialization between different sized flows, thus hurting performance.

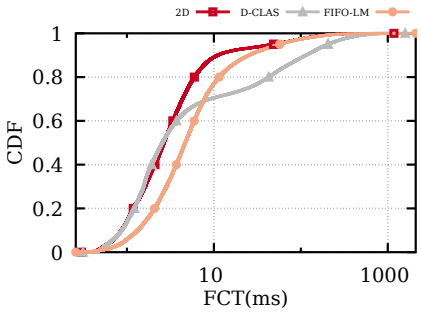
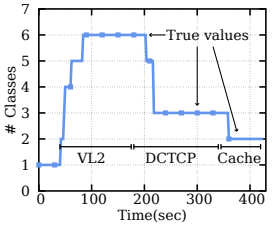


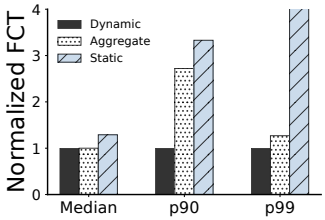
Figure 2.8: CDF of FCTs for the (light-tailed) Cache workload.

2.6.3 2D under changing and mixed workloads

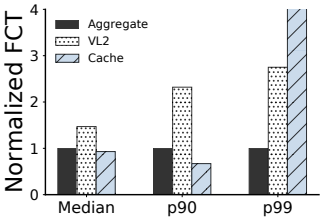
We now evaluate the ability of our controller to detect workload changes, and adapt class thresholds and rates, in an *online* fashion. We consider two scenarios, based on our discussion in §2.2.1, where such a system can be useful. First we look at a cluster-reuse setting; applications with different workloads use the same cluster one after the other. Second, we consider the case of heterogeneous clusters where



(a) Threshold detection



(b) Changing workloads



(c) Mixed workloads

Figure 2.9: 2D Controller detects workload changes and adjusts class thresholds (a) and rates accordingly allowing it to perform better than other strategies (b). FCTs for different strategies for the mixed workload scenario (c); the learning approach beats the strategies that use thresholds and rates corresponding to one of the possible workloads.

applications with different workloads are co-located in the same cluster. Finally, we evaluate 2D's robustness in face of changes in system load.

Cluster-Reuse. In our experiment, we introduce three very different types of workloads sequentially. At the start, the controller bootstraps itself with a single default class. We start the data mining (VL2) workload at 30 seconds; at 170 seconds, we switch to the DCTCP workload; and finally, we introduce the Cache workload at 350 seconds.

We compare the performance of 2D with controller (Dynamic) versus a realization of 2D that: i) simply uses the thresholds and rates, computed offline, for the data mining workload (Static), and ii) uses thresholds and rates computed, offline, according to the aggregate workload generated from these three workloads (Aggregate).

Figure 2.9a shows the ability of the controller to learn workload changes and adjust thresholds accordingly. We report the number of classes determined by the threshold algorithm, and corroborate this with the number of classes calculated in

an offline setting - referred to as "True value" in the figure.

Figure 2.9b shows that using thresholds and rates computed for one workload do not work well for others. It also shows that considering the aggregate workload when computing thresholds and rates does not ensure robustness across performance metrics; they only work well for some percentiles. In comparison, the controller enables 2D to achieve the best median and tail completion times.

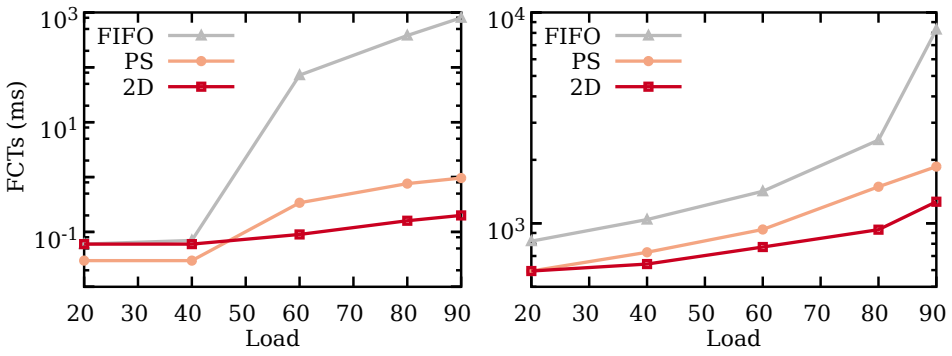
Overall, we make the following observations regarding the results:

- The controller is able to learn the workload characteristics, whether it is a new workload that is introduced at the start, or a change in workload. This validates 2D's ability to provide robust performance across changing workloads (Fig.2.9a).
- Dynamic 2D improves tail FCTs by 3.33× and 2.72× at p90 compared to the static and aggregation strategy (Fig.2.9b).
- The controller converges to the optimal settings fairly quickly, within a few iterations for all workloads.
- The convergence times depend on the workload characteristics. Workloads that have large number of classes, with some of the classes having low occupancy, require more time because the controller needs a larger sample to cover the full range of flow sizes. This explains the difference in the (relatively) larger convergence time for the VL2 workload (which has the most number of classes) and the Cache workload (which has the least number of classes).

Mixed Workloads. We also evaluate the benefits of learning workloads in the context of heterogeneous clusters - running co-located applications with different workloads. Note that unlike the previous experiment, in this experiment, workloads perfectly overlap. To realize this setting, we allocate nine nodes to generate traffic according to the Cache workload while the other nine generate it according to the data mining workload. We compare the performance of two strategies: i) 2D with controller computing class rates and thresholds in an online fashion, and ii) 2D

without controller, using thresholds and rates computed for cache and data mining workloads in isolation.

Figure 2.9c shows the normalized FCTs achieved in this mixed workload setting. With the controller enabled, 2D is able to learn the aggregate workload and set the best thresholds and rates. Simply picking these according to one of the workloads can be a naive strategy and results in poor performance. 2D is robust in comparison to a strategy that either uses thresholds and rates computed for cache or the data mining workloads.



(a) Median (b) p99

Figure 2.10: 2D vs PS and FIFO across varying system load for the data mining workload at low and high percentiles.

Changing Load. Finally, figure 2.10 shows 2D's performance across varying network loads under the data mining workload. Observe that 2D performs better than PS across all loads in terms of median and tail FCT (p99). However, at low loads, 2D achieves higher median FCT than PS due to the sequencing overhead as batching opportunity is low. At high loads, this overhead gets amortized through 2D's sequencing optimizations.

2.7 Related Work

Network Flow Scheduling. Our work is inspired by a large body of recent work on network flow scheduling. Throughout the paper, we have highlighted the differences between 2D and other network scheduling proposals. Here we summarize the

key differences in terms of their scheduling policy and mechanisms.

2D's scheduling policy objective is unique: it considers *robustness* across workloads and performance metrics. To achieve these objectives, it uses serialization and multiplexing in a principled way, effectively combining the benefits of FIFO and PS. Its principled approach to rate and threshold calculation also sets it apart from contemporary data center scheduling schemes [33, 52, 66].

2D's mechanism also leverages key ideas from existing techniques, but is customized to support the combination of PS and FIFO. Similar to centralized approaches (e.g., FastPass [168] and PASE [145]), 2D uses a controller, but its controller is not on the critical path of flows, and operates at coarser timescales. Similar to end-host based approaches (e.g., pHost [78]), 2D uses local FIFO scheduling, but it is supplemented by a loose global FIFO ordering. For global FIFO coordination, 2D uses FIFO specific optimizations. Our hybrid framework is most similar to Aalo – the key difference is in the global FIFO ordering (in addition to the fundamental differences in policy which we described earlier).

Cluster Scheduling. There has been significant recent interest in scheduling policies for compute clusters [102, 160]. We believe 2D can provide benefits in those scenarios as well, although its mechanism would need to be adopted for the cluster setting. A common approach to implementing different scheduling policies in compute cluster is to use a centralized manager that can support a range of policies, including FIFO and PS. Our 2D controller can be integrated with the cluster manager, and used in scenarios where workload distribution is known. A more challenging use-case for 2D adoption in compute clusters is decentralized scheduling (e.g., Sparrow [160]) which is often optimized for a particular scheduling policy. However, the decoupling approach we take in 2D may potentially be applicable in those scenarios as well.

Other Tail Latency Proposals. Improving tail latency is an important problem that has attracted a wide range of solutions. Most popular non-scheduling based solutions include the use of replicas, either through intelligent replica selection techniques

or through use of redundant requests to different replicas [189, 108, 58]. Scheduling-based proposals, including 2D, are orthogonal to these techniques: proposals like 2D focus on deciding the best schedule for a request that has *already arrived* whereas the other category of proposals focus on *where* to send the request.

2.7.1 Subsequent Work

The related work described above reflects the state of the field at the time of 2D's publication in 2018. Since then, the design space of network scheduling and congestion control has expanded significantly, with a growing emphasis on adaptive and learning-based mechanisms. Below we highlight two strands of follow-on research most relevant to 2D's focus on workload sensitivity and tail performance.

Learning-based Network Resource Management. Building on the idea that static techniques are insufficient for diverse workloads, several proposals have explored the use of learning to drive network resource management (e.g., scheduling decisions). AuTO [46] applied reinforcement learning to data center traffic engineering, demonstrating how data-driven methods can dynamically respond to changing traffic patterns. NetLLM [209] proposes adapters for Large Language Models (LLMs) for various networking optimization problems, including cluster job scheduling. Pappone et al. show how learning can be used to intelligently select from and combine existing congestion control algorithms [161]. Other systems have employed simulation-guided or adaptive strategies to select among scheduling primitives or to tune parameters at runtime [139, 121]. These works reflect a broader shift toward workload-adaptive design, a theme 2D anticipated in showing how tail-optimal scheduling could be achieved robustly across light- and heavy-tailed workloads.

Congestion Control for Modern AI Workloads. In parallel, the rise of large-scale distributed machine learning has motivated congestion control mechanisms specialized for training workloads [196]. These proposals often incorporate learning or workload feedback for enhanced performance. For example, CASSINI [171]

adapts congestion windows based on iteration-level traffic patterns. Such systems highlight how the principle of workload-aware adaptation, as emphasized in this chapter, has become increasingly central in the design of networking support for AI infrastructure.

2.8 Discussion

We now reflect on broader implications, extensions, and limitations of 2D's design. While 2D targets the core challenge of providing robust flow scheduling under varying workloads and performance objectives, its architecture opens the door to further applications and adaptations. In particular, we explore how the ideas behind 2D could generalize beyond the current scope — to settings like coflow scheduling, non-clairvoyant environments, and spatially heterogeneous data centers.

2D with Coflows. While the focus of our mechanism has been on flows (rather than coflows), 2D's policy – and its gains – can potentially extend to a coflow setting as well. This, however, may require non-trivial changes in the mechanism, depending on whether coflow scheduling is done in a centralized or distributed fashion. For centralized co-flow schedulers (e.g., Varys [54]), 2D's scheduling policy can easily be implemented at the centralized controller, which computes the rates according to the 2D policy, and the end-points can enforce this rate. For distributed co-flow scheduling (e.g., Baraat [66]), 2D's current mechanism will require substantial changes, including a new flow-to-class mapping mechanism based on the coflow id rather than the flow size. Similarly, the serialization mechanism will also need to obey a FIFO order based on coflows (rather than flows). On the controller side, the compute engine will remain the same as it is agnostic of the notion of flows (or coflows) but the ME engine will need to collect coflow statistic rather than flow-level statistics.

2D without Flow-size information. 2D's current scheduling policy requires a-priori flow size information, Which may not be available in all scenarios [33, 52].

We foresee the potential to estimate or learn flow size information using machine learning techniques. Traditionally, non-clairvoyant schedulers use the age of a flow (bytes sent so far) to estimate its actual size. In addition, attributes that may prove useful include: spatial attributes (e.g., source-destination, intra-rack vs. inter-rack flow) and temporal attributes (e.g., over 40% big-data jobs in compute clusters are recurrent and have similar characteristics [112]). These features can potentially be used to make predictions about a flow's size.

Capturing Workload Variation Across Space. Another dimension to the variability in workloads is the *spatial* dimension – when workloads differ at different locations of a cluster or a data center. Arjun et al. show that for Cache and Hadoop clusters, the intra-rack and inter-rack flow size distributions are very different [176]. 2D's control mechanism can be potentially extended to learn workloads at different granularities (e.g., rack level, cluster level, etc) across the data center fabric, with suitable updates to class-rates and thresholds. For example, for flows traversing several hops, 2D can use class rates and thresholds of the bottleneck link for such flows.

Multiple Bottlenecks. While we focus on single bottleneck scenarios in our evaluation, 2D's policy can be extended to multi-hop settings. Virtual network partitioning for inter-rack and intra-rack communication can allow 2D's policy to apply to flows that traverse multiple hops.

2.9 Conclusion

Existing flow scheduling schemes for data centers optimize for a specific workload and performance metric. We proposed 2D, a new learning-based scheduling framework that is robust across metrics and changing workloads – a ground existing scheduling policies are unable to cover. To achieve this, 2D combines multiplexing and serialization in a principled way, ensuring tail optimal performance across workloads while also improving the lower percentiles. With 2D, coarse time-scale decisions based on

workload and load changes are made by a centralized controller whereas per-flow serialization decisions are made in a distributed fashion involving only the end-points and sequencer(s). Our testbed experiments show that, for realistic cloud workloads, 2D provides consistent gains at the tail and average flow completion times compared to basic scheduling techniques (e.g., FIFO, PS) as well as recent heuristic-based schedulers (e.g., Aalo, Baraat). We believe that 2D sets out a new direction in data center scheduling, where robustness is elevated as a key metric, and principled form of "learning" as a key strategy; eventually helping with the adoption of such schemes in practice.

Chapter 3

PCS: Predictability-Centric Scheduling

3.1 Overview

In this chapter we make a case for providing job completion time estimates to GPU cluster users, similar to providing the delivery date of a package or arrival time of a booked ride. Our analysis reveals that providing predictability can come at the expense of performance and fairness. Existing GPU schedulers optimize for extreme points in the trade-off space, making them either *extremely unpredictable* or *impractical*.

To address this challenge, we present PCS, a new scheduling framework that aims to provide predictability while balancing other traditional objectives. The key idea behind PCS is to use Weighted-Fair-Queueing (WFQ) and find a suitable configuration of different WFQ parameters (e.g., queue weights) that meets specific goals for predictability. It uses a simulation-aided search strategy to efficiently learn and discover WFQ configurations that lie around the Pareto front of the trade-off space between these objectives. We implement and evaluate PCS in the context of scheduling ML training workloads on GPUs. Our evaluation, on a small-scale GPU testbed and larger-scale simulations, shows that PCS can provide accurate completion time

estimates while marginally compromising on performance and fairness.

3.2 Motivation

Humans desire predictability in their daily lives [143]: from knowing how long their home-to-office commute will be to the arrival time of an Amazon package [177] or an Uber ride [8]. Fortunately, most real world systems (e.g., transportation, e-commerce, etc) meet this need by providing their users with a (reliable) prediction (e.g., estimated delivery date). As more and more of our lives move to the cloud (e.g., Metaverse [155, 92]), it begs the question of whether the cloud can offer a similar notion of predictability? More concretely, when a user submits a "job" (e.g., train a Machine Learning (ML) model) to the cloud, can the cloud provide a reliable job completion time prediction?

According to studies in human psychology [61, 105], such feedback can improve the quality of experience and ease user frustration; perhaps more emphatically than simply making the cloud faster or fairer.

Several aspects of the user-cloud ecosystem can impact the (lack of) predictability of a job's completion time (e.g., failures [114], shared vs. dedicated resources [117], knowledge of individual job sizes [71, 129], workload characteristics etc.). The focus of this chapter is on understanding the unpredictability stemming from the *scheduling* mechanism used by the cloud (sub)systems (e.g., FIFO vs. Fair Sharing vs. other policies). While this discussion has broader applicability in various scenarios (e.g., CPU scheduling, network bandwidth scheduling), we situate it in the context of multi-tenant GPU clusters (e.g., PAI [200], Philly [114], etc) and discuss the opportunities and challenges in supporting predictable scheduling in that context.

3.2.1 Why provide JCT predictions (JCTpred)?

Providing JCTpred can have two broad benefits: (1) Alleviating User frustration. Several studies on real-world systems (e.g., online retail [167], airlines [36]) show that

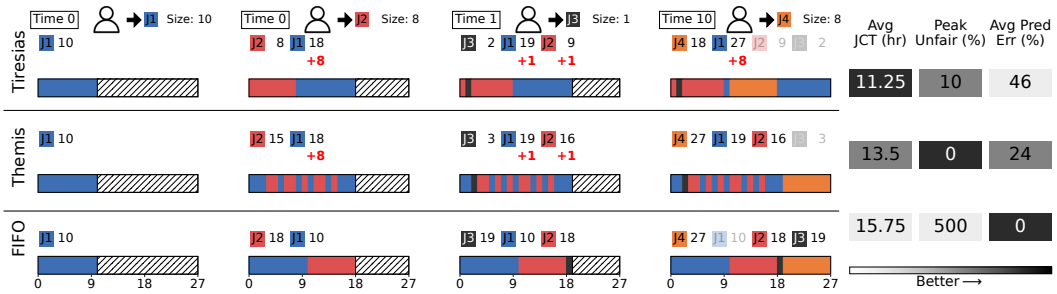
providing a timeline to users can help ease frustration in the face of long and variable waiting times [218, 105, 107]. JCTpred can offer a similar role in the context of multi-tenant GPU clusters where users can suffer from large and unpredictable delays, inevitably leading to a poor and frustrating experience [61, 101]. Measurements on Microsoft's GPU cluster (Philly) show that ML training jobs can face up to 100 hours of queuing and preemption related delays [114], hinting that organizational GPU clusters are heavily oversubscribed. Research shows that users are often trying to guess when their training jobs will complete and that user-driven predictions can be off by more than 100% (i.e., the system takes twice as long to complete the job compared to the user's prediction), with some users finding it *impossible* to make any meaningful predictions [79]. With the paradigm of AutoML, jobs that spawn hundreds of DNN trials [136, 128], and LLMs (e.g., GPT4 [5]) becoming mainstream, these issues will only exacerbate [31]. Additionally, predictability expectations are higher for users submitting repetitive jobs [117] and according to one study, 60% of training jobs exhibit DNN architecture similarity [122], emphasizing the need to provide JCTpred in such scenarios.

(2) Enabling decision making. In real-world systems, if the predicted timeline is long, customers may elect to perform other tasks or seek alternatives [137]. For example, estimated delivery dates can help shoppers decide between e-commerce platforms (e.g., Amazon [2] vs Temu [7]) and even between sellers within a platform. Today's cloud users have similar choices to make and JCTpred can enable them to make these choices in a more informed way. For example, it can help users decide between different cloud systems to run their ML workloads on, each option potentially offering a different cost-JCTpred trade-off. As a forward looking avenue, JCTpred can facilitate the growing eco-system around inter-cloud brokers which orchestrate seamless access to multiple clouds with low user effort (e.g., SkyPilot [214, 187, 109, 55]). Within a cloud, JCTpred can facilitate users in selecting between different model variants/pipelines to train, based on the expected accuracy-JCTpred trade-off [37, 56, 220, 212].

Why are Deadlines not the answer? One may wonder how the predictability metric is different from deadlines (and the large body of work on deadline-based scheduling for GPUs and beyond [79, 49, 50, 126]) where a user provides a deadline along with their job and the system tries to satisfy it. The fundamental difference is that in the deadline-based context, the burden lies on the *user* to provide a timeline to the system, with the system deciding the user's fate. We posit that it should instead be the *system* that provides the user with a timeline (i.e., a JCTpred), empowering them to decide whether it is acceptable or not. Our approach is analogous to real-world systems like ride-sharing where most users request a ride, wanting it ASAP (i.e., no deadline) while the system comes up with the expected arrival time of the ride.

Even if we try to shoehorn predictability into deadlines, it will be challenging for two reasons. First, coming up with a reasonable deadline is hard because the slowdown of a job is highly dependent on: i) cluster load (which can be highly variable and bursty at short timescales) and ii) underlying job-to-resource mapping which is (dynamically) determined at run time [117] and can result in significant variation due to heterogeneity in the underlying resources (e.g., low vs. high end GPUs [200, 43, 149], RDMA vs. TCP [80, 166], etc.,). Second, unless there is an inherent difference in user requirements (and hence deadlines), users have the incentive to specify a small deadline (i.e., to act greedy), which limits any prioritization the system can enable. In both the above cases, the lack of reasonable deadlines will render the system ineffective.

Feasibility of providing JCTpred. Computing JCTpred requires the knowledge of a job's size and its demand function (i.e., how its execution time will change based on the allocated GPUs). Fortunately, several attributes of ML workloads allow us to (approximately) estimate these. (1) Intra-job predictability. DNN training and inference jobs [136, 114, 89] exhibit intra-job predictability; the time it takes to run an inference job [90] or train a DNN for a specified number of epochs is fairly deterministic [136]. By profiling [106, 170, 149, 159] or modelling [165, 136, 226, 80,



(a) Impact of future arrivals on scheduling (b) Summary

Figure 3.1: Toy example with a single GPU illustrating the limitations of existing schedulers. (a) shows how job ordering evolves as new jobs arrive over time under Tiresias [89], Themis [136], and FIFO [193]. Time advances left to right, with jobs arriving at $t = 0, 0, 1$, and 10 units, with sizes $10, 8, 1$, and 8 units, respectively. The expected completion time for each active job (its size plus scheduling delay) is displayed above the schedule; completed jobs are grayed out. Numbers shown in red (e.g., +8) denote the additional delay introduced whenever a new job arrives. (b) summarizes the comparative outcomes of these policies in terms of performance, fairness, and predictability.

213] the job's throughput and combining it with the provided job information (e.g., total number of epochs, convergence criteria, budget), its size and demand function can be estimated. (2) Recurring jobs. ML workloads are known to contain recurring jobs [122, 200, 63]. This can make history [163] based strategies highly effective in estimating job sizes.

3.2.2 Limitations of Existing Schedulers

Reliably predicting the completion time of a user's job requires the underlying scheduling system to be predictable [79]. In this section, we highlight and analyze why existing schedulers used by GPU clusters today are either not amenable to reliable completion time predictions or are not practical.

Unbounded Preemption: the Price of Fairness and Performance. Performance and fairness-oriented schedulers frequently utilize *unbounded preemption* to prioritize and distribute resources among jobs. Preemption collectively refers to when some or all of the resources assigned to a job are reallocated or when its position in the queue is altered because of another (future) job. Although preemption is

essential for achieving the goals of these schedulers, it can lead to unpredictability in a job's completion time [117]. Under preemptive scheduling policies, the arrival of future jobs can affect the completion times of current jobs by preempting the resources (e.g., GPUs) they are using.

Preemption manifests in today's cloud systems in the following ways: (1) Prioritization. When a higher priority job arrives and needs to be scheduled, running jobs are paused or waiting jobs are pushed further back in the queue. Several schedulers use prioritization to minimize JCTs and meet deadlines [89, 165, 79, 130, 29, 50, 126]. (2) Elastic Sharing. Jobs may need to be multiplexed together to achieve fairness and efficiency [136, 210, 43, 81, 106]. As new jobs arrive, the GPU share of existing jobs is reduced, stretching their completion times [117] or the scheduler takes away GPUs from existing less-efficient jobs and assigns them to new jobs that can utilize them more efficiently [106, 38].

Takeaway: Unbounded preemption results in unpredictability, making it challenging to provide a reliable JCTpred. A scheduler which utilizes bounded preemption will be more predictable.

Fixed Trade-offs. The other option is to use non-preemptive schedulers such as First-In-First-Out (FIFO) [193] and reservation based schemes [114] which are highly predictable as they guarantee resource allocation throughout the lifetime of a job — future job arrivals do not impact current jobs in the system. However, such schemes suffer from well known performance issues such as Head-Of-Line (HOL) blocking in the case of FIFO [74, 66, 148, 106, 89] and poor utilization for reservation based schemes [211, 114, 200]. There is no clear way to tune these schedulers that lie on extreme ends, to offer different trade-offs between predictability and other objectives. This is an issue because different cluster operators may want to settle for different (intermediate) trade-offs rather than switch between these two extremes.

Takeaway: Existing schedulers offer a fixed trade-off: predictable but high-/unfair JCTs (non-preemptive) or low/fair but unreliable JCTs (unbounded preemptive). A scheduler which offers different trade-offs between these competing

objectives is more practical.

Motivating Example. We use a simple toy example (Fig. 3.1a) with four jobs (J1-J4) to demonstrate these limitations. We analyze the performance of three schedulers — FIFO, Tiresias, and Themis — on reducing JCTs, unfairness, and unpredictability. FIFO is the default scheduler used in YARN [193]. Tiresias [89] prioritizes DNN training jobs with smaller remaining service times, while Themis [136] strives to minimize peak unfairness.[1] Tiresias and Themis are representative of a large space of policies which either use size based or fair scheduling, respectively. Unpredictability is captured as $\text{Pred}_{err} = \frac{JCT_{true} - JCT_{pred}}{JCT_{pred}}\%$, while unfairness is captured as the additional time it takes for a job to complete compared to its Fair Finish-Time (FFT) [136, 44] in percentage terms. JCTpred is computed at the time of a job's submission and is defined as the time it takes for a submitted job to complete given a scheduling policy and the current cluster state (i.e., GPU allocations to existing jobs). We provide a practical way to compute it for all scheduling policies in §3.4.

As new jobs arrive (moving left to right in Fig. 3.1a), both Tiresias and Themis result in a change in completion times of previous jobs. For instance, in Tiresias (top row), when J2 and J4 arrive in the system (second and fourth column), there is an eight time unit increase in J1's predicted JCT each time. While Tiresias achieves the minimum average JCT, it results in the highest average prediction error — 46% Pred_{err} in our example. Similarly, in Themis (middle row), the scheduler's multiplexing of J1 and J2 causes J1's predicted completion time to increase by eight time units (second column). While Themis ensures all jobs finish before their FFT (unfairness of 0) and also avoids HOL blocking, it has an avg Pred_{err} of 24%. The FIFO scheduler (bottom row) achieves a prediction error of 0 as it is non-preemptive, but is the most unfair strategy and has the highest average JCT. Figure 3.1b summarizes these outcomes.

[1] We use a lease duration of 1 time unit for Themis and assume job size information is known for all schedulers

3.3 Predictability-Centric Scheduling (PCS)

Requirements. Our analysis in the previous section reveals that a scheduling policy with *no preemption* (i.e., FIFO) results in maximum predictability. However, this comes at a high cost in terms of performance (i.e., JCTs) and fairness, which makes it an *impractical* option. On the other extreme, there are scheduling policies that have *unbounded preemption* (e.g., Fair-Share, Shortest Job First, etc.). In these policies, an influx of future arrivals can arbitrarily stretch the completion time of an existing job, making them unsuitable for providing predictability.

This insight distills into the following two requirements that a scheduling policy must satisfy:

- R1 Predictability Requirement: a scheduling policy must have *bounded preemption*. This is essential in order to provide reliable JCT predictions.
- R2 Flexibility Requirement: it should be able to approximate maximum predictability, optimal performance, and maximum fairness. Most importantly, it should be able to achieve Pareto-optimal trade-offs between these. This is essential for practicality.

PCS Overview. Our solution to this end is PCS, a generic scheduling framework (Fig.3.2), which captures these requirements using a high level preference interface (§3.3.2), and meets them by using the well-known Weighted-Fair-Queuing (WFQ) [62] policy in a novel way. The inherent properties of WFQ, careful selection of various WFQ parameters (number of queues, weights, etc) along with how each job is mapped to a queue and processed within it, allow us to meet our objectives (§3.3.1). Specifically, WFQ creates a fixed number of queues, assigns each of them a guaranteed share of the resource capacity (weights) and schedules jobs within a queue in FIFO order – this allows WFQ to satisfy our predictability requirement (R1). Similarly, the number of queues, weights, and how jobs are mapped to each queue are *tunable*, allowing it to offer the desired flexibility (R2).

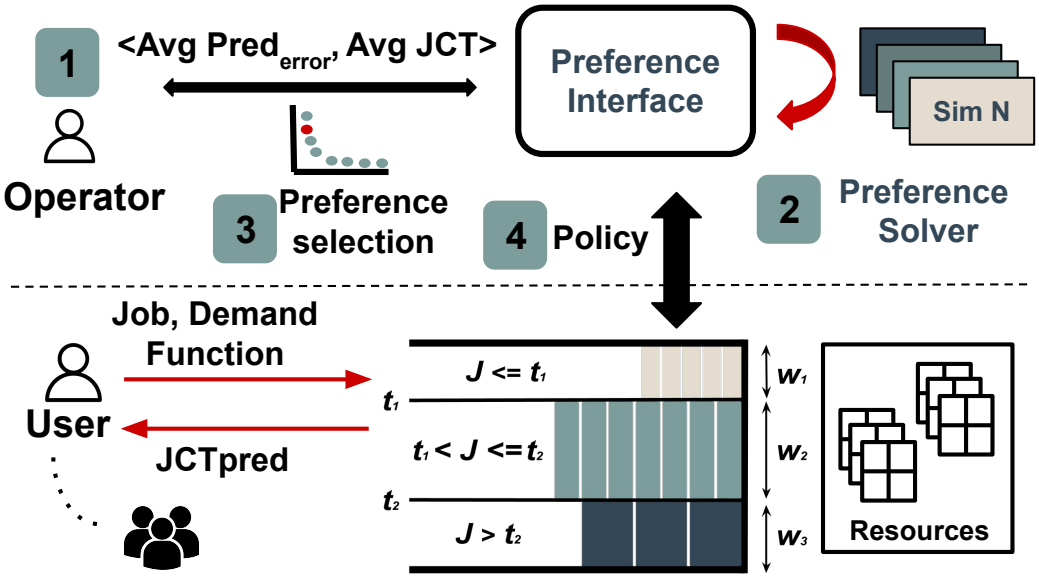


Figure 3.2: Key components of PCS: The preference framework can be used by operators to specify high level objectives. The preference solver uses a simulation-based search strategy to find Pareto-optimal WFQ configurations that are then shared with the operator. On the critical path, users submit their jobs along with the job's demand function and are given a JCTpred.

A key component of PCS that enables the above functionality is the *preference solver* (§3.3.3), which translates the specified high level objectives into a set of Pareto-optimal WFQ configurations using a simulation-based search strategy. The simulation based search strategy is *not* on the critical path of a submitted job; it operates at coarser timescales, aligned with changes in workloads. Since ML workloads are fairly stable, expending the time to search for Pareto-optimal WFQ configurations is feasible. While the space of possible configurations is large, we use an intelligent parameterization of WFQ (e.g., coefficient of variation of job sizes within a queue) to navigate it in a feasible manner. Once a particular WFQ configuration is selected, it can be used to schedule submitted jobs as they arrive.

An important benefit of PCS is that it is a generic scheduling policy – it operates on the notion of a *job* which could be a network flow or a DNN training job etc. To deal with the varying needs of these different scenarios, in PCS, a job is defined using a demand function. The demand function is a mapping between the job's execution time and the resources allocated to it i.e., $demand(n) \mapsto T_{exec}$

and has a minimum ($demand_{min}$) and maximum ($demand_{max}$) resource allocation bound, denoting the execution time under the lowest and highest possible allocation. For ML workloads, in particular DNN training, the demand function is sophisticated, as different models can have different speedups based on the GPU type and affinity and is estimated on the users behalf, as discussed in §3.4. For scenarios like network (co)flow scheduling [66, 74, 52], the demand function is simpler, as we discuss in §3.7.

Finally, the user submits their job, optionally including its demand function. PCS then computes and returns the predicted completion time (JCTpred).

We now explain in detail, our choice of using WFQ as a building block (§3.3.1), followed by preference solver and interface.

3.3.1 WFQ under PCS

We begin by motivating why WFQ is a useful starting point and then share PCS's careful usage of WFQ in meeting our objectives. Our observation is that a lack of preemption, as in FIFO, and a non-zero guaranteed resource share for jobs is crucial for predictability. WFQ uses FIFO scheduling within each queue and across queues the resources are shared according to, strictly positive, queue weights, helping us satisfy the predictability requirement. To highlight the flexibility of WFQ, we show how it can be configured to optimize for extreme points in the trade-off space of maximum predictability, performance and fairness.

- **Maximum Predictability:** WFQ with a single queue is exactly FIFO scheduling which achieves a prediction error of 0
- **Near-optimal Performance:** Shortest Job First (SJF) is near-optimal in minimizing avg JCT for a single bottleneck [186]. WFQ can map each job to its own queue and give a higher weight to queues with smaller jobs, approximating SJF as shown by prior work [198, 52].
- **Max-Min Fairness:** If each job is mapped to its own queue and each queue gets an equal weight, WFQ can emulate Max-Min fair allocation which minimizes unfairness for a single bottleneck [82].

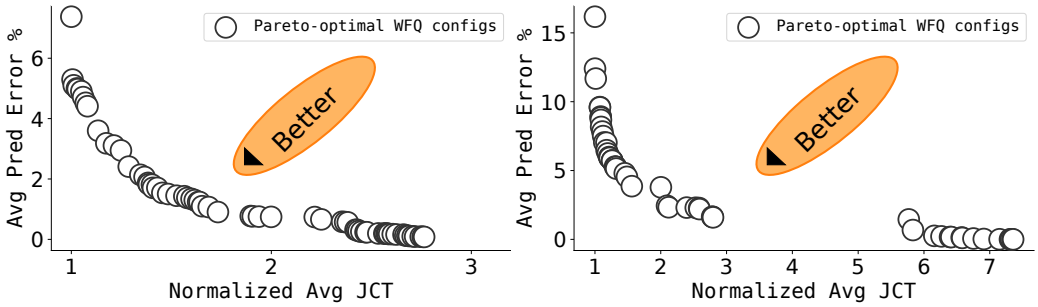
As our analysis in §3.2 reveals, a combination of these objectives is more practical. WFQ offers the necessary baseline flexibility in the queue creation, job mapping and weight assignment strategy. This motivates that we can achieve a combination of these objectives as well, which leads to PCS's preference interface §3.3.2.

Beyond vanilla WFQ. Our core idea is the novel use of WFQ to meet our objectives. First, PCS intelligently chooses the number of queues, weights and the job-to-queue mapping strategy to find various Pareto-optimal configurations, including extreme points, such as FIFO, SJF and Max-Min Fair Share. In PCS, jobs are mapped to different queues based on their size and a set of thresholds ($t_i's$), while strictly positive weights (w_i's) dictate the guaranteed resource share for each queue. For example, jobs with size $> t_k$ and $\leq t_{k+1}$ will be mapped to the k^{th} queue.

Second, within a queue, PCS deviates slightly from a strict FIFO schedule in favor of improving performance and fairness. In PCS, a job's demand function is used to cap the resources allocated to it. For example, a job at the head of its queue may not be assigned all of the guaranteed resource share of its queue (as in strict FIFO); instead, some of the resources may be allocated to the jobs behind it. This allows PCS to handle jobs that exhibits diminishing speedup w.r.t. increase in allocated resources, such as ML training jobs (§3.4).

Finally, to ensure work-conservation, any residual allocation is then redistributed first within a queue in FIFO order by incrementally relaxing the cap on each job's demand function and then across queues proportional to their weights. We expose the weights, thresholds and the demand capping criteria to the preference solver which searches over the space of possible choices of these parameters in order to discover Pareto-optimal configurations (§3.3.3).

Pred$_{err}$ in PCS. Since PCS is work-conserving, a job may get a higher resource share compared to its guaranteed share. For example, if a job arrives when no other job is present, it will be allocated all the available resources (up to $demand_{max}$). This can lead to prediction errors (Pred$_{err}$) if in the future, other jobs arrive and



(a) Trace #1 (b) Trace #6

Figure 3.3: Pareto front of the trade-off between Pred_{err} and normalized average JCT for workload-2 (§3.5). *Better* indicates WFQ configurations that achieve a tight bound on average/tail Pred_{err} while incurring the smallest possible increase in average JCT.

occupy different queues.

In PCS, we bound these errors in a few ways. Firstly, a job's worst-case completion time is strictly bounded, irrespective of the number of future arrivals in other queues or its own queue. This is possible because each queue is assigned a strictly positive weight and uses FIFO scheduling (both properties of WFQ). By bounding the worst-case completion time of a job, the number of preemption events a job will experience during its lifetime is bounded, resulting in bounded Pred_{err}. Second, we exploit the fact that cloud systems are typically highly loaded [89], and by limiting the queues created we can reduce the likelihood of sudden and drastic changes in queue occupancy due to future arrivals. Furthermore, the exact load of a queue is controlled by the thresholds and weight assignment strategy. These observations guide us in discovering Pareto-optimal WFQ configurations.

3.3.2 Preference Interface

PCS exposes a simple yet expressive bi-directional interface that allows operators to specify high level objectives and present Pareto-optimal trade-offs (WFQ configurations) to choose from. This is unlike other tunable systems [149, 139, 121] which assume operators are aware of the trade-offs involved i.e., PCS *actively* tries to help the operator in making an informed choice. Our decision to use Pareto-optimal

choices, as a way to support informed decision making, is grounded in fundamental literature on multi-objective decision making [83, 154], which maps neatly to the problem PCS is trying to address: predictability while being practical.

The preference interface itself, is general enough to be used in scenarios beyond predictability as well. For example it can be used to strike a balance between fairness and performance (e.g., Carbyne [86]) and between minimizing average and tail JCTs [86, 74, 66, 148]. The preference interface exposes the following API:

```
void SetPreference(
    Obj1 <Metric, Measure>,
    ...,
    ObjN <Metric, Measure>
);
List<WFQConfig> UpdateParetoFront();
void SetWFQConfig(WFQConfig config);
```

The current PCS API supports three `Metrics`: Performance (JCT), Fairness ($unfairness$), and Predictability (Pred_{err}). The `SetPreference()` method is used to specify the list of objectives; repeated entries are allowed to support exploring trade-offs across *different* measures of the *same* metric. For each objective, `avg(.)` or a particular `percentile(p)` needs to be specified as a `Measure`.

We envision the following API usage life cycle from an operators perspective: (1) Upon cluster deployment or drastic workload changes, the operator uses the `UpdateParetoFront()` method to kick-start the preference solver (§3.3.3). (2) The preference solver uses the updated workload information and preferences to discover the set of Pareto-optimal WFQ configurations. (3) Once complete, the operator can choose a specific WFQ configuration (`WFQConfig`) to be used by invoking the `SetWFQConfig()` method.

`UpdateParetoFront()` requires PCS to passively collect job size information

and maintain a workload history. When bootstrapping, PCS starts with a default WFQ configuration, which can be any one of the extreme points in the trade-off space (e.g., FIFO) described in §3.3.1. When sufficient workload information is gathered, the preference solver is initiated.

We now show how the API is used to target scenarios covered in our evaluation.

Average JCTs *vs.* Average Pred$_{err}$: Minimizing average JCTs is a popular performance objective and has been a focus of several scheduling policies [89, 165]. To explore the trade-off between performance and predictability, one can specify it as `SetPreference(<JCT, avg>, <Pred`$_{err}$`, avg>)`. We use this for evaluating PCS for workload-1 and workload-2 in §3.5.

Average JCTs *vs.* Tail Pred$_{err}$: Prediction error can be tightly bound by specifying the tail Pred$_{err}$ (e.g., p99) as a measure of predictability. In such a case, the objectives would stay the same as in the above example, however, the measure for Pred$_{err}$ would change from `avg(.)` to `percentile(99)`. PCS uses this specification for workload-3 where low p99 Pred$_{err}$ is challenging to achieve with other policies.

Pareto Fronts. Figure 3.3 shows the set of Pareto-optimal WFQ configurations generated by PCS for two realistic DNN training workloads.

3.3.3 Preference Solver

The preference solver is responsible for finding Pareto-optimal WFQ configurations for the objectives specified. It uses a multi-objective search algorithm to navigate the space of possible WFQ configurations. The optimization parameters consist of the (1) number of queues, (2) queue weights, (3) queue thresholds, and (3) resource allocation cap. These parameters are deemed relevant as they directly control the different trade-offs involved between the objectives considered by PCS. For example, the number of queues influence the degree of preemption and hence predictability,

while the resource allocation cap influences the overall efficiency and hence performance. Other common scheduling dimensions, such as explicit priorities or deadlines, are not considered as they relate to objectives beyond performance, fairness and predictability. For example, some systems may want to prioritize a longer running job. This conflicts with the goal of minimizing JCT; which is rather achieved by assigning a low priority to such jobs. Catering to such scenarios is beyond the scope of PCS.

Finding Pareto-optimal configurations is challenging due to the combinatorial nature of the configuration space. The solver intelligently parameterizes each configuration to make the search process feasible. It uses a simulation-based approach to evaluate the performance, predictability and fairness of a particular WFQ configuration. These are fed to the search algorithm, which decides the configurations to keep, try out next, and discard.

Intelligent Parameterization. To reduce the number of optimization parameters we use the following heuristics:

- **Creating Queues and Thresholds:** Large variation in job-sizes within a queue can lead to HOL blocking but creating too many queues increases preemption events and deteriorates predictability. In PCS, queues are created based on the squared coefficient of variation (C^2) in the job-sizes, inspired by the approach we take in 2D (Chapter 2). We use a tunable parameter $0 < \mathcal{T} < C^2_{max}$ to ensure that queues are created such that C^2 of job-sizes within each queue is $\leq \mathcal{T}$, where C^2_{max} is the C^2 of the entire job size distribution. A larger (smaller) $\mathcal{T}$ results in fewer (more) queues created.

- **Systematic Weight selection:** Higher weights given to queues with smaller jobs improves performance for most workloads. On the other hand, a balanced weight assignment strategy may improve fairness instead. Based on this, we constrain the weight for the i^{th} queue to be $w_i = e^{-i \times \mathcal{W}}$. $\mathcal{W}$ is a tunable parameter which controls the relative weights for each queue. A higher (lower) value of $\mathcal{W}$ leads to a greater (lesser) disparity in weights among the queues. For heterogeneous

deployments, containing several resource types (e.g., k different GPU types) we can use $\mathcal{W}_1 \ldots \mathcal{W}_k$.

- **Finding Demand caps:** The resource efficiency of a job is used to decide its allocation cap and it is computed as $\zeta(n) = \frac{demand_{min}}{n \times demand(n)}$, where $demand_{min}$ is a job's execution time under its minimum possible allocation (e.g., 1GPU) and it is a non-increasing function of the allocated resources. For linear scaling jobs, $\zeta = 1$, while for jobs that scale sublinearly, $0 < \zeta \leq 1$. Instead of a fine-grained efficiency comparison between all jobs, we introduce a tunable threshold ζ_{min} to be used for all jobs, to reduce the search parameters. Using this, a job's resource allocation is capped at k such that $\zeta(k) \geq \zeta_{min}$. Intuitively, a low (high) ζ_{min} means the scheduler is more (less) tolerant towards inefficient jobs. Our evaluation in §3.5 shows that this heuristic is competitive compared to the approach taken by other efficiency based schedulers (e.g., AFS [106], Themis [136]).

Using these heuristics, WFQ$(\mathcal{T}, \mathcal{W}, \zeta_{min})$ becomes the succinct parameterization of each configuration. Different values for these parameters result in different trade-offs between the objectives specified by the operator. For example, setting $(\mathcal{T} = C^2_{max}, \zeta_{min} = 0)$ achieves maximum predictability (i.e., strict FIFO) as only one queue is created and no allocation cap is enforced.

Simulation-based Search. We use a simulation based approach to discover Pareto-optimal WFQ configurations. Our methodology utilizes a simulator, which accepts a WFQ configuration (denoted by $(\mathcal{T}, \mathcal{W}, \zeta_{min})$) as input. The simulator evaluates the provided configuration under a random sample ($\approx$ 1000 job arrivals) of the collected workload (i.e., size distribution and average arrival rate) and outputs the resulting JCT, FFT and JCTpred metrics. The results are then fed to the search algorithm.

The search algorithm samples the search space of possible WFQ configurations and interacts with the simulator to converge to Pareto-optimal solutions. We use SPEA2 as our choice of the search algorithm. It is based on evolutionary search and supports optimizing over multiple objectives [228]. Other multi-objective opti-

mization algorithms can also be used as an alternate, in a plug-and-play fashion. To improve the robustness of each discovered WFQ configuration, it undergoes multiple evaluations under different random samples of the workload to increase its likelihood of being Pareto-optimal.

While we don't have any theoretical basis for the convergence and optimality properties of our approach, it works well in practice and can timely ($\approx$ 1hr) discover the Pareto front for a reasonably sized GPU cluster. Our evaluation confirms that Pareto-optimal configurations found using simulations follow the same trade-offs on the testbed experiment (§3.5.2). We micro-benchmark the feasibility of the simulation-based search strategy in §3.5.4.

3.4 PCS for GPU Scheduling

We now describe the realization of PCS for DNN scheduling on GPU clusters, highlighting important differences and how our abstraction of a job and demand function handles these differences.

DNN Jobs. A job is either a single DNN training job or a collection of DNN trials being run as part of a hyper-parameter tuning task (i.e., AutoML). The demand function for such workloads can be complicated. Modern DNNs require distributed training (e.g., data parallelism) on multiple GPUs. They are known to have sublinear speedup w.r.t to the (1) number, (2) type and (3) locality of GPUs allocated to them [106, 170, 136, 149]. For example, the VGG16 model [183] can be up to 2$\times$ slower when trained on GPUs placed on different servers [89]. PCS relies on existing techniques, such as throughput modelling and profiling, to estimate a job's demand function. As described in §3.3, the demand function describes how the job's execution time changes with different resource allocations. Since allocations have three dimensions: locality, GPU type and number of allocated GPUs, the demand function takes as input different combinations and returns the corresponding execution time. This is akin to the notion of bids in Themis [136] and throughput

in Gavel [149].

Role of Demand Functions. PCS uses $demand_{min}$ as a job's size to map it to its respective queue. The demand function is also used to cap the maximum GPU allocation for DNNs that exhibit sub-linear speedup. Allocating GPUs up to the maximum demand ($demand_{max}$) for such jobs can result in poor performance. We evaluate this approach and show that it works for DNN workloads consisting of jobs that scale sub-linearly (§3.5.3). As described in §3.3.3, the allocation cap is a tunable parameter for the preference solver and can be adjusted for different trade-offs and workloads.

Implementation. We implement PCS as a central coordinator in Python and use the Ray cluster manager [144] for GPU allocation enforcement as well as for general cluster management tasks such as fault tolerance. Each job is either a single trial or consists of multiple trials as part of a hyperparameter tuning algorithm provided by RayTune [131]. We use a custom `ray_trial_executor` to control starting, stopping and preempting individual trials based on the allocations computed by PCS. To determine the remaining service requirements of running jobs, we use various callbacks (e.g., `on_step_start`) exposed by RayTune to get the exact number of iterations completed by each job and multiply it with the profiled time per iteration.

In addition to the central coordinator, PCS consists of an agent, which uses information about running jobs to provide a prediction interface. This interface returns a JCTpred in real time to the user whenever they submit their jobs. The agent computes JCTpred by "virtually" playing out (i.e., in a simulated setting) the current snapshot of the cluster state (e.g., running jobs, available GPUs etc.), accounting for preemption overhead and demand functions of other jobs, to determine the time at which the job will end. This approach is inspired by prior work [75, 185], which use a simulator to compute a job's duration under different resource allocation strategies.

3.5 Evaluation

We evaluate PCS on a 16 GPU cluster with a realistic AutoML style workload to validate our observations. We also cover additional workloads at a larger scale using an event-based simulator. Our evaluation covers different application workloads (e.g., heavy-tailed vs. light-tailed, AutoML apps vs single DNNs), different scheduling schemes (e.g., Tiresias [89], Themis [136]) and different metrics (e.g, avg, p99).

Our evaluation attempts to answer the following key questions:

- **How does PCS perform in terms of Pred_{err} compared to other schemes?** Our testbed results reveal that PCS configurations achieve significantly lower Pred_{err} (20%) while being within 10% of high performing schemes on the performance side.
- **Does PCS work well across different workload types?** The flexibility and predictability provided by PCS holds across different workloads and across preference specifications. PCS can discover configurations that bound the tail Pred_{err} to be within 100% compared to AFS [106] and Tiresias [89] which suffer from $\geq 300\%$ error at the tail.
- **Are PCS configurations fair?** PCS configurations that are optimized for the performance vs Pred_{err} trade-off do not necessarily suffer from unfairness because each queue is guaranteed a GPU share which helps in avoiding starvation.
- **Is the search process feasible?** Our micro-benchmark reveals that the search process can complete within *O(hr)*, making it practical to use, and PCS configurations discovered using the simulation based search-strategy observe the same trade-off *trends* on the testbed.

3.5.1 Experimental Setup

Testbed. Our testbed cluster consists of 16 1-GPU c240g5 machines in the Wisconsin Cloudlab cluster [3]. Each machine has one NVIDIA P100 GPU with 12GB GPU memory.

	Testbed (16 GPUs)	Simulations (64 GPUs)	
Workload	Workload-1	Workload-2	Workload-3
Job Type	AutoML	DNN	DNN
DNN/job	1-20	1	1
GPUs/DNN	1	1-52	1-8

Table 3.1: Summary of the settings used to evaluate PCS

Simulation. We use an event-based simulator to cover workloads that contain jobs requiring *O(100)* GPUs on a homogeneous 64 GPU cluster. We have verified the fidelity of our simulator with trace results from Microsoft [114] and our testbed results with the difference being within 5%.

Pareto Search. The Pareto-optimal configurations for our workloads are discovered by the preference solver §3.3.3 running on a cluster of 10 c220g5 machines in the Wisconsin Cloudlab cluster [3]. It is important to note that these configurations are discovered and evaluated on different sampled subsets of the workload i.e., there exists a notion of training set vs testing set.

Workloads. Table 3.1 summarizes the characteristics of our candidate workloads. We now discuss these workloads in detail.

- **Workload-1:** We borrow this workload from Themis [136] (referred in their work as *Workload-1*). For our testbed evaluation we scale down the maximum number of trials per app to 20 and the maximum service time to 2 GPU-hours. The maximum number of GPUs per trial is set to 1. Each trial tunes a different hyper-parameter (learning rate and momentum) of popular vision models from the VGG family [183].

- **Workload-2:** We use traces from 6 virtual clusters from Philly [6] containing the largest number of jobs. In contrast to other workloads, jobs in these traces exhibit sub-linear scaling. We use the scaling data shared by Hwang et al. on Github [1].

- **Workload-3:** This is borrowed from Gavel [149] (referred in their work as *continuous-multiple*). It is a heavy-tailed workload, with a large number of very small jobs and few long running jobs. We run this workload at a job arrival rate of 4 jobs/hr.

A common characteristic of these workloads is that the minimum requirement of any job is 1 GPU i.e., as long as there is at-least one GPU available, a job can start. This also holds true for RayTune apps which we use in our testbed evaluation.

Scheduling Policies. We compare PCS against FIFO and recently proposed GPU scheduling systems (Themis [136], Tiresias [89], AFS [106]). All scheduling policies considered in our evaluation are "work-conserving" and elastic i.e., they redistribute unused GPUs amongst other jobs according to the policy. For example for FIFO if a job only needs k GPUs and n are available, where $n > k$, then $n - k$ are attempted to be allocated to the next-in-line jobs.

We now describe our implementation of Themis, AFS, and Tiresias that we use in our evaluation.

- **Themis** [136]. On every resource change event and lease duration expiry, in-progress jobs report their fair-finish-time and we allocate GPUs to jobs in descending order of the reported number. We do not consider the scenario where jobs could lie and thus do not require the partial allocation mechanism. The lease duration is set to 10 minutes as per the recommendations of the authors.

- **Tiresias** [89]. Since we assume complete knowledge about job sizes, here Tiresias emulates the Shortest-Remaining-Service-First (SRSF) policy — referred to as Tiresias-G in their paper. As such, GPUs are first allocated to jobs with the lowest remaining service times on every resource change event.

- **AFS** [106]. This scheduler tries to minimize avg and tail JCTs while maximizing resource efficiency. On every resource change event we compute each job's allocation using the AFS-L algorithm.

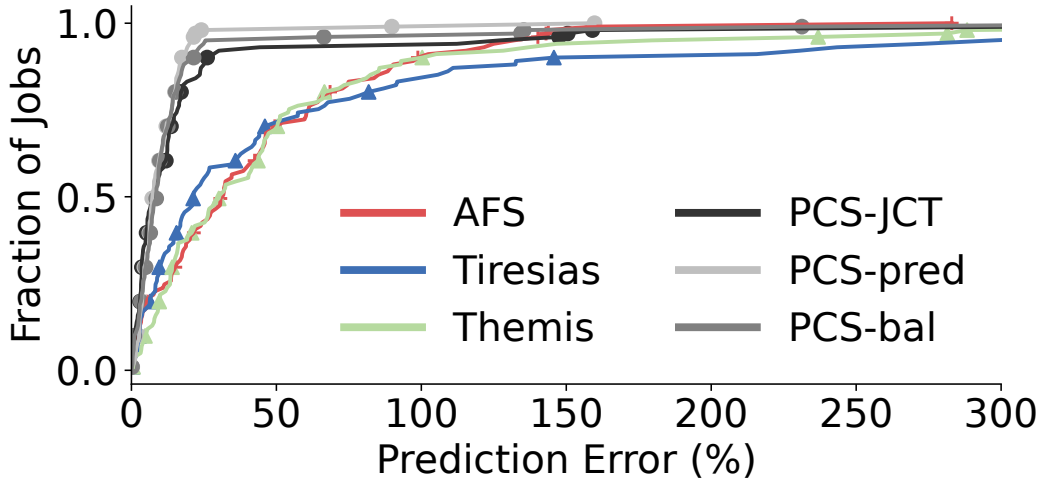


Figure 3.4: [TESTBED] Distribution of Pred_{err} showcasing three configurations of PCS discovered by PCS — performance oriented, predictability oriented and balanced compared to other schemes.

PCS Configurations. We use three configurations for PCS: (1) PCS-pred, (2) PCS-JCT, and (3) PCS-balanced. Each configuration makes a different trade-off. PCS-pred has the highest JCT but the lowest Pred_{err} amongst the three. For each workload and objective the set of WFQ configurations are different and are discovered using the preference specifications described in §3.3.2.

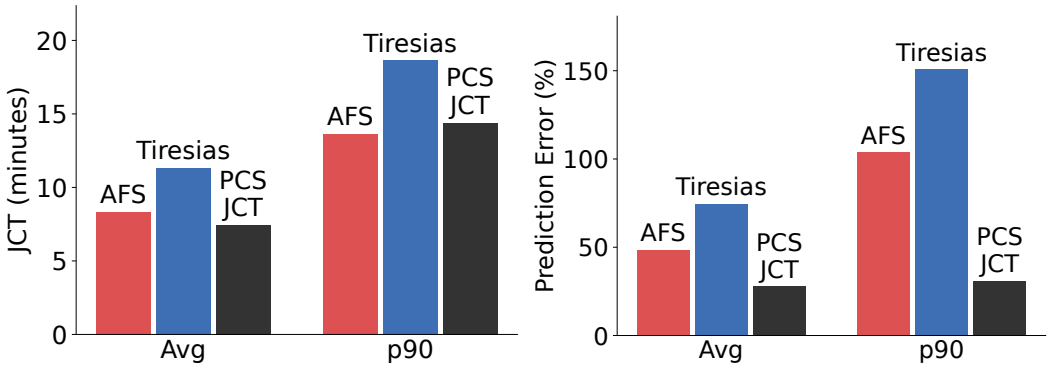
Comparison Criteria. We evaluate the merit of PCS on three fronts:

1. Job Completion Times (JCTs): A commonly used metric to evaluate the performance of scheduling policies.

2. Unpredictability (Pred_{err}): A proxy to capture the error in JCTpred.

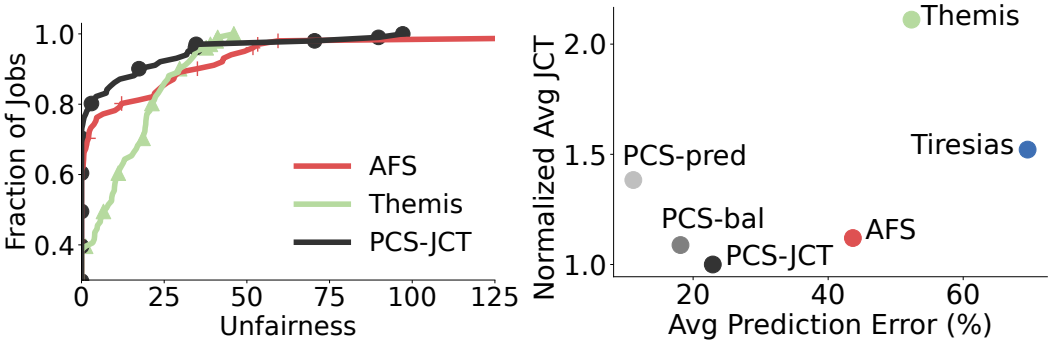
3. Unfairness: It captures the extra time taken by a job to complete, compared to its fair-finish-time (FFT) and is 0 for jobs that complete before their FFT.

We consider all important statistics such as the average and tail (e.g., p99 Pred_{err}, avg JCTs) for all objectives. For each objective, a lower value is better. The testbed result is an average across 3 seeds while simulation results are an average across 5 seeds.



(a) JCTs (b) Pred$_{err}$

Figure 3.5: [TESTBED] Zooming into the trade-off between performance and predictability. PCS is within 1.1× AFS at p90 JCT, with significant improvement to predictability.



(a) Distribution of unfairness (b) Pareto font

Figure 3.6: [TESTBED] (a) shows the CDF of unfairness showcasing that PCS does not significantly compromise on fairness compared to a policy that optimizes for it. (b) highlights the Pareto-optimal configurations discovered in a simulated environment observe the same trend on the testbed evaluation.

3.5.2 Testbed Experiment

For our main experiment we compare three PCS configurations, discovered by the preference solver for workload-1, against other schemes.

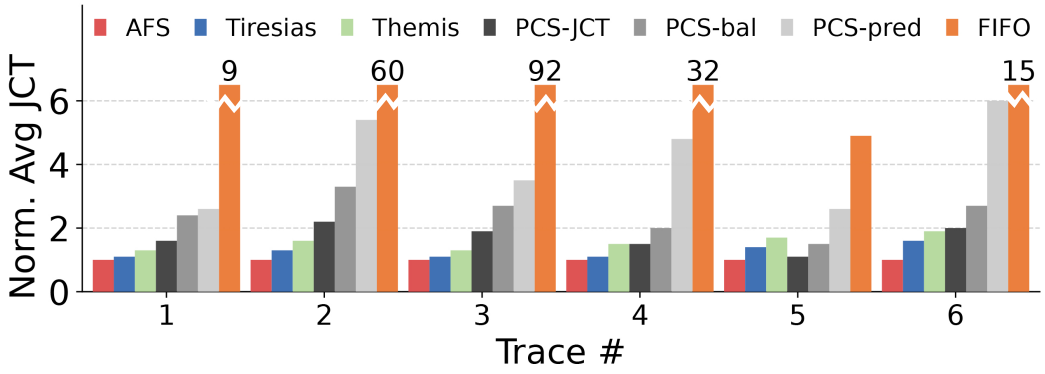
A tight bound on Pred$_{err}$. Figure 3.4 shows the CDFs of Pred$_{err}$ achieved by different scheduling schemes and the three PCS configurations. We observe that all PCS configurations are able to achieve significantly lower Pred$_{err}$. At p90, the difference is an 80% lower error achieved by all configurations compared to other

schemes. At higher percentiles, PCS-pred provides the lowest worst-case Pred_{err} of 150% while other schemes have a long tail. PCS-JCT still has a lower Pred_{err} up until p95.

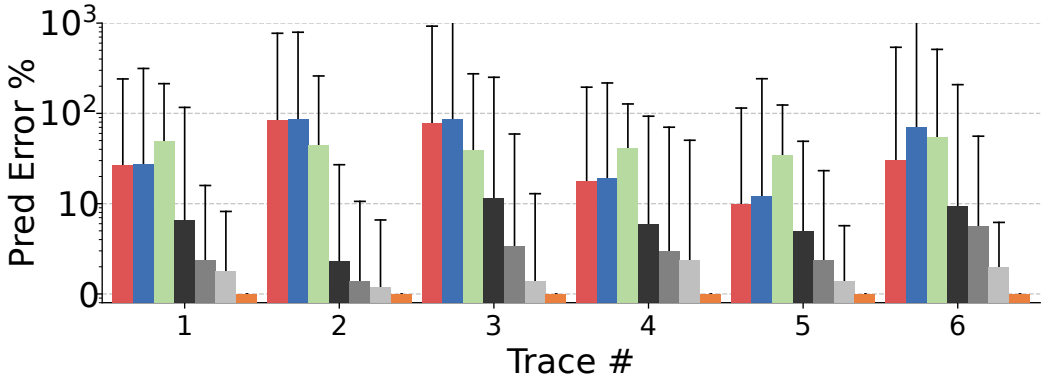
Negligible performance sacrifice for high predictability. Figure 3.5 zooms into the performance versus predictability trade-off achieved by PCS-JCT compared to AFS and Tiresias which aim to minimize JCTs. We see that PCS-JCT achieves equivalent performance to AFS and Tiresias for the average JCTs. It is within 1.1× of AFS at p90, however this trade-off results in significant improvement on the predictability front, where Tiresias and AFS suffer. Pred_{err} under PCS-JCT is within 20% for average and p90 Pred_{err} while AFS and Tiresias have $\geq 40\%(\geq 100\%)$ prediction error at the average (p90). This signifies that PCS-JCT trades off negligible performance to significantly improve predictability. Another source of improvement we observe is that since PCS makes limited use of preemption, overheads associated with preempting running jobs are reduced compared to other schedulers. This is the reason behind PCS outperforming performance oriented schedulers (i.e., AFS and Tiresias).

Unfairness. Figure 3.6a compares the unfairness for PCS-JCT compared to AFS, which optimizes for average JCT, and Themis, which minimizes unfairness. PCS achieves lowest unfairness till p95 and has the worst-case unfairness $\leq 100\%$ compared to AFS which has a worst-case unfairness $> 200\%$. Not surprisingly, Themis offers the tightest bound on the worst-case unfairness of less than 50%.

Pareto-optimality. Finally, figure 3.6b shows different PCS configurations that achieve different trade-off points in the space of avg JCT vs avg Pred_{err}. As expected, PCS-JCT has the lowest avg JCT, while PCS-pred achieves the lowest average Pred_{err}.

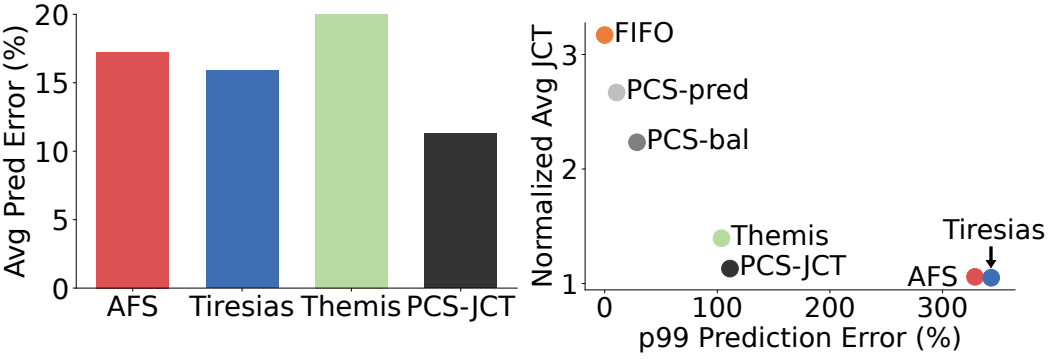


(a) Normalized avg JCT



(b) Predictability

Figure 3.7: [Sim]: PCS for workload-2. a) Most PCS configurations are within 1.5-4× of the performance optimal policies while b) shows that they drastically reduce the average and tail Pred_{err}. In b), the bar height (line) represents average (p99) Pred_{err} and the y-axis follows a logscale.



(a) Prediction Error

(b) Pareto front

Figure 3.8: [Sim]: Showing that schemes that optimize for average JCTs for workload-3 also have a small average error. For such workloads, the tail Pred_{err} becomes an important metric.

3.5.3 Simulation Experiments

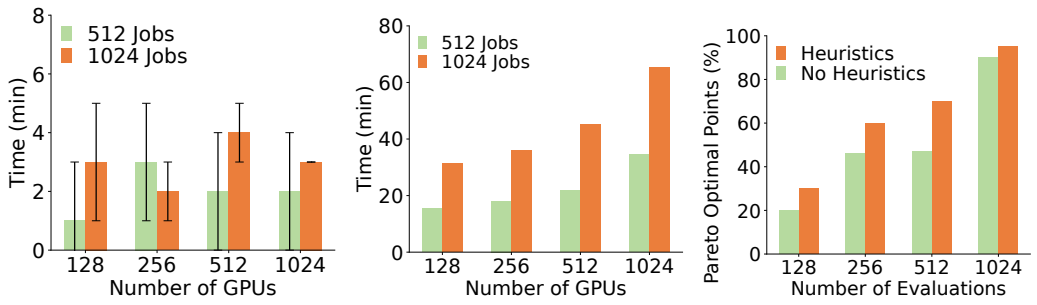
We now consider different workloads at a larger scale in simulations and show the trade-offs achieved by suitable PCS configurations compared to performance and fairness optimal schedulers.

Workload-2. Figure 3.7 compares the performance and predictability of PCS with other schedulers for workload-2. For such workloads, AFS achieves the lowest possible avg JCT by giving more GPUs to jobs with higher efficiency. Despite its conservative approach in dealing with sub-linear scaling jobs, PCS remains within 1.5 to 4× of the optimal scheme for minimizing avg JCT, while drastically reducing the avg and tail Pred_{err}. For example, PCS-JCT reduces the average Pred_{err} from 80% to 1% for Trace #2 and PCS-pred reduces the p99 Pred_{err} from 900% to 10% for Trace #3.

Workload-3. Figure 3.8 compares the different schedulers for workload-3. For this workload, we observe that schedulers optimized for performance, including PCS-JCT achieve reasonably low average Pred_{err}. This is because for workload-3, majority of the jobs are small and similar in size. For such workloads, tail Pred_{err}, becomes important owing to some jobs being starved under priority schedulers. With the appropriate preference specification, PCS discovers configurations that can drastically reduce the p99 Pred_{err}. For example, PCS-JCT reduces the Pred_{err} from $\geq 300\%$ to $\approx 100\%$ while being within 1.1 × of performance optimal schemes (Fig. 3.8b).

3.5.4 Micro-benchmarks

Feasibility of the Search Strategy. Figure 3.9a shows that PCS takes $O(minutes)$ to run a single simulation for a given load (number of jobs) and cluster size (number of GPUs). PCS extensively leverages the underlying parallelism to discover the Pareto-front — requires running $\approx$ 1000 simulations — in approximately 60 minutes (Fig. 3.9b). Figure 3.9c shows that PCS benefits from the heuristics (discussed in §3.3.3) to speed up the search and improve the quality of the Pareto-front by



(a) Single sim time (b) Search time (c) Heuristics

Figure 3.9: Feasibility of the simulation-based search strategy. (a) captures the time to run a single simulation, (b) shows the time it takes to discover the entire Pareto-front. (c) highlights that intelligent parameterization helps in discovering more Pareto optimal points for a given evaluation budget.

(a) Pred_{err} (b) Avg JCT (c) Sensitivity

Figure 3.10: Shows the effects of error in job size and load estimation. a) compares the average Pred_{err} using PCS and FIFO [193] with varying job size estimation error. b) compares the avg JCT of PCS and AFS [106] under the same error. c) shows sensitivity of WFQ configs to load changes.

discovering new points faster than searching on the un-parameterized search space.

Error in Job Size Estimation. Figure 3.10 shows the impact of estimation error in job-sizes on the predictability and performance of PCS. As job-size estimation gets poorer, the impact on avg Pred_{err} follows the same trend as the Pred_{err} under FIFO (Fig. 3.10a). Figure 3.10b, compares the avg JCTs of AFS with no error in job-size estimation to PCS with varying estimation error. PCS is still within 1.05 $\times$ of AFS. This is because as long as the job is mapped to the correct queue, the error in estimating its size has limited impact on performance.

Sensitivity of Pareto-optimal configurations. To evaluate the sensitivity of Pareto-optimal configurations, we evaluate configurations discovered for workload-1 assuming 60% load on a system actually running at 80% load. Figure 3.10c shows that while the exact trends do not hold when the estimated workload is a mismatch, 75% of the configurations are within 10% of the closest Pareto-optimal point.

3.6 Related work

Scheduling Systems. A large body of work emphasizes on intelligent GPU scheduling for DNN workloads, considering metrics such as minimizing average job completion times [217, 197, 123, 89, 106], maximizing fairness [210, 43, 136], cluster efficiency [106, 123, 211] and average DNN accuracy [219, 165]. They use preemption based techniques to achieve their objectives; we show in this paper that preemption is detrimental to predictability.

PCS can benefit from system-level techniques, such as elastic scaling [106, 159], efficient GPU preemption [210, 208, 192, 211], DNN throughput profiling [170, 135], job/AutoML app size estimation [136, 165], and sharing-safety [223] used in these systems. However, in contrast to them, PCS focuses on predictability by limited use of preemption and offering flexibility to cluster operators in choosing various trade-off points between predictability and other traditional objectives. Gavel [149] also translates different scheduling policies to optimization objectives but does not cover predictability and only finds a point solution for each objective while PCS allows operators to choose from a range of Pareto-optimal choices.

Multi-queue Scheduling. A broad category of schedulers use the idea of queue-based scheduling [74, 66, 33, 52, 148, 89, 145, 65] in different contexts to achieve performance related goals. We borrow ideas from these techniques. For example, like 2D [74], we also create queues based job size variation within a queue. Similarly, our limited use of multiplexing is inspired by the FIFO-LM scheduler [66]. However, these techniques opt for a fixed strategy in creating queues, mapping jobs to queues

and assigning weights (e.g., Baraat [66] and Tiresias [89] only use 2 queues) and will be limited to offering a fixed trade-off between objectives.

Adaptive Schedulers. There are multiple recent examples of empirical, adaptive cluster management. For example, SelfTune [121] applies reinforcement learning techniques to automatically update the cluster management policy based on periodic cluster status updates. Decima [140] uses simulations to learn optimal scheduling algorithms for data processing. SWP [224] uses a simulation guided approach to find optimal bandwidth scheduling decisions. These works show the efficacy of using simulated environments to learn system decisions. Our strategy is inspired by them.

Predictable Scheduling. Predictable scheduling and delay guarantees has been studied in broader contexts. Weirman et al [202] classify different scheduling policies based on the variation in the slowdown experienced by jobs. Other studies [105, 61] look at the benefits of providing delay information to users and understand how much delay is tolerable. CFQ [44] defines predictability as a job's FFT, similar to Themis. However, FFT is prone to variation itself as new jobs arrive [117].

3.7 Discussion

Generalizability of PCS. In this paper, we realized PCS for ML workloads, however, it is designed as a generic job scheduling framework and the core insights (e.g., utilizing WFQ to realize multiple trade-off points, bounded preemption to provide predictability, etc) still hold across different scheduling scenarios. We tease apart different aspects of PCS's current realization and discuss their broader applicability. (1) Providing JCTpred. JCTpred can be computed if a job's demand function or simply put, its size is either known or can be estimated. There are several scheduling scenarios, beyond ML, where this requirement holds. Prior work has looked at estimating job sizes for requests in microservice deployments [119, 222], network flows [71, 129], compute tasks in data processing clusters [117, 38, 60] and I/O requests in storage clusters [93, 94]. In some scenarios, like network (co)flow

scheduling, the demand function is simple: $\frac{\text{estimated (co)flow size}}{\text{allocated bandwidth}}$, while in other scenarios it may be more complicated and costly to determine. For example estimating the demand of a request as a function of geo-spatial deployment of microservices [222]. (2) Search process. The current simulation-aided search process is meant to be triggered on coarser timescales, assuming workloads are stable and predictable on shorter timescales. This is true for ML workloads as highlighted in §3.2 but also for some workloads beyond ML [113, 117]. If workload changes are highly dynamic, the search process may not be able to keep-up. This opens up an interesting avenue for future research to tailor the search process for such workloads.

Resource Heterogeneity. To handle resource heterogeneity (e.g., different GPU types), PCS can reuse an existing solution: Gavel [149], which makes a GPU scheduling policy heterogeneity-aware. It supports hierarchical policies with weighted-fairness across entities and FIFO scheduling within an entity. The different parameters of WFQ (e.g., number of queues, weights etc.) map elegantly to these primitives. Once an operator chooses a WFQ configuration, PCS can convert it into an optimization problem that Gavel can solve for.

Sophisticated prediction techniques. Using more complicated prediction techniques is orthogonal work. We posit that future arrivals, the main source of unpredictability, may be difficult to take into account in the prediction decision given that various attributes about them are unknown. For instance, a future job's demand function and its arrival time cannot be determined before it actually arrives. Our emphasis is on making scheduling *predictable* and rely on a simple prediction strategy instead.

Other use-cases of JCTpred. In addition to the use-cases discussed in §3.2, JCTpred can be used to co-design AutoML app schedulers (e.g., Hyperband [128]) with the underlying system; based on the predicted completion time, the app scheduler can decide to prioritize certain DNNs/trials over another. This can be framed as a bi-level optimization problem where the end goal is to find the most promising DNN

hyper-parameters in the quickest time. This will require widening the prediction interface to allow users the option of cancelling and making shadow reservations. Beyond ML workloads, JCTpred can facilitate user applications in i) replica selection strategies (e.g., MittOS [93]), and ii) optimizing the right parallelism and placement for network-bound data processing tasks [146, 75].

Deciding between Pareto-optimal choices. Exposing trade-offs as Pareto-optimal choices can help operators to make informed choices by narrowing down the possibilities. We still, however, rely on the operator's ability to decide between them. One potential strategy is to elicit user preferences via surveys and averaging them to come up with a cluster wide trade-off point. Allowing individual users to pick different preferences on a per job basis, however, can result in cross-user conflicts which may be difficult to resolve. We leave picking preferences on a per-job basis as future work.

3.8 Conclusion

In this paper, we called for providing predictability as a first order consideration in GPU scheduling systems. Our inspiration comes from real-world systems that provide their users with predictions (e.g., estimated delivery dates). Our solution, PCS, provides predictability while balancing other considerations like performance and fairness. It comprises of a bi-directional preference interface to empower cloud operators in making informed trade-offs between multiple objectives. To realize these trade-offs, PCS uses WFQ in unique way coupled with a simulation-based strategy to discover Pareto-optimal WFQ configurations. Our results show the flexibility of PCS in achieving a wide range of operator objectives, offering a first step towards predictable scheduling in a practical way.

Chapter 4

LLMProxy: Intent-Aware Cost-Saving Proxy

4.1 Overview

Today's Internet infrastructure is centered around content retrieval over HTTP, with middleboxes (e.g., HTTP proxies) playing a crucial role in performance, security, and cost-effectiveness. We envision a future where Internet communication will be dominated by "prompts" sent to generative AI models hosted in the cloud. For this, we will need proxies that provide similar functions to HTTP proxies (e.g., caching, routing, compression) while dealing with unique challenges and opportunities of prompt-based communication.

As a first step toward supporting prompt-based communication, we present LLMProxy, an LLM proxy designed for cost-conscious users, such as those in developing regions and education (e.g., students, instructors). LLMProxy supports three intelligent optimizations: model selection (routing prompts to the most suitable model), context management (intelligently reducing the amount of context), and semantic caching (serving prompts using local models and vector databases). These optimizations introduce trade-offs between cost and quality, which applications navigate through a high-level, bidirectional interface.

As case studies, we deploy LLMProxy in two cost-sensitive settings: a WhatsApp-based Q&A service and a university classroom environment. The WhatsApp service has been live for over twelve months, serving 100+ users and handling more than 14.7K requests. In parallel, we exposed LLMProxy to students across three computer science courses over a semester, where it supported diverse LLM-powered applications — such as reasoning agents and chatbots — and handled an average of 500 requests per day.

We report on deployment experiences across both settings and use the collected workloads to benchmark the effectiveness of various intelligent optimizations, analyzing their trade-offs in cost, latency, and response quality.

4.2 Motivation

We are transitioning into a *prompt-centric* world: instead of accessing HTTP content, such as web pages (e.g., `www.CNN.com`), users are interacting with generative AI tools, especially Large Language Models (LLMs), by crafting prompts (e.g., "Tell me about the latest..") to retrieve and access information [11]. This paradigm shift is reshaping how we consume and produce digital content including web search, ecommerce, translation, and more [9, 100], and will have impact on various aspects of the Internet-Cloud ecosystem, including the role of middleboxes.

4.2.1 A Case for LLM Proxies

An HTTP proxy is a type of middlebox that provides numerous features in today's Internet; from security and access control [72] to performance enhancements, such as compression, load balancing and caching [172, 153, 76]. This operates on the assumptions of the content retrieval nature of modern Internet traffic, such as caching the data represented by a URL. Proxies are particularly important for developing regions where bandwidth and energy is costly and low-end devices are the norm [26, 184, 199, 152]. As more and more web features adopt LLMs, we will find ourselves needing similar features (if not more) from proxies that can operate on the

Use Case	HTTP Proxy	LLM Proxy
Content Filtering	Block or allow traffic using domain-based deny/allow lists.	Filtering must consider semantic content of prompts and generated outputs.
Compression	Reduce payloads via precomputing (e.g., Prophecy [152]), or in-network compression (e.g., Flywheel [26]).	Reduce input tokens through truncation/summarization, or reducing examples in few-shot prompting [47]. Fewer tokens reduce cost but impact quality.
Routing	Route requests to replicas or data centers based on load, proximity, or content availability.	Select from multiple LLMs based on latency, load, or price. Smaller/faster models may hurt quality; routing can also combine models.
Caching	Match URLs (and aliases) for reuse; optionally prefetch future content (e.g., RC2 [153], Marauder [172]).	Use semantic caching to store prior answers [35]; augment cached results with lightweight LLMs to respond [116]; support query prefetching [133].

Table 4.1: Comparison of traditional HTTP proxy functionalities and their analogues in LLM proxies, illustrating how semantic complexity introduces new proxy requirements.

new prompt-centric network traffic. While proxies have their own drawbacks (e.g. new failure modes and lack of mobility), there have been efforts in the networking community to address these [68].

In Table 4.1, we summarize a few common HTTP proxy use cases that have analogies to LLMs and what new challenges (and opportunities) the LLM version would face. For example, instead of denylisting URLs to filter content, LLM proxies would need to handle ways users can craft prompts to bypass LLM restrictions. Even if one provider implements content filtering, a proxy may still be necessary to filter content from a collection of model providers and for specific uses (e.g., schools). Bandwidth and latency reduction through compression and prefetching is also common in proxies [26, 152]. This too can be extended to LLMs, by reducing the data needed for prompts. However, there is an additional challenge with applying this to LLMs because modifying the context, including summarization or limiting the number of examples, has an effect on output quality. Lastly, while typical HTTP traffic benefits from proxies that provide routing functions, LLMs can benefit from "model routing". An LLM proxy can route prompts not only based on proximity and load but also quality and cost of available models or even route queries to multiple models.

There are many advantages to placing these features in a proxy at a nearby

cloud (edge) location rather than a local application library. For instance, low-powered IoT devices benefit from avoiding local context storage and from not running optimizations involving additional models. A proxy that is aware of which cloud regions have available LLMs can also account for the sparsity of data centers in developing regions [141] and the limited model support in some locations [22, 20]. All while benefiting from stable network conditions between clouds when deciding the best LLM for a task [95].

Many features that are desirable to have in LLM proxies exhibit trade-offs such as increased quality at the expense of higher latency. We identify three properties that highlight the differences between an LLM proxy and traditional middleboxes:

1. Output quality can *vary* depending on the exact inputs and also which LLM is used (e.g., more or less parameters in the model).

2. Unlike most HTTP requests, the input to LLMs are *flexible* and can be modified to produce similar results.

3. Responses can be generated *iteratively* to refine and improve the result, particularly useful for emerging LLM agents.

These principles motivate the need to consider these proxies as a first-class citizen in the end-to-end communication: they need to be visible to the applications and must work together with the applications to navigate the various trade-offs. This is similar in spirit to the various cases to make hidden middleboxes in today's Internet visible to applications [194, 68, 169]. Prompt-centric proxies may have application specific needs for control and transparency as well, such as strict data governance and auditing that are likely to apply to healthcare applications using LLMs. We believe the need for proxies and applications to work together is even greater for prompt-centric communication and the right time to consider these issue is *now* rather than when these proxies are already widely deployed and used in ad-hoc ways.

4.2.2 Cost Saving LLM Proxy

While the design space for LLM proxies is large, in this paper, we focus specifically on proxy functionally that can reduce the *cost* of accessing LLMs. Cost optimizations have broad applicability, but are particularly useful for cost-conscious users, such as those in developing regions, researchers, students, etc – anyone who cannot afford to always use the highest quality (and highest cost) model available in the market. As prior work in developing regions has shown, users in these areas are particularly cost sensitive and are willing to make various compromises if that results in lower cost [67].

The cost of an LLM typically varies across models and depends on the number of input and output *tokens*, with one word being roughly 1.3 tokens [15]. Generally, output tokens cost more than input tokens (5× difference for Claude 3 models [16]). As new models are released that are cheaper than their predecessors, state of the art models remain costly. For example, GPT-4.5 is 250× as costly as GPT4o-mini [17].

We identify how well known optimization techniques can fit into the design of a proxy, specifically the API, as well as evaluate these common techniques on real world datasets from our WhatsApp deployment 4.5.1 which gives insights into their applicability 4.5. Next, we review three areas of cost-optimization.

Model selection. The most important aspect in determining the cost of an LLM task is typically the choice of model, which affects a number of factors like quality of responses, cost and latency. Some APIs integrate with multiple providers [42], but it is not always clear *how* to select a model. Other frameworks are restricted to only some models - for example, OpenAI's Assistants API [10] does not work with smaller models (e.g., Phi [14]).

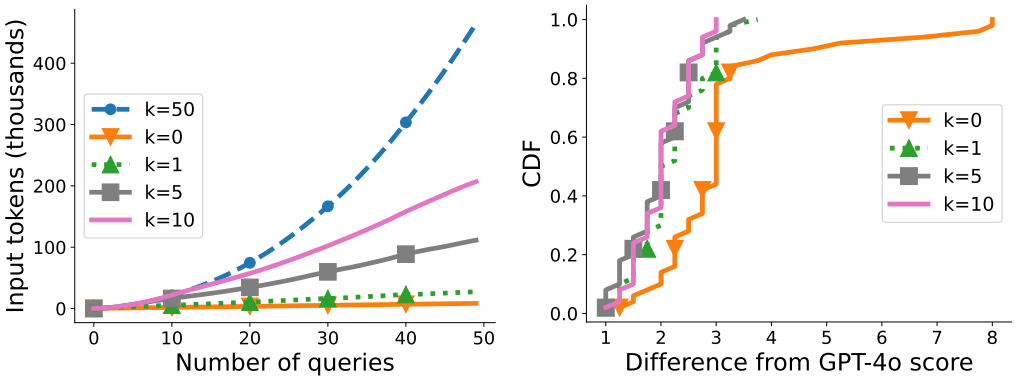
The key observation underlying this optimization is that most expensive models can be an overkill for certain, easier, queries. Therefore, an intelligent strategy for picking an appropriate model, also referred to in recent work as model routing [47, 158, 64], may significantly reduce costs while maintaining the quality of the most expensive model.

Context Management. The amount of context provided to a model affects the number of input tokens, and therefore cost, as well as the latency to process the input tokens. One strategy for chat applications, "last-k", provides the k previous messages as context for the next message. A larger k may increase response quality by including more relevant information, at the expense of higher cost. Additional strategies exist to improve quality without providing all possible context [116, 124, 134], and we explore how LLMProxy can support these in §4.3.4.

To motivate potential cost saving strategies of a proxy, we evaluate cost and response quality using 5 values of k for "last-k" in a 50 query conversation from our WhatsApp deployment (§4.5.1). If we assume all N queries have the same number of input and output tokens, I and O, then the input tokens used with $k = N$ is: $I * N + (I + O)N(N - 1)/2$. This is $O(n^2)$, and Fig 4.1a shows that including all context (k=50) grows quadratically while $k = 0$ grows linearly. The maximum context conversation uses 55× the input tokens of no context and $k = 1$ is only a 3× increase. The quality of these conversations judged against using full context is shown in Fig 4.1b. Each response is given a score (S) from GPT-4o which is averaged over four runs. Although using no context is the lowest quality, for this workload of real WhatsApp queries the difference is most evident only in the tail 20% of messages. This motivates a simple strategy – using context only when necessary to substantially improve quality. We present this strategy in §4.3.4.

Caching. This is a well-known strategy for lowering costs and latency by reducing reliance on an expensive or distant resource. With prompts there is similar potential: a suitable cache can eliminate the need to use an LLM altogether, or cached information can be used to supplement a cheaper LLM to obtain high-quality information at a lower cost.

A traditional HTTP cache is based on exact matches. This, however, can be limiting for prompts, since natural language queries are often *semantically* equivalent despite surface-level differences. For instance, prompts like "Explain how HTTP works" and "Can you describe the HTTP protocol?" may yield identical responses



(a) Input tokens of context strategies

(b) Quality of context strategies

Figure 4.1: 4.1a Compares the cost, measured by input tokens, when various amounts of previous messages (k) are in the the context. 4.1b Compares the quality of each strategy with $k = 50$ as the reference.

and could ideally share the same cache entry. Traditional cache systems that rely on exact matches fail to capture this semantic overlap, underscoring the need for more flexible caching mechanisms.

One way to support this is with embeddings of the input text [116, 84]. Embeddings with a high similarity can be considered cache hits. Since computing embedding requires less resources than generating the response, it may be appropriate to compute locally which enables applications to use the closest cache match when LLMs are unavailable.

An LLM cache can also be populated with high quality data for a cache-local model to construct its responses from. Another well known systems technique, prefetching, is enabled by this caching approach. Requesting additional information related to a relevant topic from a high quality model (e.g., follow-up questions, reasoning-chains [133]) populates the cache, which can then be used by a local model to generate responses for subsequent queries.

4.2.3 Need for a Richer Interface

To leverage these components in order to achieve cost-efficiency, current LLM APIs often require developers to manually configure low-level parameters — such as model selection, context truncation — without clear guidance on how these affect cost,

latency, or quality. Yet in practice, users care about high-level outcomes like "low cost" or "best quality", not the specific configuration knobs. This gap motivates the need for a proxy that offers a more expressive interface.

A suitable LLM proxy should allow applications to *delegate* the responsibility of making such decisions by specifying high-level preferences (e.g., "minimize cost"). In return, it should be *transparent* about how each prompt was handled; by indicating whether a cached response was used, which model was selected, how much context was included etc. Finally, the proxy should support *iterative* refinement, allowing applications to refine or regenerate responses based on this feedback. Together, these capabilities could allow the proxy to capture richer application semantics and enable applications to navigate complex cost-quality trade-offs without micromanaging low-level choices.

4.3 Design

The properties of prompt-centric communication map to the following design requirements that a cost-saving proxy must satisfy:

- **Simplifying navigating the trade-off space.** The *combined* effect of different optimizations on cost and quality are unknown to the applications a priori. The proxy should offer a *high-level* interface to capture application preferences. High level preferences can be mapped to a *myriad* of low-level optimizations (e.g., model and context selection strategies). Barring simple scenarios, it can be challenging to come up with suitable mappings. To address this, the proxy should support *delegation* of responsibility.

- **Providing transparency.** The proxy should inform applications how their prompts were resolved (e.g., whether a cached response was served), particularly in scenarios involving delegation. This is similar to HTTP proxies which insert additional information in the response (e.g., `X-Cache`) to indicate how they processed the request.

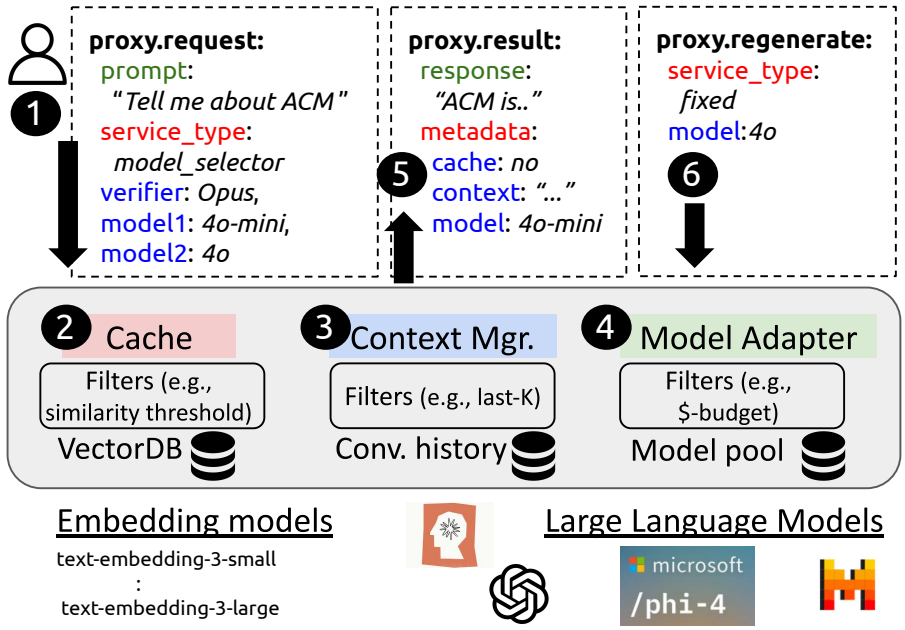


Figure 4.2: Overview of LLMProxy design.

- **Allowing adjustments.** Ultimately, applications are best positioned to assess the quality of the generated responses. In addition to providing transparency, the proxy should allow applications to refine the responses in an *iterative* fashion.

4.3.1 Overview

As shown in Fig 4.2, LLMProxy is a proxy that sits between users or applications (e.g., chatbots) and the various LLMs that are available, including proprietary (e.g., OpenAI, Claude), open-source (e.g., Llama, Mistral, DeepSeek), and custom models. It offers a high-level bi-directional interface (§4.3.2) that captures application preferences (❶), while ensuring transparency (❺) and enabling iterative refinement (❻) of prompt resolution.

Under the hood, LLMProxy consists of three building blocks: the Model Adapter (§4.3.3), Context Manager (§4.3.4), and Cache (§4.3.5), which enable various cost-saving optimizations (§4.2.2) through a filter-based abstraction. The filters provide low-level control (e.g., setting thresholds) but, more importantly, facilitate *delegation* of responsibility by incorporating low-cost models.

The Model Adapter can *route* prompts to different LLMs and *combine* multiple models, based on their respective abilities (e.g., supported context window) and usage cost. The Context Manager retrieves relevant context (e.g., previous message) to supplement prompts, and supports a number of ways to select context including using a low-cost model to reduce the amount of context sent to expensive LLMs. Lastly, the cache stores data (e.g., previous prompts, high-quality information) that could help in replying to new prompts - using a low-cost model to turn cached data into suitable replies.

LLMProxy uses the `service_type` accompanied with each request to decide the order in which to call these components. Different `service_types` can map to different orderings. For all the `service_types` included in this paper, the order ❷–❹ as shown in Fig 4.2 is followed. We now describe the API and the individual components in more detail.

4.3.2 API

`request(prompt, service_type, {key:value..})`	Sends a request to LLMProxy. The result contains the response as well as metadata.
`regenerate(service_type, {key:value..})`	Regenerates the response using either a different `service_type` or the same one.

Table 4.2: LLMProxy API

The API for LLMProxy is shown in Table 4.2. It allows applications to specify a `service_type` as part of the request (`proxy.request`) which maps to a particular configuration of each internal component. The API is designed to work iteratively, using a bi-directional interface where the proxy responds (`proxy.result`) with details of the settings that were ultimately used and applications can regenerate responses (`proxy.regenerate`) using a different `service_type`.

How does the API enable delegation? The `service_type` supports both low-level and high-level specifications, offering a spectrum of delegation of work — from none to a high degree.

An application can use basic `service_types`, specifying low-level parameters (e.g., model to be used). More importantly, the API exposes more powerful `service_types` which delegate these choices to the proxy, which in turn uses low-cost models within each component to decide (e.g., route prompts to LLMs based on complexity). Some `service_types` require additional parameters, which can be specified as key-value pairs. We start with basic `service_types` and progressively move to higher levels of delegation:

- `fixed`: Uses a fixed configuration for model, context, and cache. Configurations can be specified as: `(model=modelID, cache=skip,..)`
- `quality`: Uses the most expensive model and as much context as the model window allows Only considers the cache if a highly similar cached prompt is present.
- `cost`: Uses the cheapest model with no context
- `model_selector`: Employs the model selection strategy detailed in §4.3.3, initially using a cheaper LLM and falling back to an expensive one if the response quality is below a threshold. It uses 5 previous messages (i.e., prompt-response pairs) as context to avoid low quality responses, while keeping the cost down.
- `smart_context`: Uses a small model to determine whether the context should include the last five messages or none. This is useful for applications aiming to reduce input token costs while accepting a potential quality trade-off due to false positives, as explained in §4.3.4.
- `smart_cache`: Uses small model to determine whether a prompt can be answered using cached information. If there is a cache hit, the small model is used to reply to the prompt given the extra information in the cache. This is described in §4.3.5.

Transparency. While delegating responsibility relieve applications from specifying low-level options (e.g., which LLM to use), they make the prompt-resolution process opaque; applications don't know *how* their prompt was handled. Similar to HTTP proxies which disclose request resolution details (e.g., `AGE` and `X-Cache` for

cached content), LLMProxy's responses include *metadata* to provide transparency. The metadata captures the low-level choices made by each component on behalf of the application, including the model(s) used, the amount of context added, and whether the response was returned from the cache. This can assist applications wanting to refine responses iteratively, as we discuss next.

Iterative nature. LLMProxy makes iterative refinement a first class concept, as a way to provide finer control to applications. This is based on the fact that ultimately applications are best suited to asses the quality of the responses. LLMProxy allows applications to regenerate a response, either using the same `service_type` or a different one, based on the information in the metadata provided. Using the same `service_type` in the regenerate request will nudge the proxy to prioritize quality over cost. For example, for the `smart_context`, regenerating a response entails using more context. For different `service_types`, the implementation of `proxy.regenerate()` can be different.

The iterative nature of prompt resolution is a natural fit for many LLM applications (e.g., ChatGPT) which already include an option in the user interface to regenerate responses or provide other forms of feedback [157, 110]. For example, the WhatsApp Q&A service (§4.5.1) we have built on top of LLMProxy contains a "Get Better Answer" button for every message; when pressed, the application signals the proxy to regenerate the response using a higher cost model.

4.3.3 Model Adapter

The model adapter provides two functions: a unified interface that wraps calls to third party LLMs (which may have different APIs, formats etc.) and a way for applications to delegate the choice of the LLM used.

The model adapter maintains a model pool, containing different LLMs and their attributes such as their IDs, cost-per-token, availability (e.g., different regions) and capabilities (derived from publicly available benchmarks). It exposes a filter based interface to select appropriate models and can *combine* them based on the

provided attributes and the selected `service_type`:

```
Filter([Model], attributes) -> [Model]
```

Applications can specify low-level attributes such as model IDs and cost-per-token (to pick a particular model). Alternatively, they can also choose to *delegate* the choice of the LLM by using the `model_selector` service type, letting LLMProxy find the LLM best suited for the application needs.

There are many concurrent efforts to build model selection strategies [158, 64, 47, 115, 25]. These can be supported as different `service_types`. Our implementation of delegating model selection uses a verification based strategy involving a low cost LLM (M_1), a high cost LLM (M_2) and a verifier LLM. For all queries, M_1 first answers the prompt. Then, the verifier judges the response on a scale of 1-10 using a pre-configured judging prompt. M_2 is consulted for a final answer only if the score generated by the verifier is less than a configurable threshold.

The model adapter picks appropriate selections for these models from the model pool using suitable filters. The heuristic it applies is that the cost-per-token of the verifier should be less than M_1's which should in turn be less than M_2's cost-per-token. Applications can also specify which LLMs they desire for this strategy as key-value pairs as shown in Fig 4.2. In §4.5 we show real world benefits this simple strategy has on our production WhatsApp Q&A dataset.

Finally, if the response is unsatisfactory, applications can invoke `proxy.regenerate()` which will directly route the prompt to the more expensive LLM.

4.3.4 Context Manager

The context manager tracks the history of users' conversations. This is additional input that an LLM may process along with each prompt, therefore increasing cost. Keeping context management in the proxy has two key benefits: first, it allows LLMProxy to optimize exactly what context is used (analogous to data compression in HTTP proxies), and secondly, it aids iterative prompt-resolution — applications don't have to resend context each time they choose to regenerate responses.

Filter	Description
`SmartContext(LLM)`	LLM decides if context is needed; otherwise no context is included.
`[LastK(5), SmartContext]`	Either the last 5 messages or no context.
`[[LastK(4), SmartContext], LastK(1)]`	Either the last 4 messages or just the last message.
`Similar(`θ`)`	Messages with similarity $> \theta$ to the current prompt.
`Summarize(LLM)`	LLM summarizes the context messages into a single message.

Table 4.3: Examples of context API. The second example is evaluated in §4.5.3. In the third example, the second dimension ensures that one context message is always included, even if `SmartContext` deems context unnecessary.

To support several context management strategies, LLMProxy uses a filter API where each filter can narrow down which messages are included in the context:

```
Filter([Message], prompt) -> [Message]
```

A message is defined as a prompt-response pair. Table 4.3 demonstrates different ways to this interface, including combining different sets of filters.

The default behavior is to add all available context that fits in the context window of a model. As Fig. 4.1b shows, a simple well known strategy like last-k can be much more efficient. Applications using this strategy can opt to delegate the choice of k to the context manager via the `smart_context` service type. In this mode, the context manager uses a low-cost model (`context-LLM`) to decide how much context (i.e., value of k), and thus input tokens, are required. We implement this as the SmartContext filter.

A false positive occurs when `context-LLM` wrongly excludes required context, while a false negative occurs when it wrongly includes unnecessary context. The latter increases cost while the former reduces the quality of responses. To reduce false positives and ensure high quality responses we invoke the `context-LLM` at most two times and only consider the prompt to not require context if both LLM calls deem it standalone. This is feasible since `context-LLM` is relatively inexpensive and fast.

The filter based API (Table 4.3) also supports other well known context management strategies. The "Summarize" filter uses the `context-LLM` to reduce a long history of messages into a short summary. The "Similar" context filter returns messages in order of their similarity to the current prompt, as opposed to order of recency. This uses the vector database managed by the cache and is another reason the two components benefit from being part of the same proxy.

The design can be extended to enable richer context management strategies such as using `context-LLM` to derive users' interests, language preferences, location, upcoming events (e.g., meetings) etc.

A final consideration is how the context is updated. Typically, when the context is retrieved it will be updated to include the next message, but this is not always the case. Consider a chat application that has one prompt to reply to a user query and another to determine the user's mood from past messages [98]. The second prompt includes the context, but does not update it. In these cases, the coordinator must retrieve context but not insert any.

4.3.5 Cache

In LLMProxy, we build a caching system based on the primitives offered by a vector database (i.e., semantic search). For a semantic cache to be effective, it is important to support a rich set of operations on the `PUT` and `GET` paths. For example, on the `PUT` path, an important consideration is the keys used to store objects (e.g., prompt vs. response) — unlike traditional caches which typically use a single well-defined key, such as the object's hash.

When applications desire fine-grained control, the cache interface accepts low-level specifications (e.g., similarity thresholds). The interface also allows LLMProxy to delegate responsibility to the cache, both on the `PUT` (e.g., generate appropriate keys) and `GET` (e.g., rewrite cached response) paths. This is similar, in principle, to the delegation based strategies employed by the model adapter (§4.3.3) and context manager (§4.3.4).

PUT operation. The cache needs to store objects which could be an LLM interaction (i.e., prompt-context-response trio) or an externally supplied piece of information (e.g., document). Each object can consist of several cached types (e.g., `Prompt`, `Context` etc.) which can potentially act as keys in the database. This is captured by the following `PUT` interface:

```
PUT(Object, optional=[(CachedType, Key)])
```

Embeddings — vector representations — are created from the keys supplied and stored in a vector database. Generally more meaningful keys will result in more useful embeddings [77]. Providing keys is optional; if they are not specified the delegated `PUT` (described later) is used.

Example. If an application wants to cache an LLM generated response with only the prompt as the key, it can specify this as:

```
PUT('Use data structures like B-trees & Tries',
 [(Prompt, 'How do I speed up my cache?')])
```

A future prompt: "Give me examples of popular data structures?" will likely not match with "How do I speed up my cache?" — the cosine similarity is 0.18[1] — but is likely to match with the response : "Use data structures like B-trees & Tries" (similarity of 0.64) and can be rewritten by a small model to be more suitable for the new prompt. Thus, the application can also use responses, and other cached types, as keys. This can be done as follows:

```
PUT('Use data structures like B-trees & Tries',
 [(Prompt, 'How do I speed up my cache?'), (Response,
 'Use data structures like B-trees & Tries')])
```

Delegated PUT. Supplying fine-grained keys hinges on the application's knowledge of future prompts and are *optional* parameters of the `PUT` interface. The delegated `PUT` mode allows applications to leave it up to the cache to decide the best key generating strategy. This is useful when the application wants to populate the cache with complex objects (e.g., a Wikipedia article). In such settings, creating an

[1]Based on OpenAI's text-embedding-3-large

embedding of the entire object may not be useful. To support this delegation, the cache leverages a low-cost local model to intelligently generate keys based on the nature of the object to be cached.

In the delegate mode, the cache uses a small model (`cache-LLM`) to break down a complex object into smaller chunks and generate meaningful keys for each chunk. In addition to using the chunk itself as the key, extra keys are generated based on: *hypothetical questions* that the chunk can help answer and *key-words* extracted from the chunk. The cache also generates modified versions of the chunk: a summary and list of facts present in the chunk (useful when the workload consists of factual queries as we show in §4.5). Similar ideas have been explored by other proposals in the RAG scenario (e.g., LangChain [42]), motivating the benefits of making them part of our cache.

GET operation. The `GET` interface provides low-level control to applications to retrieve objects based on semantic similarity. This is captured via a filter based API:

```
GET([(Key, [Filter])])->[response]
```

Applications can provide a set of filters based on 1. cached types (e.g., `Prompt`, `Document`), 2. a minimum similarity threshold (s), or 3. maximum number of items to be returned (k).

For example, a simple look up to return all responses for which the prompt-to-prompt similarity is above a threshold (e.g., 0.9) can be specified as:

```
GET(['How do I speed up my cache?',[(Prompt,s=0.9)]])
```

Delegated GET. Applications can also delegate this responsibility to the cache by specifying an LLM based filter — we call this strategy "SmartCache". SmartCache internally retrieves top-k items across all cached types and determines whether the retrieved objects are relevant/appropriate (similar to SmartContext §4.3.4). It then uses the retrieved objects to generate a suitable response. The response could be 1. the cached object as-is, 2. a rewritten response or 3. one generated using the user's prompt, context and the cached information.

4.4 Implementation

LLMProxy has been in production for over a year. It is implemented as a Python application running in WAS Lambda functions [12]. It offers access to 10+ LLMs, including OpenAI [10], Anthropic [16], Llama [21], and Phi [24] models. The necessary state such as the conversation history is stored in key-value stores (DynamoDB) and a SQL table (RDS) with vector-search is used to support the cache.

Each incoming request is first converted into an embedding (we use OpenAI embeddings [13]) and looked up in the cache (§4.3.5) for a response. If the cached response is not available or used, the context manager (§4.3.4) retrieves past messages from the conversation history. Finally, the model adapter (§4.3.3) is queried to use provider-specific APIs to generate an LLM response.

LLMs can vary widely in time to fully generate a response, particularly when we are combining models. To ensure requests are processed in the expected order we use a per-user FIFO queue (AWS SQS). Every incoming request goes through this queue, and is only removed from the queue when a response has been sent.

Using a serverless architecture has made it convenient to set up a development and production environment — a useful enabler for incrementally adding features. Production is a stable copy of the various functions and continues serving user requests. Another key benefit is reduced cost; since the functions themselves are light-weight (in compute and memory requirements), they are amenable to the serverless architecture. To reduce the impact of cold starts, which can be >1s according to our measurements, all features of the proxy are in one serverless function.

4.5 Evaluation

To understand LLMProxy's generalizability and effectiveness across real-world workloads, we evaluate its use in a range of settings that vary in user expertise, latency sensitivity, and cost constraints. We begin by highlighting the diverse range of applications built using LLMProxy.

We then present a detailed case study of our WhatsApp-based Q&A ser-

Prototypes & Projects	Features
TWIPS: LLM powered texting app for Autistic users [98]	Determine user tone (simple) Improve user messages (complex)
Morshid: LLM powered educational app	Similar questions asked by users
LLM-based HTTP proxies: LLMs for improving webpage accessibility	Large HTML pages included as context

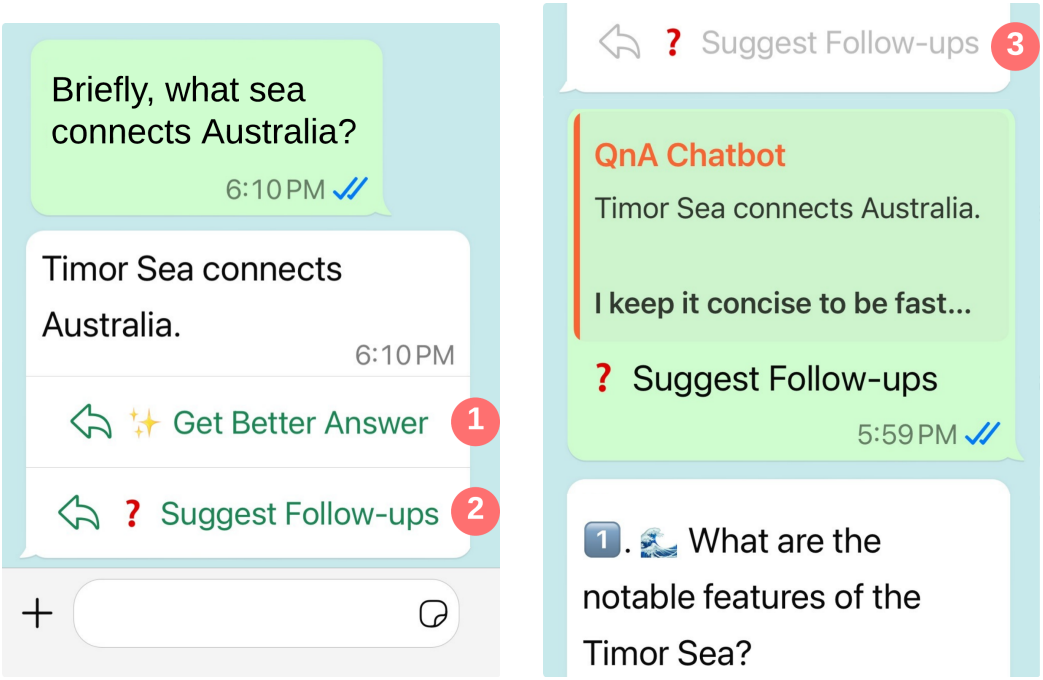
Table 4.4: Use cases of LLMProxy that supports features required by various applications

vice (§4.5.1), showcasing the behavior and impact in a real-world, cost- and latency-sensitive deployment. Next, we describe its use in academic course projects (§4.5.2), which offers insight into LLMProxy's accessibility, flexibility, and usability among student developers. Finally, we present microbenchmarks of LLMProxy's core components (§4.5.3), quantifying the cost, latency, and quality trade-offs enabled by its design.

Supporting different applications. LLMProxy has been used to build a variety of applications, including research prototypes, academic course projects, and a WhatsApp-based Q&A service that we developed. Research prototypes include AI-powered assistive technologies (e.g., detecting user tone or rephrasing messages) and educational tools. Academic course projects have focused on AI powered web accessibility enhancement, multi-agent systems, and chatbots addressing social good applications. Across these use cases, LLMProxy's unified interface and built-in support for context management, caching, and model selection enabled rapid prototyping and experimentation, even by novice users. Table 4.4 summarizes some of these applications and highlights some of their specific features that benefit from LLMProxy.

A unifying theme across these settings is cost-sensitivity: research prototypes and course projects were developed by students operating under budget constraints. Similarly, the user base of the Q&A service, described in the next section, *primarily*

comprises individuals from developing regions (e.g., Pakistan, Sudan) and members of the diaspora in the United States — all of whom are cost-conscious.



(a) User query and response.

(b) Response to button 3.

Figure 4.3: The WhatsApp Q&A service. Buttons 1–3 have pre-fetched (and cached) responses, which are returned when a user interacts with them to avoid delays and keep the conversation responsive.

4.5.1 Case Study I: WhatsApp Q&A Service

We have built and deployed a WhatsApp based Q&A service using LLMProxy. Our small-scale deployment has been rolled out for over 12 months, during which over 100 users, across different countries (e.g., Pakistan, Sudan, UAE), have subscribed and sent over 14.7K requests (4K free-form messages). We share the service's rich set of features, challenges unique to a WhatsApp based deployment (e.g., message oriented nature) and how the proxy helps to support these features.[2]

The Q&A service provides its users access to the latest LLMs via WhatsApp's familiar interface (Fig. 4.3). Cost considerations are crucial since a sizeable fraction

[2]A separate paper describes this service in detail [73]. Here we focus on the its usage of LLMProxy features.

of our user base is from developing regions where WhatsApp is popular [18]. At a basic level, users type-in and send their queries (topics range from health to politics and sports) to our service and get a response. To provide a good user experience, our service supports a number of features: i) anticipating follow-up queries and pre-fetching (and caching) suitable content to enhance responsiveness; follow-up queries show up at the end of the response as buttons, ii) allowing users to regenerate a response, typically more detailed using a higher quality model, iii) pushing recommended content (e.g., trending questions, recent questions, etc.) to users, iv) giving points to users on asking questions and maintaining a leaderboard with daily and overall rankings.

These features also use the limited but powerful WhatsApp affordances (e.g., buttons), and have pushed-based content (e.g., question of the day, questions asked by others) which nudge users to opt for options that are already cached — 13% of the total interactions consist of users requesting the cached content. These features have led to a notable user engagement level, with 20% of users active for several (>10) days, and at least 300 requests sent to our service per week across users. We next discuss how the deployment used various aspects of LLMProxy.

Model Adapter. Having a unified interface to access different LLMs has offered ease of use. Our Q&A service richly leverages the capabilities of various AI models; within (GPT4o vs GPT4o-mini) and across (OpenAI vs Anthropic) LLM-families. Tasks range from responding to user queries, generating user interests, and identifying queries with broad appeal to generate recommended content for our user-base. These tasks have also required combining models; for example using a cheap LLM (Haiku) to filter out candidates from a large set of queries and judiciously applying an expensive model (GPT4o) to identify those likely to be popular. Additionally, with newer and improved LLMs, we have shifted from using GPT-3.5 to GPT4o-mini without a significant difference in the quality of responses sent to our users. While user queries have largely remained the same in complexity, newer model families have become smarter, narrowing the quality gap between cheaper models and more expensive ones — effectively delivering more intelligence at a lower cost.

Different models exhibit varying latency characteristics. For example, our deployment logs show for larger models (e.g., GPT4o, GPT3.5) the mean (p99.9) latency is $3.8s$ ($78s$) while for smaller ones (e.g., Haiku, GPT4o-mini) it is $1.2s$ ($15s$). These characteristics have motivated us to experiment with "latency-centric" `service_types` as well. For example, the Q&A service uses the fastest (and also cheap) model to generate a short initial response (achieved via a suitable prompt) to a query while pre-fetching a higher quality response asynchronously from a more expensive model. This can be elicited via a "Get Better Answer" button (Fig.4.3a).

Context Management. Having a context management module in the proxy facilitates seamlessly switching between different models, and more importantly across different family of models, *during* a conversation. For our service, the context manager maintains user messages in chronological order and manages a few nuances including the scenario where a user requests a regeneration of their response, in which case the initial response is removed from the context.

By decoupling context from specific models, we have observed a form of in-context learning: responses generated by one model, when passed as context to a different model, can influence that model's behavior. This has both positive and negative implications. On the positive side, lower-quality models often begin to produce improved outputs when their context includes messages generated from higher-quality models. Similarly, models may adopt stylistic elements — such as tone — from the responses of other models. However, these inherited behaviors can also introduce inconsistencies. For instance, when responses generated using a grounded model (e.g., Gemini 2.0 Flash [85]) — which include citations like URLs — are used as context for a model without grounding capabilities, the latter may hallucinate sources in its responses. Future work could explore context management strategies to bridge these differences and ensure a more consistent user experience.

Caching. LLM applications often employ streaming to hide the end-to-end latency of generating a response. However, WhatsApp is message oriented, requiring creative ways to mask latency. Our service aggressively pre-fetches data and uses the cache as a masking strategy. Specifically, the Q&A service anticipates follow-up

queries the user may have. These are generated using an LLM and stored in the cache, and are explicitly suggested as buttons (Fig. 4.3b). The proxy uses an exact match to retrieve them, in case the user presses the buttons. This is in addition to using the cache for semantic matches, which we evaluate later.

4.5.2 Case Study II: LLMs for Classroom Settings

Following our deployment in a production-facing Q&A service, we evaluated LLMProxy in a second cost-sensitive setting: undergraduate and graduate classrooms. A subset of LLMProxy's features was exposed via a RESTful API to approximately 60 students across three computer science courses, where it was used to build a variety of LLM-powered applications — including web accessibility enhancement features, multi-agent reasoning systems, and chatbots for social good. Over the course of 145 days, students issued approximately 75K requests, averaging ≈500 requests per day. Several projects spanned the full semester, demonstrating LLMProxy's ability to support sustained, iterative development and maintain responsiveness under long-running, educational workloads.

Although students were not directly charged for LLM usage, this deployment reflects the practical constraints of instructional settings, where instructors or institutions typically bear infrastructure costs. Compounding this, students came in with widely varying levels of programming and LLM experience, which meant that the system had to be both cost-efficient and approachable for novice users. This combination of cost oversight and user diversity introduced a distinct but equally challenging deployment axis compared to the WhatsApp service. In the face of these design pressures, LLMProxy offered a low barrier to entry, enabling quick iteration, and experimentation while abstracting away provider-specific complexity and cost-related pitfalls.

Usage-based service types. To keep costs predictable, we implemented a usage-based `service_type` for LLMProxy. This allowed us to limit access to a curated set of relatively inexpensive models — GPT4o-mini, Azure Phi-3, Claude Haiku, and Meta LLaMA-3 variants — similar in spirit to domain denylists used in HTTP

proxies. The system also supported usage quotas based on input/output tokens and request counts. Despite these limits, students successfully built rich, domain-specific applications.

Supporting RAG-style workflows. Students leveraged LLMProxy's context management and caching components to implement retrieval-augmented workflows. They uploaded diverse reference materials — including policy documents, FAQs, and course-specific content — which the `cache-LLM` automatically chunked and indexed for semantic retrieval. A key challenge was the structural variability of these documents: policy files benefited from section-based chunking, while FAQs required segmentation around question–answer pairs, and so on. LLMProxy's delegated caching interface handled these differences, allowing students to store rich context and later retrieve semantically relevant chunks using a proximity-based threshold. This ability to reuse cached content across prompts as context, allowed us to keep total LLM inference costs under $10 across all three courses.

Models used. Usage logs showed that 73% of all requests were directed to GPT4o-mini, followed by 13% each for Claude Haiku and Meta LLaMA-3 variants, and 1% for Phi-3. Students who experimented with multiple models typically did so to benchmark response quality for their specific workloads.

This usage pattern has motivated us to introduce a batch-mode interface in the future, where users can submit a batch of prompts to be processed by multiple models simultaneously. Such a feature would lower the development overhead of benchmarking and compositional workflows, while aligning with pedagogical goals, such as teaching students to reason about the importance of prompts, and trade-offs in cost, latency, and output quality across models [32].

Another interesting observation — albeit from a single student's project in the multi-agent reasoning systems course — was a connection between the models used and the nature of the prompts they handled[3]. This student employed three models: Phi-3, GPT4o-mini, and Claude Haiku. A closer inspection of their logs

[3]A chi-squared test over the prompt distribution confirmed a statistically significant association between prompt type and model used ($p < 0.001$).

revealed that prompts sent to Phi-3 were generally more structured, rule-based, and imperative in tone, and often used simple, command-style grammar. In contrast, prompts directed to GPT4o-mini and Claude Haiku tended to be less rigid, incorporating softer constraints and exhibiting a more collaborative or conversational tone. These stylistic and structural differences may reflect the student's perception of each model's strengths, with Phi-3 being favored for precise, deterministic behavior and the others for more nuanced or flexible responses. Future versions of the API could surface such model-specific characteristics as part of higher-level `service_types`.

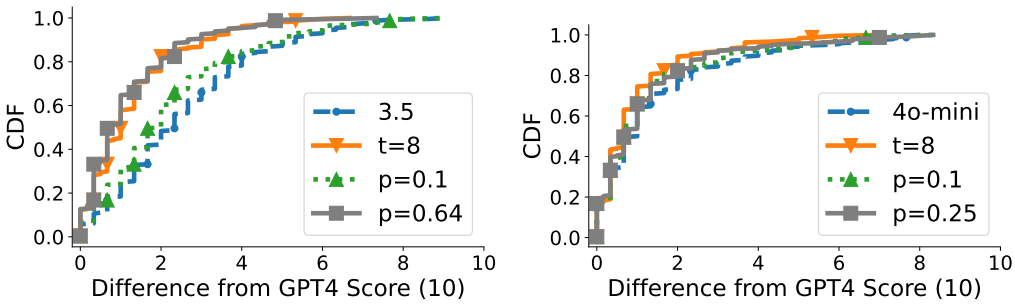
Qualitative feedback reinforced these findings. In post-course surveys[4], 75% of respondents indicated that LLMProxy was easy to get started with, and over half said it integrated smoothly into their project workflows. Students particularly appreciated the simplicity of switching between models and the ability to reuse cached content across queries; features that were especially helpful for minimizing costs and supporting those with limited prior experience building LLM-based applications.

Together, these deployments show that LLMProxy supports effective experimentation and development in resource-constrained settings, from cost-sensitive production services to educational environments with diverse user expertise. We now leverage the real-world workloads gathered from the WhatsApp deployment to benchmark different cost-optimization strategies supported by LLMProxy. We highlight their trade-offs in terms of cost, latency, and output quality.

4.5.3 Microbenchmarks

We present the results of the optimization strategies in each component of LLMProxy. While each strategy is similar to those proposed in related work (§4.6), our evaluation uses a production dataset (D) of 10 conversations selected from one month of our production WhatsApp Q&A service with > 10 messages in each conversation. In total there are 244 queries.

[4]The survey was distributed to a subset of students; we report results based on completed responses.



(a) Original Models

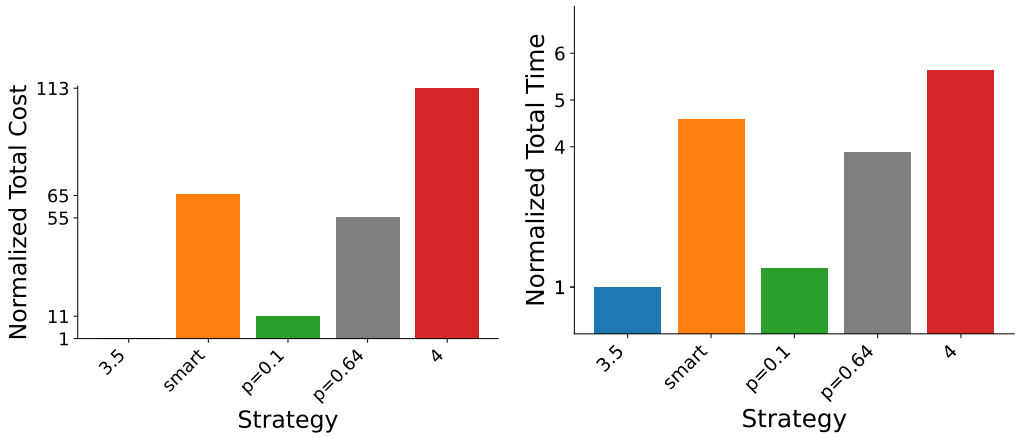
(b) New Models

Figure 4.4: Fig. 4.4a compares the quality of verification with $t = 8$ and random strategies with $p = 0.64$, $p = 0.1$ using an earlier generation of models (GPT 3.5, GPT4, Opus). Fig. 4.4b is the same but with new models (GPT4o-mini, GPT4o).

Model Selection. We evaluate the verification-based model selection strategy discussed in §4.3.3, which shows how to intelligently combine a cheaper and expensive model to save costs with little impact on the quality of responses.

Setup. We evaluate using both models that were available at the time we collected the data as well newer models that are the latest at the time of writing. First, we set the less expensive model (M_1) to GPT3.5, expensive model (M_2) to GPT4, and use Claude Opus as our verifier. The newer models are GPT4o-mini as M_1 and GPT4o as M_2 and the verifier. Both are evaluated on D using the strategy described in §4.3.3. This is compared to only using M_1 to answer all the questions. The response from M_2 is assumed as the reference, and hence always gets a score of 10. We compare our intelligent strategy with a strategy that randomly selects a model — a common practice in optimization (e.g., hyperparameter tuning [99]). This strategy randomly uses M_2 with a probability of p, and otherwise uses M_1. With $t = 8$, M_2 is used to answer $> 60\%$ of the prompts with the original models and 25% of the prompts with new models. Based on this, each experiment shows the random strategy $p = 0.64$ or $p = 0.25$ as well as $p = 0.1$ to demonstrate a lower cost alternative.

Results. As shown in Fig. 4.4a, our verification strategy noticeably outperforms using M_1 all the time and has noticeably more answers within 1 to 3 pts of M_2's answers than M_1. Interestingly, Fig. 4.4b indicates that newer generation of



(a) Cost Comparison

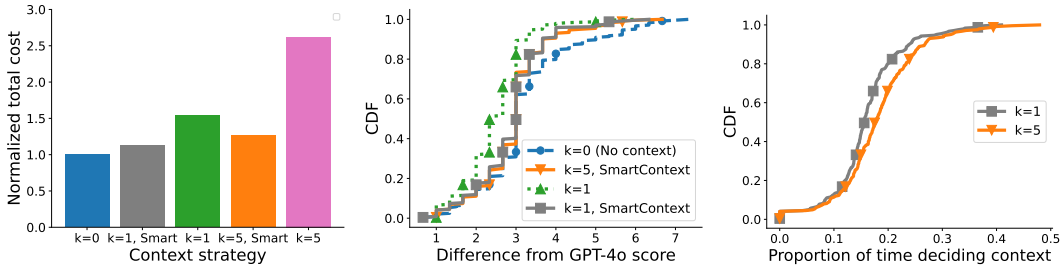
(b) Time Comparison

Figure 4.5: 4.5a compares the cost of answering all prompts using our verification strategy with $t = 8$ and our random strategy with $p = 0.64$. 4.5b compares the total time. Both are normalized to GPT3.5

models are capable of answering the kinds of questions users ask our service even with the cheaper variants (4o-mini). This result is also demonstrated by the proportions of prompts that are routed to each model. When the 4o family of models is used a smaller percentage of prompts are routed to the large model. While the $p = 0.64$ random strategy performs similarly to our model selection, picking the optimal percentage a priori can be challenging ($p = 0.1$ is worse).

In Fig. 4.5 we show the cost and time comparisons of these strategies using older generation models. Fig. 4.5a shows the (normalized) total cost of each corresponding strategy and demonstrates that our model selection has a 40% reduction in cost from using M_2 only. Fig. 4.5b demonstrates that model selection is significantly faster than using M_2 exclusively, although it is about 5× slower than using only M_1. Overall, these results show that our simple model selection can reduce costs while maintaining quality. Consistent with these results, we updated our Q&A service to use GPT4o-mini instead of GPT3.5.

Context Manager. We evaluate the `smart_context` service type, which uses a low cost LLM to decide if context needs to be supplied to a high cost LLM. We see an up to 50% reduction in cost compared to last-k, while limiting the tail of low



(a) Cost (b) Quality (c) Time

Figure 4.6: Results of context experiments. 4.6a shows cost, normalized with the lowest to 1, for each strategy. No context is cheapest, as expected. Smart strategies are ∼30% and ∼50% cheaper for k=1 and k=5, respectively. 4.6b is a CDF of response quality for each strategy. k=0 has the worse quality, as expected. Both smart context strategies are similar in quality, falling between k=0 and k=1. k=5 is the baseline that quality is scored against. 4.6c is a CDF of the proportion of time replying to each prompt that is spent determining if context should be used for the k=1 and k=5 SmartContext strategy.

quality responses resulting from using no context (§4.2.2).

Setup. Queries in D were replayed with following strategies: last-k with k=0 (no context), 1 (most recent message), 5 (baseline), SmartContext with k=1, and SmartContext with k=5. After replaying each conversation we judged the quality of conversations with the LastK(5) conversation used as reference.

For each of the N messages in a conversation, C_i, and the reference conversation, R_i, where $0 \leq i \leq N$, the judge gave a score $0 \leq S_i \leq 10$ based on inputs C_i, C_{i-1}, R_i, R_{i-1}. The results are averaged across three runs.

Results. The results of our experiments for cost, quality, and time are shown in Fig. 4.6. Results show SmartContext combined with $k = 1$ or $k = 5$ can reduce costs by 30-50% while being higher quality than the no-context response. Quality is similar whether $k = 1$ or $k = 5$, suggesting most of the quality difference improvement is present with just one message in the context. SmartContext quality is particularly higher than no-context in the tail 20% of queries, following from our intuition that only some queries require the context. This shows how SmartContext only slightly reduces quality for large benefits in cost-savings. Our results also indicate the total time using the SmartContext strategies is increased by $< 20\%$ for about 80% of messages when k=1, and the largest increase is $< 50\%$.

Cache. We evaluate the `smart_cache` service type, which uses a local small model combined with high quality cached information to generate responses. An issue with small models is their tendency to hallucinate (especially with factual information). For such queries, `smart_cache` is able to improve the worst-case quality by 4×. While similar to RAG systems [124], an interesting insight is the use of widely available information sources to *intelligently* populate the cache.

Setup. The cache is populated with Wikipedia [19] articles on topics gathered from our WhatsApp service usage, using the delegated `PUT`. We select 170 queries across 17 user conversations. These queries represent the last 10 requests per user, at the time of running the experiment, sent to the Q&A service.

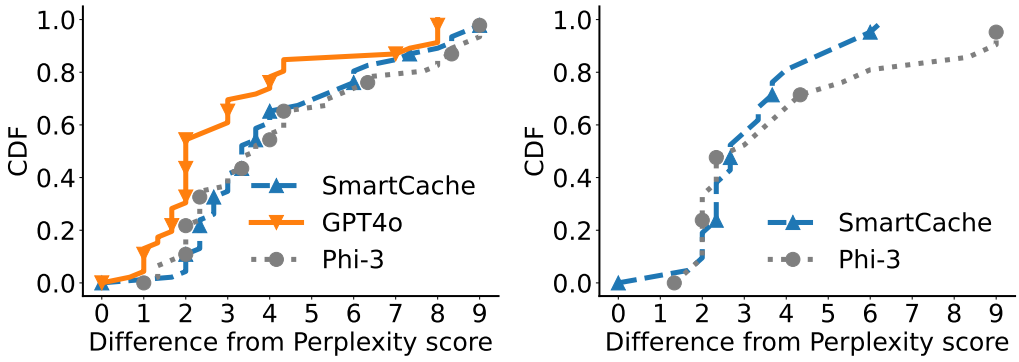
We focus on queries that are factual (using GPT4o to determine this), which consist of 30% of the overall queries (i.e., 51 queries), since this has the largest opportunity in leveraging a cache populated with factual information.

The `smart_cache` uses Phi-3 [14] (3.8B parameter model). We compare our approach against directly (i.e., with cache disabled) using GPT4o and Phi-3 to answer queries.

Conversations are replayed and the response quality is judged with a reference answer generated using Sonar-Huge-Online [23]. Sonar-Huge-Online serves as a strong baseline since it has access to information on the internet and the responses it generates are factually grounded; an important facet of our experiment.

Results. The results of our experiment are shown in Fig. 4.7. We first focus on the overall quality of responses generated by different strategies (Fig. 4.7a). GPT4o is considerably superior compared to using Phi-3 (as expected) for the majority of the factual queries, the worst-case being ∼8pts from the reference. `smart_cache` is able to bridge the quality gap, in particular for 20% of the queries with the lowest quality. While marginal, the improvement using `smart_cache` is the difference between a factually sound (but a less detailed/creative) answer and a hallucinated (those generated by using Phi-3 alone) one.

To emphasize the benefits of leveraging the cached content we narrow down on the subset of queries where `smart_cache` decides to use the cached information



(a) Overall (b) Subset of queries

Figure 4.7: 4.7a shows the quality CDF of the `smart_cache` vs. directly using GPT4o/Phi-3. 4.7b highlights the benefit of using factual information for smaller models which have a propensity to hallucinate.

in Fig. 4.7b. For such queries, `smart_cache` has a considerable advantage over using Phi-3 in isolation — the lowest score achieved on this subset by `smart_cache` is 4pts vs. 1pt when using Phi-3 alone.

4.6 Related Work

Abstractions: Systems such as Parrot [132] and Teola [190] propose a more expressive LLM API that reveals dependencies between requests allowing for application level optimizations rather than request level. Our proxy interfaces with existing LLM APIs and focuses specifically on cost optimizations that do not require modification in the LLM serving infrastructure. Other abstractions such as LangChain [42] provide many building blocks, several of which can benefit LLMProxy such as context summarizing (§4.3.4), to build LLM applications. However, it does not provide a high level API like ours, requiring applications to figure out the appropriate low-level configurations.

Model routing: The problem of selecting the right LLM for a task is an active area of research, with many concurrent works, such as HybridLLM [64], RouteLLM [158], FrugalGPT [47] and using benchmarks [182] all involving a "router" to select the best model for a task. LLMBlender [115], which combines the strengths of multiple LLMs,

and others have studied cascading approaches similar to our strategy in §4.3.3 [221, 91]. These strategies can potentially be made part of LLMProxy's design as well. Our approach is simplistic and meant to highlight the design of LLMProxy while not requiring any additional model training (which can be expensive) and still performing well on our production dataset. Future work could provide a quantitative evaluation of the pros-and-cons of these different approaches and more insights into what kind of workloads a given strategy should be used for.

Context management: Other works lower cost by reducing the number of input tokens through models trained for this purpose [180, 134]. This could work in tandem with our SmartContext strategy as another context filter. While LLMProxy targets QnA style LLM uses, other systems have more complex context management requirements such as generative agents [162]. They treat context as a "memory stream" that surfaces relevant memories for new queries. With some modification we believe our filter based API can also work for this style of context.

Caching: Systems such as GPTCache [35] and MeanCache [84] use embedding models to reply to LLM queries with saved responses. Others have improved on the embedding models for more effective caching [227] and used LLMs to generate test inputs for semantic caches [174]. Our interface for LLMProxy is flexible enough to benefit from these efforts, and can also accommodate our strategy of intelligently populating the cache with high quality factual knowledge and using an inexpensive LLM to respond to user queries (§4.3.5).

Other optimizations: There have been other recent works that optimize aspects of LLM scheduling [215, 181] and caching intermediate computation [116] which can also benefit LLM APIs when they are used by LLMProxy. Benchmarking model quality is also a recent area of research and we use LLM as a judge, inspired by [225].

Proxies: There are many examples of performance optimizing proxies, some of which are primarily meant to reduce cost [69, 151, 26, 152]. Others work at the

transport level to improve performance [41, 88], and others take into account application specific knowledge to improve performance [153, 51]. These use optimizations such as caching and pre-fetching, which, with modification, can be used to improve LLM usage.

4.7 Conclusion

We introduced LLMProxy, a proxy for supporting prompt-centric communication. As a starting point, LLMProxy focuses on cost-optimizations, offering a suitable interface to capture application preferences and internally using model selection, context management, and caching. Our design, implementation, and evaluation highlight the quantitative and qualitative benefits of our approach, in supporting rich services such as the WhatsApp Q&A service, and diverse use-cases as well as providing cost benefits in various scenarios.

Chapter 5

Conclusion

This dissertation advocates a new paradigm for systems design — one that centers user intent and leverages intelligence (e.g., principled form of learning) to adapt low-level decisions. Across domains as varied as network flow scheduling, GPU job orchestration, and LLM-based inference, the systems presented in this work demonstrate that exposing high-level objectives and using intelligent, workload-aware mechanisms can yield infrastructures that are both high performing and adaptable. Rather than optimizing for a fixed target or assuming a known workload, these systems embrace workload uncertainty and diversity, enabling robust behavior. Looking ahead, this vision opens rich opportunities for future work, from rethinking API abstractions for agents, to building end-to-end infrastructures that automatically align with complex application goals.

5.1 Future Work

Outlook. This dissertation advocates a design paradigm for cloud systems where user intent is a first-class concept, and system behavior is guided by learning/intelligence rather than fixed assumptions. The systems presented — spanning network scheduling, GPU orchestration, and prompt serving — demonstrate how exposing high-level objectives and using data-driven adaptation can yield robust, user-aligned infrastructure.

As we move toward a more agentic computing paradigm — where AI agents dynamically plan and act on behalf of users — these principles become even more critical. Systems must not only respond to user goals but also surface predictable behaviors, support dynamic resource adaptation, and close the loop between application-level decision-making and infrastructure-level execution.

Building on these foundations, this dissertation calls for a thrust in three key directions: (1) rethinking system interfaces for agents and humans to expose predictability and control; (2) deepening learning-based adaptation across broader system components and timescales; and (3) developing end-to-end frameworks that automatically align infrastructure behavior with complex, evolving application goals. Together, these directions push systems research toward infrastructure that is not just performant, but anticipatory, personalized, and intent-aware by default.

5.1.1 Interfaces and APIs for Agentic Systems

LLM-powered agents are increasingly being tasked with complex workflows that span multiple layers of modern infrastructure — invoking external APIs, composing cloud services, and reacting to runtime feedback. However, these agents currently operate in a world of unpredictability. Two fundamental gaps prevent them from acting intelligently: (1) external APIs do not expose execution characteristics like latency or cost, and (2) systems lack standardized, goal-driven interfaces that allow agents (or users) to declare their intent and receive actionable feedback.

Building on insights from Chapter 3 and 4, predictability should be a first-class concern in the design of agentic systems, enabling agents to make informed decisions under uncertainty, reason about trade-offs, and iteratively refine their plans based on system feedback.

Predictability-Aware APIs. Today's cloud APIs are built around a narrow request-response abstraction, offering no visibility into how long a request might take, how much it will cost, or whether it will succeed. This opacity is particularly limiting for agents that need to plan multi-step workflows. Without visibility into

performance characteristics, agents are forced into brittle or conservative behavior — retrying failed steps, hardcoding timeouts, or overprovisioning resources.

To address this, APIs should expose predictive feedback such as: (1) Estimated latency or cost distributions; (2) Confidence intervals around success likelihood; (3) Expected queuing delays or resource contention. Armed with such feedback, agents could intelligently select among alternative services (e.g., choosing between fast/expensive and slow/cheap models), reorder execution plans, or speculatively cancel requests that exceed certain thresholds. This requires new API semantics — such as modifiable or transactional calls — and systems support for providing lightweight forecasts, simulating outcomes, or reserving resources without full commitment.

Goal-Aligned Interfaces. Beyond individual API calls, agents need higher-level interfaces that align with their overarching objectives. However, today's systems offer no consistent medium for expressing such goals, nor mechanisms for exploring trade-offs. Each system (e.g., schedulers, caching layers) reinvents its own knobs and abstractions, forcing callers to micromanage behavior in domain-specific ways.

This motivates a broader design: bidirectional, declarative interfaces where agents (or users) express goals such as "maximize predictability under cost constraints", and the system responds with previews of possible outcomes. These previews might include expected latency, cost breakdowns, success likelihoods, or what-if counterfactuals ("what happens if I relax this constraint?"). Such feedback helps agents iterate on their intent, ultimately arriving at plans that are both feasible and aligned with system behavior.

Building these interfaces raises several research questions: (1) What abstractions can express intent across domains like inference, scheduling, and storage?; (2) How can systems efficiently generate previews without full execution?; (3) Can learning-based models (e.g., simulators, predictors) be shared across systems?

5.1.2 Intent Arbitration in Multi-Agent Systems

As we move toward more agentic computing models, where autonomous agents interact with shared infrastructure and collaborate on high-level tasks, a new class of coordination problems arises. Unlike traditional applications, agents often act on behalf of users with personalized goals — and they do so in environments where resources are limited and interactions are interdependent. This creates challenges not only in managing contention for shared resources, but also in ensuring fairness and efficiency during inter-agent delegation. Addressing these challenges requires new arbitration mechanisms that operate at both the system level (e.g., for scheduling or resource control) and the application level (e.g., for workflow planning or task sharing).

Intent Arbitration for Resource Contention. As LLM agents become increasingly common, they will compete for shared infrastructure resources such as compute, memory, and bandwidth. Traditional schedulers lack awareness of high-level agent intents — whether a request is exploratory, deadline-sensitive, and much more — and agents lack visibility into system load or expected delay. To bridge this gap, systems must arbitrate between conflicting intents, aligning agent goals with resource constraints. This opens a new design space for goal-aware resource scheduling that blends techniques from auction theory (e.g., VCG bidding), intent classification, and multi-objective fairness (e.g., dominant resource fairness or weighted max-min). One particularly promising direction is leveraging slack — the flexibility in task urgency or quality thresholds — to enable opportunistic resource redistribution and cooperative scheduling among agents. Key questions include: How should systems quantify and compare intents across agents? How can they ensure incentive-compatible behavior? And how do these policies scale with dynamic workloads and partial observability?

Fair Delegation in Multi-Agent Collaboration. Beyond infrastructure contention, agents will also delegate tasks to one another; a planning agent may rely on a summarizer or translator agent to complete a workflow. Without coordina-

tion, this can lead to asymmetric delegation where certain agents (or users) become overloaded or exploited. Addressing this requires fair delegation mechanisms that account for agent capabilities, workloads, and user ownership.

One promising direction is to model delegation as a task allocation game, incorporating ideas from multi-agent coordination, cooperative game theory, or contract-based systems. Inspired by insights from Chapter 3, a proxy-based architecture is especially well-suited for implementing such arbitration logic: it sits between agents, can observe cross-agent interactions, and is amenable to auditing and accountability. By tracking historical interaction patterns, delegation outcomes, and feedback loops, the proxy can detect persistent imbalances and enforce fairness-aware delegation policies. This opens up new challenges around negotiation protocols, and scalable enforcement of delegation-aware fairness objectives.

5.1.3 Resource-Adaptive Model Scaling

As personalized and fine-tuned AI models become mainstream, scaling them (pre- and post-training) efficiently under variable resource availability is increasingly important. Existing scheduling systems attempt to optimize resource allocation under constraints, but treat the model architecture as static. This creates a fundamental limitation: even the best scheduling policies are bottlenecked by the inflexibility of the job being scheduled. By enabling the model itself to adapt, growing or shrinking dynamically *during* scaling, we can unlock additional optimization opportunities.

In dynamic compute environments, such as shared GPU clusters, spot instances, or edge deployments with fluctuating capacity, fixed-architecture training can be brittle. A promising future direction is *elastic model scaling*, where architectural changes are treated as first-class control primitives that can respond to real-time signals from the system. For example, schedulers might trigger model shrinkage when anticipating preemption, or suggest expansion during low-load periods to improve model accuracy.

For growth, techniques like Net2Net [48] transformations or progressive layer stacking can safely expand model capacity mid-training. For shrinkage, we envision

a distillation-based strategy: the larger model acts as a temporary teacher, seeding a smaller model that inherits knowledge while consuming fewer resources [103]. This smaller model can then resume training, preserving momentum even under tighter compute constraints. Similar ideas can be applied to inference-time scaling.

These transitions enable new scheduling strategies: jobs are no longer fixed points in a resource-time-cost space, but adaptive entities with a range of possible trajectories. This unlocks exciting co-design opportunities between schedulers, and training and inference frameworks, where architectural transitions are jointly optimized with job placement, load balancing, or budget control.

Several systems and algorithmic challenges, however, will need to be addressed: (1) designing lightweight shrinkage heuristics, (2) coordinating distillation timing and transition costs, (3) preserving training state and optimizer dynamics across architecture shifts, and (4) interfacing with scheduling policies to make transition decisions informed and predictable.

5.1.4 End-to-End Goals and Application-Aware Infrastructure

While individual system components, such as schedulers, caches, or model selectors, can be optimized in isolation, real-world applications often care about broader goals that span multiple layers. For example, in the WhatsApp-based LLM service, the end-user's objective is to receive a helpful answer quickly; not merely to minimize latency or token cost in isolation. Achieving this requires coordinated reasoning across subsystems (e.g., routing, caching, generation) and aligning system behavior with emergent goals like responsiveness or engagement.

One promising strategy is to use LLMs not just as application logic but also as meta-controllers that infer latent goals and select appropriate tactics from a well-defined set of optimization strategies. This elevates the role of infrastructure from a passive executor to an active collaborator, capable of flexibly aligning system behavior with application-level values.

More concretely, modern systems expose a large and growing set of knobs and techniques: caching, batching, scheduling, multiplexing, prefetching, context

shaping, model pruning, and more. These mechanisms live at different layers of the stack and interact in complex, often opaque ways. Today, application developers must manually compose and configure these optimizations, often relying on intuition or trial-and-error. Yet, a new application will require rethinking the design of these strategies from scratch.

A central future direction is to automate this translation: given a high-level goal like "maximize responsiveness" or "minimize cost under quality constraints", the infrastructure should determine how to combine these knobs effectively. This demands systems that are workload-, history-, and application-aware, and are able to learn from usage patterns, resource conditions, and feedback loops.

Beyond automation, there is a deeper systems design opportunity: to identify generalizable principles that approximate a wide spectrum of application goals using reusable optimization strategies. Recent work like SysGPT [164] demonstrates that many sequential systems optimizations, such as batching, caching, precomputing, etc., can be unified under just three core transformations: task removal, task replacement, and task reordering. By systematically identifying and composing these transformations, systems can cover diverse optimization objectives with minimal design effort. Applying this kind of principled, composable strategy to application-aware infrastructure opens new avenues: systems can offer not just point solutions, but broad operating envelopes that approximate optimal behavior across different objectives such as latency, quality, and cost.

Bibliography

[1] AFS-Simulator. https://github.com/chhwang/schedsim.

[2] Amazon. http://www.amazon.com.

[3] CloudLab. https://www.cloudlab.us.

[4] Emulab. www.emulab.net.

[5] GPT. https://openai.com/blog/chatgpt.

[6] Philly traces. https://github.com/msr-fiddle/philly-traces.

[7] Temu. http://www.temu.com.

[8] Uber. https://www.uber.com.

[9] Amazon debuts 'Rufus,' an AI shopping assistant in its mobile app. https://techcrunch.com/2024/02/01/amazon-debuts-rufus-an-ai-shopping-assistant-in-its-mobile-app/, 2024.

[10] Assitants overview - OpenAI API. https://platform.openai.com/docs/assistants/overview, 2024.

[11] Can This A.I.-Powered Search Engine Replace Google? It Has for Me. https://www.nytimes.com/2024/02/01/technology/perplexity-search-ai-google.html, 2024.

[12] Cloud Computing Services - Amazon Web Services (AWS). https://aws.amazon.com, 2024.

[13] Embeddings - OpenAI API. https://platform.openai.com/docs/guides/embeddings, 2024.

[14] Introducing Phi-3: Redefining what's possible with SLMs . https://azure.microsoft.com/en-us/blog/introducing-phi-3-redefining-whats-possible-with-slms/, 2024.

[15] Introduction - OpenAI API. https://platform.openai.com/docs/introduction, 2024.

[16] Meet Claude Anthropic. https://www.anthropic.com/claude, 2024.

[17] Openai service pricing - microsoft azure. https://azure.microsoft.com/en-us/pricing/details/cognitive-services/openai-service/, 2024. Accessed: 2024-04-30.

[18] WhatsApp Users by Country 2024. https://worldpopulationreview.com/country-rankings/whatsapp-users-by-country, 2024.

[19] Wikipedia. https://www.wikipedia.org/, 2024.

[20] Azure openai service models. https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models, 2025.

[21] Llama. https://www.llama.com/, 2025.

[22] Model support by aws region in amazon bedrock. https://docs.aws.amazon.com/bedrock/latest/userguide/models-regions.html, 2025.

[23] Perplexity Sonar Models. https://docs.perplexity.ai/guides/model-cards#perplexity-sonar-models/, 2025.

[24] Phi. https://azure.microsoft.com/en-us/products/phi, 2025.

[25] Route prompts between models - amazon bedrock intelligent prompt routing - aws. https://aws.amazon.com/bedrock/intelligent-prompt-routing/, 2025.

[26] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's data compression proxy for the mobile web. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 367–380, Oakland, CA, May 2015. USENIX Association.

[27] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, 2010.

[28] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center tcp (dctcp). In *Proc. ACM SIGCOMM*, 2010.

[29] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[30] V. Anantharam. Scheduling strategies and long-range dependence. *Queueing Systems*, 1999.

[31] G. Appenzeller, M. Bornstein, and M. Casado. Navigating the high cost of ai compute. April 2023.

[32] I. Arawjo, C. Swoopes, P. Vaithilingam, M. Wattenberg, and E. L. Glassman. Chainforge: A visual toolkit for prompt engineering and llm hypothesis testing. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24. Association for Computing Machinery, 2024.

[33] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian. Information-agnostic flow scheduling for commodity data centers. In *Prox. Usenix NSDI*, 2015.

[34] W. Bai, L. Chen, K. Chen, and H. Wu. Enabling ecn in multi-service multi-queue data centers. In *Proc. Usenix NSDI*.

[35] F. Bang. GPTCache: An open-source semantic cache for LLM applications enabling faster answers and cost savings. In L. Tan, D. Milajevs, G. Chauhan, J. Gwinnup, and E. Rippeth, editors, *Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS 2023)*, Singapore, Dec. 2023. Association for Computational Linguistics.

[36] M. Baranishyn, B. Cudmore, and T. Fletcher. Customer service in the face of flight delays. *Journal of Vacation Marketing*, 2010.

[37] P. Behnam, J. Tong, A. Khare, Y. Chen, Y. Pan, P. Gadikar, A. Bambhaniya, T. Krishna, and A. Tumanov. Hardware-software co-design for real-time latency-accuracy navigation in tinyml applications. *IEEE Micro*, 2023.

[38] R. Bhardwaj, K. Kandasamy, A. Biswal, W. Guo, B. Hindman, J. Gonzalez, M. Jordan, and I. Stoica. Cilantro:{Performance-Aware} resource allocation for general objectives via online feedback. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, 2023.

[39] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Computer Communication Review*, 2014.

[40] O. Boxma and B. Zwart. Tails in scheduling. In *SIGMETRICS '07*, volume 34, pages 13–20.

[41] C. Caini, R. Firrincieli, and D. Lacamera. Pepsal: a performance enhancing proxy designed for tcp satellite connections. In *2006 IEEE 63rd Vehicular Technology Conference*, volume 6, pages 2607–2611, 2006.

[42] H. Chase. LangChain, Oct. 2022.

[43] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, and S. Viswanatha. Balancing efficiency and fairness in heterogeneous gpu clusters for deep learning. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. Association for Computing Machinery, 2020.

[44] C. Chen, W. Wang, S. Zhang, and B. Li. Cluster fair queueing: Speeding up data-parallel jobs with delay guarantees. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017.

[45] G. Chen, Y. Lu, Y. Meng, B. Li, K. Tan, D. Pei, P. Cheng, L. Luo, Y. Xiong, X. Wang, et al. Fast and cautious: Leveraging multi-path diversity for transport loss recovery in data centers. In *USENIX ATC*, 2016.

[46] L. Chen, J. Lingys, K. Chen, and F. Liu. AuTO: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *SIGCOMM '18*. Association for Computing Machinery, 2018.

[47] L. Chen, M. Zaharia, and J. Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

[48] T. Chen, I. Goodfellow, and J. Shlens. Net2net: Accelerating learning via knowledge transfer, 2015.

[49] Z. Chen, W. Quan, M. Wen, J. Fang, J. Yu, C. Zhang, and L. Luo. Deep learning research and development platform: Characterizing and scheduling with qos guarantees on gpu clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[50] D. Cheng, J. Rao, C. Jiang, and X. Zhou. Resource and deadline-aware job scheduling in dynamic hadoop clusters. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 956–965. IEEE, 2015.

[51] B. Choi, J. Kim, D. Cho, S. Kim, and D. Han. Appx: an automated app acceleration framework for low latency mobile app. In *Proceedings of the 14th*

International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '18, page 27–40, New York, NY, USA, 2018. Association for Computing Machinery.

[52] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review*, 45(4):393–406, 2015.

[53] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. *SIGCOMM Computer Communication Review*, 2011.

[54] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. *ACM SIGCOMM Computer Communication Review*, 2014.

[55] R. Cordingly and W. Lloyd. Enabling serverless sky computing. In *2023 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2023.

[56] D. Crankshaw, G. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2020.

[57] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *2011 Proceedings IEEE INFOCOM*, April 2011.

[58] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[59] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *Proceedings of the 2015 USENIX Annual Technical Conference*, number CONF. USENIX Association, 2015.

[60] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on*

Architectural Support for Programming Languages and Operating Systems. Association for Computing Machinery, 2014.

[61] B. G. Dellaert and B. E. Kahn. How tolerable is delay?: Consumers' evaluations of internet web sites after waiting. *Journal of interactive marketing*, 1999.

[62] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 1989.

[63] B. Derakhshan, A. R. Mahdiraji, Z. Abedjan, T. Rabl, and V. Markl. Optimizing machine learning workloads in collaborative environments. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 2020.

[64] D. Ding, A. Mallick, C. Wang, R. Sim, S. Mukherjee, V. Ruehle, L. V. S. Lakshmanan, and A. Awadallah. Hybrid llm: Cost-efficient and quality-aware query routing. In *ICLR 2024*, February 2024.

[65] F. R. Dogar, L. Aslam, Z. A. Uzmi, S. Abbasi, and Y. Kim. Cam01-3: Connection preemption in multi-class networks. In *IEEE Globecom 2006*, pages 1–6. IEEE, 2006.

[66] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. In *Proc. ACM SIGCOMM*, 2014.

[67] F. R. Dogar, I. A. Qazi, A. R. Tariq, G. Murtaza, A. Ahmad, and N. Stocking. Missit: Using missed calls for free, extremely low bit-rate communication in developing regions. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI)*, pages 1–12, 2020.

[68] F. R. Dogar and P. Steenkiste. Architecting for Edge Diversity: Supporting Rich Services Over an Unbundled Transport. In *Proc. ACM CoNext*, 2012.

[69] F. R. Dogar, P. Steenkiste, and K. Papagiannaki. Catnap: Exploiting high bandwidth wireless interfaces to save energy for mobile devices. In *Proc. ACM MobiSys*, 2010.

[70] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting congestion control for consistent high performance. In *NSDI '15*. USENIX Association, 2015.

[71] V. Dukic, S. A. Jyothi, B. Karlas, M. Owaida, C. Zhang, and A. Singla. Is advance knowledge of flow sizes a plausible assumption. In *Proc. USENIX NSDI*, 2019.

[72] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. D. Bailey, J. A. Halderman, and V. Paxson. The security impact of https interception. In *NDSS*, 2017.

[73] H. Eltigani, R. Haroon, A. Kocak, A. B. Faisal, N. Martin, and F. Dogar. WaLLM – insights from an LLM-powered chatbot deployment via WhatsApp, 2025.

[74] A. B. Faisal, H. M. Bashir, I. A. Qazi, Z. Uzmi, and F. R. Dogar. Workload adaptive flow scheduling. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. ACM, 2018.

[75] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM european conference on Computer Systems*, 2012.

[76] R. Gandhi, Y. C. Hu, and M. Zhang. Yoda: a highly available layer-7 load balancer. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16. Association for Computing Machinery, 2016.

[77] L. Gao, X. Ma, J. Lin, and J. Callan. Precise zero-shot dense retrieval without relevance labels. *arXiv preprint arXiv:2212.10496*, 2022.

[78] P. X. Gao, A. Narayan, G. Kumar, R. Agarwal, S. Ratnasamy, and S. Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15. ACM, 2015.

[79] W. Gao, Z. Ye, P. Sun, Y. Wen, and T. Zhang. Chronus: A novel deadline-aware scheduler for deep learning training jobs. In *Proceedings of the ACM Symposium on Cloud Computing*, 2021.

[80] X. Y. Geoffrey, Y. Gao, P. Golikov, and G. Pekhimenko. Habitat: A runtime-based computational performance predictor for deep neural network training. In *Proc. USENIX ATC*, 2021.

[81] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. USENIX Association, March 2011.

[82] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013.

[83] I. Giagkiozis and P. J. Fleming. Pareto front estimation for decision making. *Evolutionary Computation*, 2014.

[84] W. Gill, M. Elidrisi, P. Kalapatapu, A. Anwar, and M. A. Gulzar. Privacy-aware semantic cache for large language models. *arXiv preprint arXiv:2403.02694*, 2024.

[85] Google Cloud. Gemini 2.0 flash. `https://cloud.google.com/vertex-ai/generative-ai/docs/models/gemini/2-0-flash`, 2025.

[86] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in Multi-Resource clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[87] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM'09*.

[88] J. Griner, J. Border, M. Kojo, Z. D. Shelby, and G. Montenegro. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, June 2001.

[89] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, Boston, MA, 2019. USENIX Association.

[90] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462. USENIX Association, November 2020.

[91] N. Gupta, H. Narasimhan, W. Jitkrittum, A. S. Rawat, A. K. Menon, and S. Kumar. Language model cascades: Token-level uncertainty and beyond, 2024.

[92] A. Haeberlen, L. Phan, and M. McGuire. Metaverse as a service: Megascale social 3d on the cloud. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2023.

[93] M. Hao, H. Li, M. H. Tong, C. Pakha, R. O. Suminto, C. A. Stuardo, A. A. Chien, and H. S. Gunawi. Mittos: Supporting millisecond tail tolerance with fast rejecting slo-aware os interface. In *Proc SOSP*, 2017.

[94] M. Hao, L. Toksoz, N. Li, E. E. Halim, H. Hoffmann, and H. S. Gunawi. LinnOS: Predictability on unpredictable flash storage with a light neural network.

In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190. USENIX Association, November 2020.

[95] O. Haq, M. Raja, and F. R. Dogar. Measuring and improving the reliability of wide-area cloud paths. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, page 253–262, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[96] M. Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, New York, NY, USA, 1st edition, 2013.

[97] M. Harchol-Balter, M. Crovella, and C. Murta. To queue or not to queue?: When fcfs is better than ps in a distributed system. Technical report, Boston, MA, USA, 1997.

[98] R. Haroon and F. Dogar. Twips: A large language model powered texting application to simplify conversational nuances for autistic users. In *Proceedings of the 26th International ACM SIGACCESS Conference on Computers and Accessibility (to appear)*, 2024.

[99] X. He, K. Zhao, and X. Chu. Automl: A survey of the state-of-the-art. *Knowledge-based systems*, 212:106622, 2021.

[100] Z. He, T. Liang, W. Jiao, Z. Zhang, Y. Yang, R. Wang, Z. Tu, S. Shi, and X. Wang. Exploring human-like translation strategy with large language models. *Transactions of the Association for Computational Linguistics*, 12:229–246, 03 2024.

[101] M. Hertzum and K. Hornbæk. Frustration: Still a common user experience. *ACM Trans. Comput.-Hum. Interact.*, jan 2023. Just Accepted.

[102] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proc. USENIX NSDI*, 2011.

[103] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network, 2015.

[104] C. Hong, M. Caesar, and P. B. Godfrey. Finishing flows quickly with preemptive scheduling. *SIGCOMM Computer Communication Review*, 2012.

[105] M. K. Hui and L. Zhou. How does waiting duration information influence customers' reactions to waiting for services? *Journal of Applied Social Psychology*, 1996.

[106] C. Hwang, T. Kim, S. Kim, J. Shin, and K. Park. Elastic resource sharing for distributed deep learning. USENIX Association, 2021.

[107] R. Ibrahim. Sharing delay information in service systems: a literature survey. *Queueing Systems*, 2018.

[108] A. M. Iftikhar, F. Dogar, and I. A. Qazi. Towards a redundancy-aware network stack for data centers. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, 2016.

[109] A. Isenko, R. Mayer, and H. Jacobsen. How can we train deep learning models across clouds and continents? an experimental study. *arXiv preprint arXiv:2306.03163*, 2023.

[110] F. Jahanbakhsh, Y. Katsis, D. Wang, L. Popa, and M. Muller. Exploring the use of personalized ai for identifying misinformation on social media. CHI '23. Association for Computing Machinery, 2023.

[111] A. Jajoo, Y. C. Hu, X. Lin, and N. Deng. A case for task sampling based learning for cluster job scheduling. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, 2022.

[112] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data*

Communication, SIGCOMM '15, pages 407–420, New York, NY, USA, 2015. ACM.

[113] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. *SIGCOMM Comput. Commun. Rev.*, 2015.

[114] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 2019.

[115] D. Jiang, X. Ren, and B. Y. Lin. Llm-blender: Ensembling large language models with pairwise comparison and generative fusion. In *Proceedings of the 61th Annual Meeting of the Association for Computational Linguistics (ACL 2023)*, 2023.

[116] C. Jin, Z. Zhang, X. Jiang, F. Liu, X. Liu, X. Liu, and X. Jin. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *arXiv preprint arXiv:2404.12457*, 2024.

[117] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated SLOs for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 2016.

[118] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 437–450, Berkeley, CA, USA, 2016. USENIX Association.

[119] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang. Grand-

slam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[120] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *USENIX Annual Technical Conference*, pages 485–497, 2015.

[121] A. Karthikeyan, N. Natarajan, G. Somashekar, L. Zhao, R. Bhagwan, R. Fonseca, T. Racheva, and Y. Bansal. SelfTune: Tuning cluster managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[122] F. Lai, Y. Dai, H. V. Madhyastha, and M. Chowdhury. ModelKeeper: Accelerating DNN training via automated training warmup. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[123] T. N. Le, X. Sun, M. Chowdhury, and Z. Liu. Allox: Compute allocation in hybrid clusters. In *Proc. EuroSys*, 2020.

[124] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. *arXiv preprint arXiv:2005.11401*, 2021.

[125] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proc. ACM SIGCOMM*, 2016.

[126] D. Li, C. Chen, J. Guan, Y. Zhang, J. Zhu, and R. Yu. Dcloud: deadline-aware resource allocation for cloud computing jobs. *IEEE transactions on parallel and distributed systems*, 27(8):2248–2260, 2015.

[127] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 104–120, New York, NY, USA, 2017. ACM.

[128] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. 2017.

[129] W. Li, X. He, Y. Liu, K. Li, K. Chen, Z. Ge, Z. Guan, H. Qi, S. Zhang, and G. Liu. Flow scheduling with imprecise knowledge. In *Proc. USENIX NSDI*, 2024.

[130] R. Liaw, R. Bhardwaj, L. Dunlap, Y. Zou, J. E. Gonzalez, I. Stoica, and A. Tumanov. Hypersched: Dynamic resource reallocation for model development on a deadline. In *Proceedings of the ACM Symposium on Cloud Computing*, 2019.

[131] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez, and I. Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

[132] C. Lin, Z. Han, C. Zhang, Y. Yang, F. Yang, C. Chen, and L. Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 929–945, Santa Clara, CA, July 2024. USENIX Association.

[133] K. Lin, C. Snell, Y. Wang, C. Packer, S. Wooders, I. Stoica, and J. E. Gonzalez. Sleep-time compute: Beyond inference scaling at test-time. *arXiv preprint arXiv:2504.13171*, 2025.

[134] J. Liu, L. Li, T. Xiang, B. Wang, and Y. Qian. TCRA-LLM: Token compression retrieval augmented large language model for inference cost reduction. In H. Bouamor, J. Pino, and K. Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 9796–9810, Singapore, Dec. 2023. Association for Computational Linguistics.

[135] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo. Bgl: Gpu-efficient gnn training by optimizing graph data i/o and preprocessing. *arXiv preprint arXiv:2112.08541*, 2021.

[136] K. Mahajan, A. Balasubramanian, A. Singhvi, S. Venkataraman, A. Akella, A. Phanishayee, and S. Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[137] D. H. Maister et al. *The psychology of waiting lines*. Citeseer, 1984.

[138] M. Mandjes and B. Zwart. Large deviations of sojourn times in processor sharing queues. *Queueing Systems*, 52(4):237–250, Apr 2006.

[139] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19. Association for Computing Machinery, 2019.

[140] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*. 2019.

[141] N. Martin and F. Dogar. Divided at the edge - measuring performance and the digital divide of cloud edge data centers. *Proc. ACM Netw.*, 1(CoNEXT3), nov 2023.

[142] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 2008.

[143] S. M. Miller. Predictability and human stress: Toward a clarification of evidence and theory. Advances in Experimental Social Psychology. Academic Press, 1981.

[144] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 2018.

[145] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar. Friends, not Foes: Synthesizing Existing Transport Strategies for Data Center Networks. In *Proc. ACM SIGCOMM*, 2014.

[146] A. Munir, T. He, R. Raghavendra, F. Le, and A. X. Liu. Network scheduling aware task placement in datacenters. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16.

[147] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti. NUMFabric: Fast and flexible bandwidth allocation in datacenters. In *SIGCOMM '16*. Association for Computing Machinery, 2016.

[148] J. Nair, A. Wierman, and B. Zwart. Tail-robust scheduling via limited processor sharing. *Performance Evaluation*, 2009.

[149] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. USENIX Association, 2020.

[150] U. Naseer and T. A. Benson. Configanator: A data-driven approach to improving CDN performance. In *NSDI '22*. USENIX Association, 2022.

[151] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, Mar. 2016. USENIX Association.

[152] R. Netravali and J. Mickens. Prophecy: accelerating mobile page loads using final-state write logs. In *Proceedings of the 15th USENIX Conference on*

Networked Systems Design and Implementation, NSDI'18, page 249–266, USA, 2018. USENIX Association.

[153] R. Netravali and J. Mickens. Remote-control caching: Proxy-based url rewriting to decrease mobile browsing bandwidth. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, HotMobile '18, page 63–68, New York, NY, USA, 2018. Association for Computing Machinery.

[154] P. Ngatchou, A. Zarei, and A. El-Sharkawi. Pareto multi objective optimization. In *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*, pages 84–91, 2005.

[155] H. Ning, H. Wang, Y. Lin, W. Wang, S. Dhelim, F. Farha, J. Ding, and M. Daneshmand. A survey on the metaverse: The state-of-the-art, technologies, applications, and challenges. *IEEE Internet of Things Journal*, 2023.

[156] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, M. P. Kumar, A. See, S. Chaudhuri, G. Holland, A. Davies, S. Nowozin, P. Kohli, and M. Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025.

[157] A. Oelen and S. Auer. Leveraging large language models for realizing truly intelligent user interfaces. CHI EA '24. Association for Computing Machinery, 2024.

[158] I. Ong, A. Almahairi, V. Wu, W.-L. Chiang, T. Wu, J. E. Gonzalez, M. W. Kadous, and I. Stoica. Routellm: Learning to route llms with preference data. *arXiv preprint arXiv:2406.18665*, 2024.

[159] A. Or, H. Zhang, and M. Freedman. Resource elasticity in distributed deep learning. *Proceedings of Machine Learning and Systems*, 2020.

[160] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84. ACM, 2013.

[161] L. Pappone, A. Sacco, and F. Esposito. Mutant: Learning congestion control from existing protocols via online reinforcement learning. In *NSDI '25*. USENIX Association, 2025.

[162] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, UIST '23, New York, NY, USA, 2023. Association for Computing Machinery.

[163] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18. Association for Computing Machinery, 2018.

[164] S. Park, M. Guan, X. Cheng, and T. Kim. Principles and Methodologies for Serial Performance Optimization. In *Proceedings of the 19th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2025.

[165] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: An efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018.

[166] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proc. ACM SOSP*, 2019.

[167] A. Perkiomaki. How estimated delivery dates (edds) enhance user experience and drive transactions for ecommerce brands. September 2023.

[168] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized "zero-queue" datacenter network. In *SIGCOMM'14*.

[169] L. Popa, A. Ghodsi, and I. Stoica. Http as the narrow waist of the future internet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in*

Networks, Hotnets-IX, New York, NY, USA, 2010. Association for Computing Machinery.

[170] A. Qiao, S. K. Choe, S. J. Subramanya, W. Neiswanger, Q. Ho, H. Zhang, G. R. Ganger, and E. P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 2021.

[171] S. Rajasekaran, M. Ghobadi, and A. Akella. CASSINI: Network-Aware job scheduling in Machine Learning clusters. In *NSDI '24*. USENIX Association, 2024.

[172] M. Ramanujam, H. V. Madhyastha, and R. Netravali. Marauder: synergized caching and prefetching for low-risk mobile app acceleration. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '21, page 350–362, New York, NY, USA, 2021. Association for Computing Machinery.

[173] J. Rasley, B. Stephens, C. Dixon, E. Rozner, W. Felter, K. Agarwal, J. Carter, and R. Fonseca. Planck: Millisecond-scale monitoring and control for commodity networks. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, 2014.

[174] Z. Rasool, S. Barnett, D. Willie, S. Kurniawan, S. Balugo, S. Thudumu, and M. Abdelrazek. Llms for test input generation for semantic applications. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, CAIN '24, page 160–165, New York, NY, USA, 2024. Association for Computing Machinery.

[175] Red Hat, Inc. *Chapter 32: Linux Traffic Control*. Red Hat, Inc., 2020.

[176] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *SIGCOMM '15*.

[177] N. Salari, S. Liu, and Z. M. Shen. Real-time delivery time forecasting and promising in online retailing: When will your package arrive? *Manufacturing & Service Operations Management*, 24(3):1421–1436, 2022.

[178] L. E. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Oper. Res.*, 1968.

[179] N. K. Sharma, M. Liu, K. Atreya, and A. Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 1–16, Renton, WA, 2018. USENIX Association.

[180] S. Shekhar, T. Dubey, K. Mukherjee, A. Saxena, A. Tyagi, and N. Kotla. Towards optimizing the costs of llm usage. *arXiv preprint arXiv:2402.01742*, 2024.

[181] Y. Sheng, S. Cao, D. Li, B. Zhu, Z. Li, D. Zhuo, J. E. Gonzalez, and I. Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, Santa Clara, CA, July 2024. USENIX Association.

[182] T. Shnitzer, A. Ou, M. Silva, K. Soule, Y. Sun, J. Solomon, N. Thompson, and M. Yurochkin. Large language model routing with benchmark datasets. In *First Conference on Language Modeling*, 2024.

[183] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.

[184] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. Flexiweb: Network-aware compaction for accelerating mobile web transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, MobiCom '15, page 604–616, New York, NY, USA, 2015. Association for Computing Machinery.

[185] R. Singhal and A. Verma. Predicting job completion time in heterogeneous mapreduce environments. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.

[186] D. R. Smith. A new proof of the optimality of the shortest remaining processing time discipline. *Operations Research*.

[187] I. Stoica and S. Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. Association for Computing Machinery, 2021.

[188] L. Stolyar. Control of end-to-end delay tails in a multiclass network: Lwdf discipline optimality. *Annals of Applied Probability*.

[189] L. Suresh, M. Canini, S. Schmid, and A. Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *Proc. Usenix NSDI*, 2015.

[190] X. Tan, Y. Jiang, Y. Yang, and H. Xu. Teola: Towards end-to-end optimization of llm-based applications. *arXiv preprint arXiv:2407.00326*, 2024.

[191] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: A software-defined storage architecture. In *Proc. ACM SOSP*, 2013.

[192] J. Thorpe, P. Zhao, J. Eyolfson, Y. Qiao, Z. Jia, M. Zhang, R. Netravali, and G. H. Xu. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[193] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*. Association for Computing Machinery, 2013.

[194] M. Walfish, J. Stribling, M. N. Krohn, H. Balakrishnan, R. T. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, volume 4, pages 15–15, 2004.

[195] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. E. Ng, M. Kozuch, and M. Ryan. c-through: part-time optics in data centers. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, Aug. 2010.

[196] H. Wang, H. Tian, J. Chen, X. Wan, J. Xia, G. Zeng, W. Bai, J. Jiang, Y. Wang, and K. Chen. Towards Domain-Specific network transport for distributed DNN training. In *NSDI '24*. USENIX Association, 2024.

[197] S. Wang, O. J. Gonzalez, X. Zhou, T. Williams, B. D. Friedman, M. Havemann, and T. Woo. An efficient and non-intrusive gpu scheduling framework for deep learning training systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020.

[198] S. Wang, Y. Wang, and K. Lin. Integrating priority with share in the priority-based weighted fair queuing scheduler for real-time networks. *Real-Time Systems*, 2002.

[199] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding up web page loads with shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 109–122, Santa Clara, CA, Mar. 2016. USENIX Association.

[200] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. USENIX Association, 2022.

[201] A. Wierman, N. Bansal, and M. Harchol-Balter. A note on comparing response times in the m/gi/1/fb and m/gi/1/ps queues. *Oper. Res. Lett.*

[202] A. Wierman and M. Harchol-Balter. Classifying scheduling policies with re-

spect to higher moments of conditional response time. *ACM SIGMETRICS Performance Evaluation Review*, 2005.

[203] A. Wierman and B. Zwart. Is tail-optimal scheduling possible? *Oper. Res.*, 2012.

[204] Wikipedia contributors. Coefficient of variation — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Coefficient_of_variation&oldid=861918317`, 2018.

[205] Wikipedia contributors. K-means clustering — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=K-means_clustering&oldid=862579058`, 2018.

[206] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 2011.

[207] K. Winstein and H. Balakrishnan. Tcp ex machina: Computer-generated congestion control. In *Proc SIGCOMM*, 2013.

[208] B. Wu, Z. Zhang, Z. Bai, X. Liu, and X. Jin. Transparent {GPU} sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, 2023.

[209] D. Wu, X. Wang, Y. Qiao, Z. Wang, J. Jiang, S. Cui, and F. Wang. NetLLM: Adapting Large Language Models for Networking. In *SIGCOMM '24*. Association for Computing Machinery, 2024.

[210] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, F. Yang, and L. Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 2018.

[211] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *Proc. USENIX OSDI*, 2020.

[212] Y. Xu, A. Khare, G. Matlin, M. Ramadoss, R. Kamaleswaran, C. Zhang, and A. Tumanov. Unfoldml: Cost-aware and uncertainty-based dynamic 2d prediction for multi-stage classification. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2022.

[213] G. Yang, C. Shin, J. Lee, Y. Yoo, and C. Yoo. Prediction of the resource consumption of distributed deep learning systems. *Proc. ACM Measurement and Analysis of Computing Systems*, 2022.

[214] Z. Yang, Z. Wu, M. Luo, L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. Sifei, G. Mittal, S. Shenker, and I. Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, 2023.

[215] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[216] H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 123–133, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[217] P. Yu and M. Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610*, 2019.

[218] Q. Yu, G. Allon, and A. Bassamboo. How do delay announcements shape

customer behavior? an empirical study. *Management Science*, 63(1):1–20, 2017.

[219] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, 2017.

[220] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, and S. Garg. Model-Switching: Dealing with fluctuating workloads in Machine-Learning-as-a-Service systems. USENIX Association, 2020.

[221] K. Zhang, L. Peng, C. Wang, A. Go, and X. Liu. Llm cascade with multi-objective optimal consideration, 2024.

[222] Y. Zhang, R. Isaacs, Y. Yue, J. Yang, L. Zhang, and Y. Vigfusson. Latenseer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, 2023.

[223] H. Zhao, Z. Han, Z. Yang, Q. Zhang, F. Yang, L. Zhou, M. Yang, F. C. Lau, Y. Wang, Y. Xiong, and B. Wang. Hived: Sharing a gpu cluster for deep learning with guarantees. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2020.

[224] K. Zhao, P. Goyal, M. Alizadeh, and T. E. Anderson. Swp: Microsecond network slos without priorities. *arXiv preprint arXiv:2103.01314*, 2021.

[225] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*, 2023.

[226] P. Zheng, R. Pan, T. Khan, S. Venkataraman, and A. Akella. Shockwave: Fair and efficient cluster scheduling for dynamic adaptation in machine learning. In

20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). USENIX Association, 2023.

[227] H. Zhu, B. Zhu, and J. Jiao. Efficient prompt caching via embedding similarity. *arXiv preprint arXiv:2402.01173*, 2024.

[228] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 1999.