FedCache: Predictive GenAI Model Caching for Edge GPU Clusters

Niki Mahmoudi

A thesis submitted to the Faculty of Graduate Studies in partial fulfilment of the requirements for the degree of Master of Science

GRADUATE PROGRAM IN COMPUTER SCIENCE

York University
TORONTO, ONTARIO

July, 2026

© Niki Mahmoudi, 2026

Abstract

Serverless generative-AI inference at the edge must serve a heterogeneous, multi-tenant mix of diffusion components, including base checkpoints, LoRA adapters, ControlNets, and full pipelines. The combined footprint of these components is usually much larger than the VRAM available on a single edge node. As a result, requests often target models that are not resident in GPU memory, forcing the system to load weights from storage and incur a multi-second cold start. Reactive replacement policies, such as LRU, can decide what to evict after a miss, but cannot avoid the latency of the request that caused the miss. Avoiding this cost requires loading the model before the request arrives. We present FedCache, a predictive VRAM-caching system that formulates model prefetching and eviction as a sequential decision problem. FedCache learns over a compact macro-action space, using a demand-aware state to decide when to prefetch, evict, or idle while resolving the target model deterministically. To support edge deployments where request traces are private and non-IID, FedCache trains the policy across edge nodes with asynchronous federated policy gradients. The nodes share updates rather than raw traces, so request logs are never transmitted directly, and the server uses delay-adaptive lookahead and staleness damping to reduce the effect of slow workers. Under a trace-driven, queue-aware simulation of the Alibaba GenTD26 production trace, FedCache improves cache-hit rate over LRU and three learned predictive baselines (IceBreaker, XGBoost, and ACPM). Given memory headroom, it also attains a higher hit rate than an eviction-only Belady reference; that reference sees the whole future and loads instantaneously but cannot prefetch, so the two occupy different information settings and neither bounds the other.

The advantage persists on held-out periods, and asynchronous federation reduces convergence time as more edge nodes participate. A four-node edge deployment reproduces the hit-rate improvement (+7 percentage points over all baselines); with a demand-priority IO scheduler, FedCache also achieves up to 40% lower latency than the baselines on nodes with fast storage. This latency figure reflects model-load byte movement measured in replay, with diffusion inference emulated rather than executed, and it does not extend to nodes whose IO path is already saturated.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Hamzeh Khazaei, for his constant support and guidance throughout my graduate studies. His advice, encouragement, and insight played a key role in shaping this work and helping me grow as a researcher.

I am also grateful to my friends and colleagues in the Performant and Available Computing Systems (PACS) Lab. Their support, discussions, and collaboration made this journey both productive and enjoyable, and greatly contributed to my academic and personal development.

Finally, I would like to thank my parents, who believed in me even when I doubted myself. Their unconditional support, patience, and love have been the foundation of everything I have achieved, and I am deeply grateful to them.

Table of Contents

Abstract **ii**

Acknowledgements **iv**

Table of Contents **v**

List of Tables **ix**

List of Figures **xii**

Acronyms **xv**

Declaration of Authorship **xvi**

1 Introduction **1**

1.1 Thesis Contributions 4

1.2 Thesis Organization 6

2 Background and Related Work **7**

2.1 Preliminaries 7

2.1.1 Serverless GenAI Serving 7

2.1.2 Reinforcement Learning and Proximal Policy Optimization . 8

2.1.3 Federated Learning 9
2.2 Serverless Cold Starts 10
2.3 GPU Model Serving and Multi-Adapter Inference 11
2.4 Learned Caching and Prefetching 11
2.5 Federated and Asynchronous Reinforcement Learning 12
2.6 Summary and Research Gap 13

3 System Model and Problem Formulation 14
3.1 Serverless GenAI Serving on Edge GPU Clusters 14
3.2 VRAM Contention and Model Heterogeneity 15
3.3 The Generative AI Cold-Start Penalty 16
3.4 Problem Formulation 17

4 FedCache Architecture 19
4.1 Decentralized MDP: Macro-Action Space and Demand-Aware State 19
4.1.1 Demand-Aware State 20
4.1.2 Macro-Action Space 21
4.1.3 Reward 22
4.2 The Local Edge Agent: Predictive Policy 23
4.2.1 The Exploration Challenge 23
4.2.2 Local Reinforcement Learning 23
4.3 Delay-Adaptive Lookahead and Local Updates 24
4.4 The Asynchronous Federated Aggregator 25

5 Experimental Setup and Methodology 27
5.1 The Production Trace 27

5.2 Model-Size Parameterization and VRAM Fragmentation 31
5.3 Baselines 32
5.4 Evaluation Metrics 33

6 Results and Evaluation **35**
6.1 End-to-End Comparison 35
6.1.1 Decomposing the Switching Cost 37
6.2 VRAM Contention and the Low-Memory Crossover 38
6.3 Comparison with an Eviction-Only Belady Reference 40
6.4 Sensitivity to the Model-Size Mapping 42
6.5 Generalization to Held-Out Workloads 44
6.6 Asynchronous Scalability and Straggler Tolerance 45
6.7 Training-Scheme Comparison 47
6.8 Ablation Studies 49
6.8.1 Learning from Scratch 49
6.8.2 Demand-Aware Admission Gate 51
6.8.3 Robustness to Federation Hyperparameters 51
6.8.4 Proximal regularization 52
6.8.5 Reward shaping 52
6.8.6 Queue window 53

7 Deployment on Real Edge GPU Clusters **55**
7.1 Deployment Setup 55
7.2 Experimental Evaluation 57
7.3 Latency with the Demand-Priority Scheduler 57
7.4 Validity Bounds 58

8 Discussion and Conclusion **60**

8.1 Summary 60

8.2 Discussion 61

8.2.1 Sources of the Gains 61

8.2.2 Key Design Factors 62

8.3 Limitations 64

8.3.1 Model-Size Parameterization 64

8.3.2 Simulation Fidelity 65

8.3.3 Scope and Threats to Validity 66

8.4 Future Work 66

Bibliography **68**

A Reproducibility Details **75**

A.1 Trace Preprocessing and Splits 75

A.2 Model Universe and Size Parameterization 77

A.3 Simulator Semantics 78

A.4 Policy Network and Training 81

A.5 Baseline Configurations 82

A.6 Physical Cluster 84

A.7 Demand-Priority IO Scheduler 87

A.8 Mapping from Results to Scripts 88

List of Tables

5.1 Key training hyperparameters. 30

6.1 End-to-end multi-objective comparison on GenTD26 (N=5, C=15 GB, 5000-step evaluation); a single deterministic run, identical across evaluation replicas. Lower latency, VRAM, and thrash are better; best per column in bold. 37

6.2 Thrash decomposed (N=5, C=15 GB, 5000-step evaluation). Demand and prefetch are byte-moving loads whose target is the current or a future request; evictions are those actually performed (all implicit here, to make room for a load); wasted counts prefetches evicted before serving a request. The final column is the load-plus-explicit-eviction "thrash" of Table 6.1, which counts only explicit actions and so reduces to the load count. 39

6.3 Sensitivity to the synthetic model-size mapping (C=15 GB, 3000-step evaluation, FedCache retrained from scratch under each mapping). Each row is a different assignment of the 100 models to footprints; "best learned" is the strongest of IceBreaker, XGBoost, and ACPM. The crossover is the smallest capacity at or above which FedCache stays $\geq$ LRU. FedCache leads every learned baseline in all six mappings; it trails LRU only in the heavy-tailed mapping, whose crossover (20 GB) places C=15 in the tight-memory regime. . 44

6.4 Training-scheme comparison, budget-matched at 150 local PPO updates (N=5, C=15 GB, policy 660 KB). All schemes are evaluated on the same full request stream; only training-time data access and aggregation differ. Wall-clock is simulated time to reach a 95% hit rate. Final quality is identical throughout; federation buys convergence time, and asynchrony buys a further reduction over the synchronous barrier at equal communication volume. Centralized training moves no model weights, but requires the raw request traces to be pooled, which is the cost federation exists to avoid. 49

7.1 Experimental evaluation on the four-node matched (fast) H100 cluster (mean of four MIG slices, 5000 requests). Simulation hit rates (Table 6.1) are shown for reference; experimental latency is reported with the demand-priority IO scheduler off and on. 56

7.2 Per-node FedCache latency (ms), demand-priority IO scheduler off vs. on. On the four matched fast nodes, the scheduler drops FedCache below LRU and removes node-to-node variance; on the two slow nodes (kept for contrast), it increases latency because the IO path is already saturated. 58

A.1 Size tiers used to parameterize the $M = 100$ model universe. The ranges are the measured footprints of the corresponding component classes on the physical cluster; the share column is the probability with which an anonymized trace identifier is assigned to that tier. . 78

A.2 Simulator constants. Sizes are normalized internally by a fixed 80 GB scale factor. 79

A.3 Per-node cold-load calibration on the four matched nodes. 85

A.4 Reported results and the scripts that generate them. 89

List of Figures

4.1 FedCache overview. On each edge node, the pending request queue is summarized into a demand-aware state, a three-way PPO policy decides *when* to act, and the environment resolves the concrete model target deterministically, filtered by the demand-aware admission gate at the VRAM cache. Nodes train locally from the lookahead-extrapolated parameters $\tilde{\theta}$ and report only pseudo-gradients; the asynchronous aggregator applies each update with staleness damping and returns fresh parameters only to the reporting node. 20

6.1 Hit rate (left) and average latency (right) versus per-node VRAM capacity, for FedCache and all baselines. FedCache is trained once at C=15 GB and evaluated across the range without retraining, so this is a capacity-transfer (robustness) curve rather than a per-capacity optimum. The learned baselines (IceBreaker, XGBoost, ACPM) remain at or below LRU; under tight memory (the thrash regime) FedCache slightly trails LRU while staying above the learned baselines, and it outperforms all baselines above the $\sim$7 GB crossover. 36

6.2 Hit rate of FedCache, the learned baselines (IceBreaker, XGBoost, ACPM), and LRU against Belady's offline eviction-optimal. *The two settings are not comparable and neither bounds the other*: the Belady reference sees the whole future and loads instantaneously but is eviction-only, whereas FedCache sees an H-step queue and pays the real load delay $\Delta(m)$ but may prefetch. Above a $\sim$8.5 GB crossover FedCache attains the higher hit rate, by staging first-touch requests that a load-on-miss policy must miss; this is not evidence of surpassing an offline optimum. The baselines remain at or below LRU. 42

6.3 Hit rate on the training split versus a held-out later temporal split, for FedCache and all baselines. FedCache maintains a larger margin over LRU and the learned baselines on unseen traffic. 45

6.4 Wall-clock time (in simulator time units) to reach 95% hit rate versus the number of nodes N. Convergence time falls steeply and then saturates. Scaling is near-linear for small N before flattening as the per-update compute floor dominates. 46

6.5 From-scratch FedCache learning curve. With no expert demonstrations, the macro-action policy converges to a $\sim$98% steady-state hit rate within $\sim$10–20 global updates, well above the LRU reference. . 50

6.6 Demand-aware admission gate ablation (5000-step evaluation). Disabling the gate more than doubles prefetch churn (454 $\rightarrow$ 984) and drops the hit rate below LRU; the gate secures FedCache's high hit rate at thrash below LRU. 53

6.7 Ablations. (a) FedProx μ under non-IID clients: large μ over-regularizes; small μ has no effect. (b) Prefetch-cost sweep: performance remains flat because the demand-aware gate keeps prefetching beneficial. (c) Queue window H: hit is ~ 0 below the load delay, climbs sharply as H clears it, then is flat while thrash grows. 54

Acronyms

ACPM	Adaptive Container Provisioning Model
AFedPG	Asynchronous Federated Policy Gradient
CDN	Content Distribution Network
DNN	Deep Neural Network
FaaS	Function as a Service
FFT	Fast Fourier Transform
FL	Federated Learning
GAE	Generalized Advantage Estimation
GenAI	Generative Artificial Intelligence
GPU	Graphics Processing Unit
IID	Independent and Identically Distributed
IO	Input/Output
LoRA	Low-Rank Adaptation
LRU	Least Recently Used
LSTM	Long Short-Term Memory
MDP	Markov Decision Process
MIG	Multi-Instance GPU
PCIe	Peripheral Component Interconnect Express
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SLA	Service-Level Agreement
VRAM	Video Random-Access Memory

Declaration of Authorship

I, Niki Mahmoudi, hereby declare that this thesis titled, “FedCache: Predictive GenAI Model Caching for Edge GPU Clusters ”, and the work presented herein are my own. Parts of this thesis have been submitted as a manuscript to the ACM Symposium on Cloud Computing (SoCC 2026), co-authored with Arian Boukani and Dr. Hamzeh Khazaei, and are currently under review.

The design of FedCache, the problem formulation, the simulator, the training and evaluation code, and all simulation experiments (Chapters 3–6) are my own work. The physical-cluster deployment and measurement infrastructure used for the real-hardware validation in Chapter 7 was developed and operated jointly with Arian Boukani; the analysis and interpretation of those results are my own. This work was carried out under the supervision of Dr. Hamzeh Khazaei.

Chapter 1

Introduction

Generative AI (GenAI) inference has become a production workload in many domains, including healthcare, where latency and reliability are operational requirements rather than convenience metrics [1]. Modern diffusion services also differ from conventional model-serving workloads. A request is rarely served by one monolithic model; instead, it is assembled from a pipeline of components that may include a base checkpoint, one or more Low-Rank Adaptation (LoRA) adapters, and optional ControlNet modules, depending on the specific tenant and request [2, 3]. Production traces show thousands of base checkpoints and tens of thousands of LoRA adapters in circulation, making their combined footprint far larger than the GPU memory (VRAM) available on a single serving node [4]. This pressure is especially acute at the edge, where inference is increasingly placed near users for latency and data-locality reasons [5, 6], but where each site has limited VRAM and observes its own private request mix.

The main performance bottleneck in this setting is the *cold start*. Edge nodes may support applications with strict service-level agreements for reliability and

responsiveness [7]; reducing cold starts, therefore, affects both user experience and service viability. When a request targets a model that is not already in VRAM, the system must stream multi-gigabyte weights from storage before serving the request. In production traces, this operation can take from several seconds to tens of seconds [4]. Since the working set is much larger than VRAM and tenant demand is bursty and skewed, these misses occur frequently and dominate tail latency. Which models are kept resident in VRAM is therefore a first-order design decision for GenAI serving.

Conventional cache management addresses only part of the problem. Replacement policies such as LRU, and learned eviction heuristics [8, 9], are reactive: they choose what to keep or evict after a miss has been observed. At that point, however, the cold-start latency for the current request has already been incurred. To avoid the miss rather than merely respond to it, the system must pre-warm the model into VRAM before the request arrives. Recent work on serverless inference makes a similar observation: when large artifacts dominate startup, keeping functions warm is not sufficient, and opportunistic pre-loading is needed to hide loading latency [10, 11]. This changes the problem from reactive eviction to predictive prefetching: staging models into VRAM before the requests that need them are served, under a hard VRAM-capacity constraint.

Learning such a prefetching policy is difficult for three reasons. *First*, the action space is large and heterogeneous. At each step, the agent may need to choose among hundreds of models with different memory footprints while respecting a strict VRAM capacity constraint. *Second*, the reward is delayed and asymmetric. A useful prefetch is rewarded only when the corresponding request arrives later,

while an unnecessary prefetch consumes bandwidth and may evict a model that will soon be needed. *Third*, training data is decentralized. Each edge site sees only its own tenant traffic, and those traces cannot be centralized because they are private and non-IID (that is, not independent and identically distributed). At the same time, heterogeneous node speeds make synchronous training vulnerable to stragglers.

This thesis presents **FedCache**[1], a system that learns a predictive prefetch-and-evict policy for GenAI VRAM caching and trains it across edge nodes using asynchronous federated reinforcement learning. FedCache addresses the three challenges above. To reduce the action space, it formulates caching as a Markov decision process (MDP) over three macro-actions: do nothing, prefetch the most useful upcoming model, or evict the least useful resident model. The concrete target of each macro-action is computed from a demand-aware state derived from a short window over the pending request queue. As a result, the agent learns *when* to act instead of learning a large per-step model-selection policy. This same formulation reduces the exploration burden: the agent no longer needs to discover the correct model target among M choices before receiving delayed feedback. To train without centralizing private traces, FedCache uses asynchronous federated policy gradients. Nodes send gradient updates rather than request logs; the server applies updates as they arrive; and delay-adaptive lookahead with staleness damping limits the effect of slow workers.

We evaluate FedCache on the Alibaba GenTD26 production trace against

[1]Our implementation includes the simulator engine, all baselines, the experiment scripts that generate every figure, and the code for the experimental evaluation on cloud GPUs. It is publicly available at `https://github.com/pacslab/fedCache`.

LRU, three learned predictive baselines (IceBreaker, XGBoost, and ACPM), a synchronous federated variant, and an eviction-only Belady reference. Under the trace-driven, queue-aware simulation described in Chapter 5, FedCache achieves the highest cache-hit rate among the policies compared and reduces average request latency by cutting cold starts. With sufficient memory headroom, it also attains a higher hit rate than the eviction-only Belady reference, because it can prefetch first-touch requests that a load-on-miss policy must miss. That reference is granted the whole future and instantaneous loads but cannot prefetch, so the two occupy different information settings and neither bounds the other; the comparison is not evidence of surpassing a general offline optimum. The policy retains its advantage on held-out periods. On the systems side, asynchronous federation reduces wall-clock convergence time as nodes are added and avoids the straggler bottleneck of synchronous training. A demand-aware admission gate keeps prefetch churn below reactive LRU, allowing FedCache to improve hit rate, latency, and switching cost with only a modest increase in VRAM occupancy.

1.1 Thesis Contributions

This thesis makes the following contributions:

1. We reframe GenAI VRAM management from reactive eviction to *predictive prefetching*, and formalize it as a multi-objective MDP with an explicit load-delay model in which prefetching is the only way to avoid cold starts (Chapter 3).

2. We design a learnable prefetch policy based on a macro-action space and

a demand-aware state. This design makes reinforcement learning tractable without expert demonstrations, despite the otherwise large action space and delayed rewards (Chapter 4).

3. We develop an asynchronous federated training scheme for edge GPU clusters that avoids directly transmitting raw request traces and tolerates stragglers through delay-adaptive lookahead, staleness-scaled gradient updates, and targeted broadcast (Chapter 4). We are deliberately narrow about what this buys: on the evaluated workload it reduces wall-clock time-to-target and keeps traces on-node, but it does not change the final policy quality (Section 6.7); we present it as a systems and privacy property rather than as a source of the serving gains.

4. We evaluate FedCache on the Alibaba GenTD26 trace against reactive, learned, and federated baselines and an eviction-only Belady reference. Under the simulated conditions, FedCache improves latency, hit rate, and prefetch churn, and its admission gate prevents predictive loading from becoming excessive churn (Chapter 6).

5. We *validate the trace-driven simulation through real experimentation* on a four-node H100 MIG cluster with physical storage-to-GPU transfers. The real experiments are consistent with the central prediction of the simulator, with FedCache's hit-rate advantage widening to +7 percentage points on real GPUs. They also delineate the storage-bandwidth conditions under which the latency benefit transfers (Chapter 7).

1.2 Thesis Organization

This thesis is organized as follows. Chapter 2 provides the necessary background and reviews related research, positioning our contributions within the existing literature. Chapter 3 formalizes the system model and the predictive prefetching problem. Chapter 4 presents the design of **FedCache**, including the macro-action MDP and the asynchronous federated training scheme. Chapter 5 describes the experimental setup and methodology, and Chapter 6 evaluates the proposed system in simulation. Chapter 7 validates the trace-driven simulation through real experimentation on a physical GPU cluster. Finally, Chapter 8 concludes the thesis: it summarizes the contributions, discusses the sources of the gains and the limitations of this study, and outlines directions for future research. Appendix A documents the trace preprocessing, the model-size parameterization, the simulator semantics, the baseline and training configurations, and the cluster setup in the detail needed to reproduce every reported result.

Chapter 2

Background and Related Work

This chapter first introduces the two learning frameworks that FedCache builds on—reinforcement learning and federated optimization—and then situates FedCache in the literature. We review cold-start mitigation in serverless computing, GPU model serving and multi-adapter inference, learned caching and prefetching, and federated and asynchronous reinforcement learning, and we identify the gap that FedCache addresses.

2.1 Preliminaries

2.1.1 Serverless GenAI Serving

In serverless computing, the platform provisions and scales resources on behalf of the application: tenants submit requests, and the provider decides where and when the corresponding computation runs. Applied to generative AI, each image-generation request is served by a *pipeline* assembled from components: a base diffusion checkpoint containing the multi-gigabyte denoising network, optionally

augmented with Low-Rank Adaptation (LoRA) adapters that specialize the model for a tenant or style at a fraction of the size, and ControlNet modules that condition generation on structural inputs [2, 3]. To serve a request, every component of its pipeline must reside in GPU memory (VRAM). When a component is missing, its weights must be read from storage, staged through host memory, and copied over PCIe into VRAM—an operation that takes seconds to tens of seconds at gigabyte scale and is the cold start studied in this thesis [4]. Because production platforms serve thousands of distinct pipelines whose combined footprint far exceeds any single node's VRAM, deciding which models to keep resident—and when to load them—is the central resource-management problem this thesis addresses.

2.1.2 Reinforcement Learning and Proximal Policy Optimization

Reinforcement learning (RL) studies sequential decision-making: an agent repeatedly observes the state of an environment, selects an action, and receives a scalar reward, with the goal of maximizing the cumulative reward it collects over time [12]. The interaction is formalized as a Markov decision process (MDP) $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$ the set of actions, P the transition dynamics, R the reward function, and $\gamma \in (0, 1)$ a discount factor that trades immediate against future reward. A policy $\pi_\theta(a \mid s)$ maps states to a distribution over actions, and learning seeks parameters θ that maximize the expected discounted return $\mathbb{E}[\sum_t \gamma^t R_t]$. This framing fits cache management naturally: each prefetch or eviction is an action whose benefit (a future cache hit) arrives only after a delay, so the learner must perform credit assignment over long horizons rather than optimize each decision in isolation.

Policy-gradient methods optimize π_θ directly by ascending the gradient of the expected return. Actor–critic variants pair the policy (the *actor*) with a learned value function (the *critic*) that estimates how good a state is, which reduces the variance of the gradient estimate. Proximal Policy Optimization (PPO) [13], the local learner used in this thesis, additionally clips the policy-update ratio so that each update stays close to the previous policy; this simple constraint makes training markedly more stable. Advantage estimates are computed with Generalized Advantage Estimation (GAE) [14], which balances the bias and variance of the return estimates through an exponential weighting parameter.

2.1.3 Federated Learning

Federated learning (FL) trains a single shared model across many clients without centralizing their data: each client computes an update on its private data, and only the update—not the data—is sent to a coordinating server [15]. In the canonical FedAvg protocol, the server broadcasts the current model, waits for all selected clients to return their updates, averages them, and repeats. Synchronous rounds, however, move at the pace of the slowest client, so *asynchronous* variants apply each update as it arrives [16, 17]. Asynchrony introduces *staleness*: an update may have been computed against an older model version than the one it is applied to, and unmitigated staleness can destabilize training. Federated *reinforcement* learning extends these ideas from supervised objectives to sequential decision-making, where clients share policy-gradient updates instead of raw trajectories [18, 19]. FedCache adopts this setting because edge request traces are commercially sensitive: nodes learn a shared caching policy while exchanging only gradients.

2.2 Serverless Cold Starts

A cold start is the delay required to prepare a function's execution environment before it can handle a request. It remains one of the central challenges in serverless computing. Recent measurement studies and position papers show that cold-start behaviour remains workload- and platform-dependent rather than a solved artifact of legacy FaaS stacks [11, 20]. Existing mitigations include predictive keep-alive and pre-warming based on historical invocation patterns [21], opportunistic pre-loading for serverless inference [10], deep-learning forecasts for proactive container provisioning [22], snapshot-and-restore mechanisms for lightweight sandboxes [23, 24, 25], scalable container provisioning at production scale [26], transparent data caching for serverless applications [27], and heterogeneity-aware warming across resource tiers [28]. These approaches primarily target lightweight function containers, where the dominant cost is runtime or sandbox initialization. GenAI inference has a different cost structure: the serving environment can remain active, while the expensive operation is loading large model weights into GPU memory. Recent work accelerates this weight-loading path itself with loading-optimized checkpoint formats and locality-aware scheduling [29], which is complementary to deciding *which* models to stage and *when*. FedCache focuses on this weight-loading regime, where the pre-warmed object is a large, variably sized model rather than a lightweight container.

2.3 GPU Model Serving and Multi-Adapter Inference

Recent systems work has addressed large-model serving under tight GPU-memory capacity. For multi-tenant LoRA serving, systems such as S-LoRA, Punica, and dLoRA batch requests across many adapters and manage adapter residency to improve throughput [30, 31, 32]. Beyond adapters, Clockwork exploits the deterministic performance of DNN inference to bound tail latency while swapping models through GPU memory [33], and vLLM manages GPU memory for LLM serving with paging-inspired allocation [34]. Recent work on LLM and dynamic ML-workflow serving similarly emphasizes memory pressure and runtime orchestration in multi-tenant inference settings [35, 36]. Production studies of diffusion serving further characterize model heterogeneity and bursty demand, motivating memory-aware scheduling [4, 37]. These systems mainly make reactive residency decisions optimized for serving and batching efficiency. They do not learn to stage models ahead of demand, which is the gap addressed by FedCache.

2.4 Learned Caching and Prefetching

Learned caching replaces fixed hand-designed rules with policies adapted to the access stream. Belady's offline-optimal algorithm [38] provides an eviction-optimal reference. Online learned policies adapt to access patterns [9] or imitate Belady from logged traces [8, 39], and learned prefetchers predict future accesses to hide latency [40]. Deep reinforcement learning has also been applied to content caching at the network edge [41] and to cluster resource management more broadly [42].

Much of this work assumes uniform-size or approximately uniform-cost cache objects, such as memory lines or CDN objects. GenAI model caching violates these assumptions: objects differ greatly in size, and a miss can trigger a size-dependent, multi-second load. FedCache therefore uses a reward and macro-action design that is specific to heterogeneous model objects and high miss costs.

2.5 Federated and Asynchronous Reinforcement Learning

Federated learning trains a shared model across clients without centralizing private data. Representative methods include FedAvg [15], proximal approaches such as FedProx for statistical and systems heterogeneity [43], and asynchronous aggregation for straggler mitigation [16, 17]. Federated reinforcement learning extends this idea to sequential decision-making [18]. More recent asynchronous federated policy-gradient methods show convergence with near-linear speedup through delay-adaptive corrections [19]. FedCache uses this asynchronous policy-gradient framework for a systems resource-management problem rather than a standard control benchmark, and combines it with a macro-action MDP that makes GenAI VRAM caching learnable from scratch. Related edge-caching work has also explored multi-agent and elastic federated learning for cooperative caching [44] and federated ensemble model-based RL for sequential edge-computing decisions [45]. FedCache builds on this line by focusing on the heterogeneous, high-miss-cost regime of GenAI model serving.

2.6 Summary and Research Gap

Overall, prior work either mitigates cold starts for lightweight serverless functions, serves multi-adapter models reactively, learns caching policies for more uniform objects, or applies federated RL to general control problems. FedCache sits in the gap between these lines: it formulates GenAI VRAM management as predictive model prefetching and learns the policy through asynchronous federated reinforcement learning across edge GPU clusters, sharing gradients rather than transmitting raw request traces. We are not aware of prior work that combines these elements, although we state this as positioning relative to the literature surveyed above rather than as a claim of priority.

Chapter 3

System Model and Problem Formulation

This chapter formalizes the setting: the edge serving fleet, the VRAM contention that makes model caching a packing problem, the cold-start penalty that reactive policies cannot avoid, and the multi-objective optimization problem that FedCache solves.

3.1 Serverless GenAI Serving on Edge GPU Clusters

We consider a set of N edge serving nodes $\mathcal{N} = \{1, \ldots, N\}$. Each node is a GPU server that handles GenAI inference requests, such as image, video, text, or voice generation, for tenants local to its site. Because sites serve different tenant populations and their traces are commercially sensitive, each node observes a private and non-IID request stream. The distribution over requested models can

differ across sites, so a node cannot assume that its local traffic is representative of the global workload, and it cannot send raw traces to a central trainer. Let $\mathcal{M}$ be the universe of servable models, with $|\mathcal{M}| = M$. We abstract each composed inference pipeline, consisting of a base checkpoint, its LoRA adapters, and optional ControlNet modules, as a single cacheable model $m \in \mathcal{M}$ with an identifier and a memory footprint. Node k receives a discrete request stream $\rho^k = (r_1^k, r_2^k, \dots)$, where request r_t^k targets model $m_t^k \in \mathcal{M}$ at step t. We omit the node superscript when describing a single node.

3.2 VRAM Contention and Model Heterogeneity

Each node has a fixed VRAM capacity C (GB) for resident models. Models have different sizes: a size function $s : \mathcal{M} \to \mathbb{R}_{>0}$ maps each model m to its VRAM footprint $s(m)$ in GB. These footprints range from small LoRA adapters to large multi-stage pipelines and can differ by nearly two orders of magnitude [4]. The aggregate footprint is much larger than the capacity of a single node,

$$\sum_{m \in \mathcal{M}} s(m) \;\gg\; C, \tag{3.1}$$

so only a small subset of models can reside in VRAM at any time. Let $A_t \subseteq \mathcal{M}$ denote the set of available models, meaning models that are resident and ready to serve requests at step t. Together with the in-flight *warming* loads W_t that also occupy VRAM (defined below), it must satisfy

$$\sum_{m \in A_t \cup W_t} s(m) \;\leq\; C \qquad \forall t. \tag{3.2}$$

Because models have different sizes, admission is a packing problem rather than a simple slotted-cache decision: loading one large pipeline may require evicting several smaller adapters, while many adapters may fit in the space required by a single large model.

3.3 The Generative AI Cold-Start Penalty

Serving a request requires its target model to be available. If $m_t \in A_t$, the request is a hit and incurs a small service latency ℓ_{hit}. If $m_t \notin A_t$, the request is a miss and the system must load the model's weights into VRAM before serving it. The resulting cold-start latency increases with model size,

$$\ell_{\text{miss}}(m) \;=\; \ell_{\text{load}}\Big(s(m)\Big) \;\gg\; \ell_{\text{hit}}, \tag{3.3}$$

and production traces report values from seconds to tens of seconds [4]. This delay is therefore the dominant contributor to tail latency.

Loading is not instantaneous. A load of model m initiated at step t reserves VRAM immediately, but the model becomes available only after a size-dependent delay $\Delta(m) \geq 1$, corresponding to the time needed to stream its weights. Let W_t denote the set of warming models, i.e., loads that have been issued but have not yet completed; these models also count against capacity. This delay is what limits reactive policies. If a miss for m_t is observed at step t, then a load started in response cannot finish until at least $t + \Delta(m_t)$, after the current request has already experienced the cold start. Thus, m_t can be a hit only if its load began at some step $t' \leq t - \Delta(m_t)$. A policy that acts only after a miss can improve later

requests, but it cannot remove the miss that triggered the action. Avoiding cold starts requires predictive prefetching: the model must be staged before the request that needs it is served.

3.4 Problem Formulation

At each step, a control policy π observes the recent request history and the memory state, and may initiate a prefetch, evict a resident model, or idle. Let ℓ_t be the latency of request t under π:

$$\ell_t = \begin{cases} \ell_{\text{hit}}, & m_t \in A_t \quad \text{(hit)}, \\ \ell_{\text{miss}}(m_t), & m_t \notin A_t \quad \text{(cold start)}. \end{cases} \tag{3.4}$$

The primary objective is to minimize expected average request latency subject to the memory-capacity constraint:

$$\min_{\pi} \ \mathbb{E}_{\rho}\left[\frac{1}{T}\sum_{t=1}^{T} \ell_t\right] \quad \text{s.t.} \quad \sum_{m \in A_t \cup W_t} s(m) \le C \ \ \forall t. \tag{3.5}$$

Here W_t is the set of warming models: loads that have started but have not yet completed. Since they already reserve memory, they count against the capacity C with the available set A_t. Prefetching also has a resource cost. Loads and evictions consume storage and PCIe bandwidth and can increase transient occupancy. We therefore use a multi-objective cost that augments latency with normalized VRAM occupancy and a switching penalty:

$$\min_{\pi} \ \mathbb{E}_{\rho}\left[\frac{1}{T}\sum_{t=1}^{T} \Big(\ell_t + \lambda_{\text{v}}\, u_t + \lambda_{\text{s}}\, c_t\Big)\right], \tag{3.6}$$

where $u_t = \Big(\sum_{m \in A_t \cup W_t} s(m)\Big)/C$ is normalized VRAM occupancy, c_t counts load and evict actions at step t, and $\lambda_{\mathrm{v}}, \lambda_{\mathrm{s}} \geq 0$ control the relative weight of resource costs.

Two properties make this problem difficult. First, the policy is online: it must decide what to prefetch without observing future requests. Second, the policy must combine a discrete admission/eviction choice over a large heterogeneous model set with long-horizon credit assignment, because a prefetch pays its cost immediately but yields a benefit only if the predicted request arrives after the load completes. These properties motivate an MDP formulation and a learned policy, which we develop in the next chapter.

Chapter 4

FedCache Architecture

This chapter presents the design of FedCache: the macro-action MDP and demand-aware state that each edge agent observes, the local PPO learner, and the asynchronous federated aggregation scheme that trains a single global policy across edge nodes without centralizing request traces. Figure 4.1 gives an overview of the architecture and of the training loop that connects the edge agents to the aggregator.

4.1 Decentralized MDP: Macro-Action Space and Demand-Aware State

Each node solves the online problem from Chapter 3 as an MDP [12] $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$. Here $\mathcal{S}$ is the state space, $\mathcal{A}$ is the macro-action set, P denotes the transition dynamics induced by the request stream and simulator, R is the per-step reward in (4.5), and $\gamma \in (0, 1)$ is the discount factor. The agent learns a policy π_θ that maximizes the expected discounted return, $\mathbb{E}[\sum_t \gamma^t R_t]$.

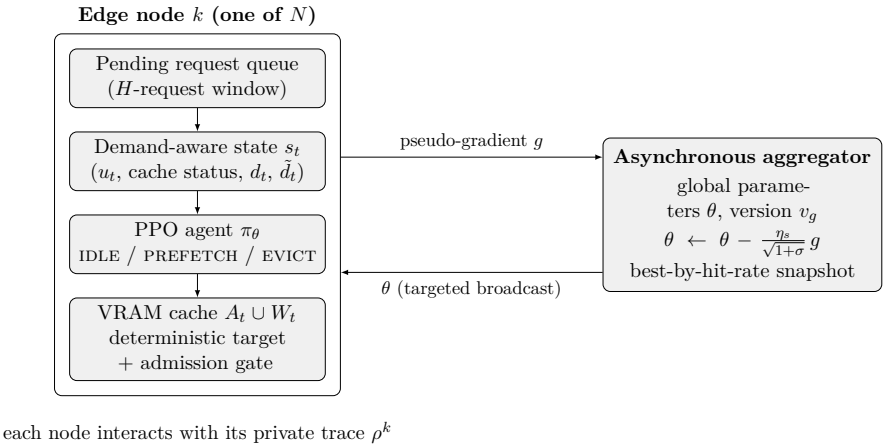

FIGURE 4.1: FedCache overview. On each edge node, the pending request queue is summarized into a demand-aware state, a three-way PPO policy decides *when* to act, and the environment resolves the concrete model target deterministically, filtered by the demand-aware admission gate at the VRAM cache. Nodes train locally from the lookahead-extrapolated parameters $\tilde{\theta}$ and report only pseudo-gradients; the asynchronous aggregator applies each update with staleness damping and returns fresh parameters only to the reporting node.

4.1.1 Demand-Aware State

The agent observes a queue of pending requests: the next H already-admitted requests $m_t, \ldots, m_{t+H-1}$ waiting for service. This queue is observable because a backlog forms when diffusion inference is slower than the arrival of the request; it is not an oracle forecast of future traffic. The mechanism is accordingly better described as queue-aware scheduling with exact knowledge of already-admitted requests than as forecasting in the conventional sense, and it presumes a deployment in which the pending queue is at least H requests deep. Under light load, where the queue is shallow or empty, the queue window carries little information and the policy has correspondingly little to act on (Chapter 7). From this queue, we

compute a recency-weighted demand vector $d_t \in [0,1]^M$:

$$d_t[m] \;=\; \frac{1}{Z_t} \sum_{j=0}^{H-1} e^{-j/\tau}\, \mathbb{1}[m_{t+j} = m]\,, \tag{4.1}$$

where τ controls the recency decay and $Z_t = \max_{m'} \sum_j e^{-j/\tau} \mathbb{1}[m_{t+j} = m']$ normalizes the vector by its peak value. We also define an unmet-demand vector that removes models already available or warming:

$$\tilde{d}_t[m] \;=\; d_t[m] \Big(1 - \mathbb{1}[m \in A_t \cup W_t]\Big). \tag{4.2}$$

The state s_t concatenates the normalized VRAM occupancy u_t, a cache-status vector that encodes each model as available (1), warming ($\frac{1}{2}$), or absent (0), per-model recency features, d_t, $\tilde{d}_t$, and raw queue-window features. The unmet-demand vector is central: its support gives the prefetch candidates, and its largest entry identifies the strongest upcoming demand not already covered by the cache.

4.1.2 Macro-Action Space

A direct policy that selects one of M models to load or evict would require a large discrete action head. Combined with delayed feedback, this makes exploration unreliable. FedCache instead exposes three macro-actions to the agent: IDLE, PREFETCH, and EVICT. Each macro-action has a small action index, and its concrete target is resolved deterministically from the state. PREFETCH selects the earliest upcoming request whose target is not already available or warming, fits

within capacity, and can be loaded before it is needed:

$$m^* = m_{t+j^*}, \tag{4.3}$$

$$j^* = \min\{\, j \geq \Delta(m_{t+j}) : m_{t+j} \notin A_t \cup W_t,\ s(m_{t+j}) \leq C \,\}. \tag{4.4}$$

The condition $j \geq \Delta(m_{t+j})$ ensures that only catchable requests are prefetched. A demand-aware admission gate then admits the candidate only when there is free space or when the candidate's demand is higher than the demand of the model it would displace. This prevents speculative prefetches from causing unnecessary churn under contention. EVICT removes the resident model with the smallest demand, $\arg\min_{m \in A_t} d_t[m]$, with ties broken by recency. The agent, therefore, learns when to act, while the model target is chosen by a deterministic rule. Infeasible macro-actions, such as eviction from an empty cache or prefetching when no catchable model can fit, are masked, and the agent idles until the state changes. In the load regime evaluated in this thesis the explicit EVICT action is rarely selected, because eviction occurs implicitly when a prefetch makes room to admit a target (Table 6.2); it is retained in the action set for generality and for regimes in which proactively freeing memory ahead of a large incoming model is beneficial.

4.1.3 Reward

The reward is the negative of the multi-objective cost in (3.6), with a small bonus for serving heavier requests from cache:

$$R_t = \Big(R_{\text{hit}} + w_{\text{c}}\, \hat{c}(m_t)\Big)\, \mathbb{1}[\text{hit}] \;-\; \kappa\, \hat{\ell}(m_t)\, \mathbb{1}[\text{miss}] \;-\; \lambda_{\text{v}}\, u_t \;-\; \lambda_{\text{s}}\, c_t, \tag{4.5}$$

where $\hat{c}(m_t) \in [0, 1]$ is the normalized completion weight of the request, $\hat{\ell}(m_t)$ is normalized size-dependent cold-start latency, u_t is normalized VRAM occupancy, and c_t is the number of load/evict actions at step t. Maximizing discounted return under this reward gives a discounted surrogate for the multi-objective problem in (3.6).

4.2 The Local Edge Agent: Predictive Policy

The policy uses an actor–critic network. A shared multilayer-perceptron backbone feeds a three-way actor head $\pi_\theta(a \mid s)$ and a scalar value head $V_\theta(s)$.

4.2.1 The Exploration Challenge

Prefetching is difficult to learn because the feedback is delayed and asymmetric. A wrong prefetch wastes bandwidth and can evict a soon-needed model, while a correct prefetch is rewarded only when the corresponding queued request is served. If the policy also had to choose among M model targets, most exploratory prefetches would be uninformative or harmful. The macro-action formulation in Section 4.1 avoids this problem by removing target selection from the learned action. The policy only decides whether to prefetch, evict, or idle, which allows reinforcement learning to converge from scratch in our experiments (Chapter 6).

4.2.2 Local Reinforcement Learning

Each node trains locally with Proximal Policy Optimization (PPO) [13]. With Generalized Advantage Estimation advantages $\hat{A}_t$ [14] and probability ratio $\rho_t(\theta) =$

$\pi_\theta(a_t \mid s_t)/\pi_{\theta_{\text{old}}}(a_t \mid s_t)$, PPO minimizes

$$\mathcal{L}(\theta) = -\,\mathbb{E}_t\Big[\min\Big(\rho_t \hat{A}_t,\ \text{clip}(\rho_t, 1{-}\epsilon, 1{+}\epsilon)\hat{A}_t\Big)\Big] + c_v\,\mathcal{L}^V(\theta) - c_e\,\mathcal{H}[\pi_\theta], \tag{4.6}$$

where $\mathcal{L}^V$ is the value loss, $\mathcal{H}$ is the entropy bonus, and c_v, c_e are coefficients.

4.3 Delay-Adaptive Lookahead and Local Updates

Local agents participate in asynchronous federated training rather than training in isolation. When the server dispatches global parameters θ with a version stamp, the client also keeps the previous parameters θ^- it received. Before sampling, the client applies delay-adaptive lookahead [19], extrapolating along the recent global trajectory to compensate for the staleness that may accumulate during local computation:

$$\tilde{\theta} \;=\; \theta + \frac{1-\alpha}{\alpha}\,(\theta - \theta^-). \tag{4.7}$$

The client loads $\tilde{\theta}$ into the network, performs the local PPO update in (4.6), and obtains θ^{loc}. It then returns the pseudo-gradient

$$g \;=\; \tilde{\theta} - \theta^{\text{loc}}, \tag{4.8}$$

which is referenced to the extrapolated starting point $\tilde{\theta}$ rather than to the originally received θ. This reference point matters for stability: g represents only the local PPO improvement, while differencing against θ would mix the extrapolation offset into the reported update and cause the server to apply it again.

4.4 The Asynchronous Federated Aggregator

The server stores the global parameters θ and a version counter v_g. It dispatches (θ, v_g) to clients and applies their pseudo-gradients as soon as they arrive, without waiting for a synchronization barrier. When a client update computed from version v arrives, the server computes its staleness $\sigma = v_g - v$ and applies a staleness-damped step:

$$\theta \leftarrow \theta - \frac{\eta_s}{\sqrt{1+\sigma}}\, g_k, \qquad v_g \leftarrow v_g + 1. \tag{4.9}$$

Because N clients can contribute local steps concurrently, the server learning rate is kept on the order of $1/N$ to avoid an aggregate update that is too large. After applying an update, the server sends the fresh parameters only to the client that just reported, allowing that client to start the next round immediately. This targeted broadcast differs from synchronous schemes that wait for all clients before broadcasting a new global model [15]. The server also periodically evaluates the global policy and keeps the highest-hit-rate snapshot to reduce the effect of transient drift. Algorithm 1 summarizes the procedure.

Algorithm 1 FedCache asynchronous federated training

Server init: $\theta \leftarrow$ random initialization; $v_g \leftarrow 0$; dispatch (θ, v_g) to all N clients

Each client k runs ClientUpdate *(below) asynchronously on its most recent dispatch and reports the pseudo-gradient g_k.*

while $v_g < T_{\max}$ **do**

  receive g_k from client k (dispatched at version v) ▷ from k's ClientUpdate

  $\sigma \leftarrow v_g - v$

  $\theta \leftarrow \theta - \dfrac{\eta_s}{\sqrt{1+\sigma}}\, g_k$; $\quad v_g \leftarrow v_g + 1$

  if periodic **then** update best-hit-rate snapshot

  dispatch (θ, v_g) to client k ▷ targeted broadcast

end while

function ClientUpdate(θ) ▷ θ: latest params dispatched to this client

  $\tilde{\theta} \leftarrow \theta + \frac{1-\alpha}{\alpha}(\theta - \theta^{-})$; $\quad \theta^{-} \leftarrow \theta$

  $\theta^{\text{loc}} \leftarrow \text{PPO}(\tilde{\theta},\ \text{local rollout})$

  return $g = \tilde{\theta} - \theta^{\text{loc}}$

end function

Chapter 5

Experimental Setup and Methodology

This chapter describes the evaluation methodology: the production trace and the trace-driven simulator built on it, the model-size parameterization, the baseline policies, and the metrics used in Chapters 6 and 7. Appendix A records the full configuration behind every reported number, including trace preprocessing, the size-tier assignment, the simulator's event ordering, the network and training budget, and the baseline hyperparameters.

5.1 The Production Trace

We evaluate FedCache with both simulation and real experiments on the cloud using the Alibaba GenTD26 trace [4], a production trace of diffusion image-generation serving. We use the request-level log, in which each request provides an anonymized target model identifier and request features such as prompt length and inference-step count; the identifier sequence defines the request stream ρ from Chapter 3.

We stream the trace in chronological chunks and use the $M = 100$ most active models.

We build a trace-driven discrete-event simulator in which an edge node maintains the available set A_t, the warming set W_t, and the VRAM capacity C. The simulator advances one request per step. A request whose model is in A_t is a hit served at $\ell_{\text{hit}} = 50\,\text{ms}$. A miss incurs a size-scaled cold-start latency in the seconds-to-tens-of-seconds range reported by the trace. One simulation step corresponds to roughly one second of request arrivals, so $\Delta(m)$ equals the model's load time, approximately 5–15 steps. Once a load is issued, the model is retained; if a request arrives while the model is still warming, it waits only for the remaining load time.

FedCache's queue window covers the next H already-admitted requests exactly, not a forecast, so the gains reported here are contingent on a pending queue at least H requests deep. This is a property of the load regime, not of the trace alone: because diffusion inference takes seconds per request while requests can arrive more frequently, a backlog forms whenever the arrival rate exceeds the per-node service rate, and it is in exactly this saturated regime that a deep queue exists and a prefetch has time to complete before its request is served. Under light or fluctuating load the queue is shallow, the queue window carries little information, and prefetching is correspondingly neutral or, when it displaces a soon-needed model, mildly harmful. We therefore scope every claim in this thesis to the sustained-overload regime; the realized queue-depth distribution over a full production deployment, and the light-load behaviour, are measurements we do not make here. The physical replay of Chapter 7 is run at the design request rate for this reason.

The simulator charges size-dependent cold starts in the seconds-to-tens-of-seconds range ($5000 + 770\, s(m)$ ms; Table A.2), following the load-latency magnitudes reported for the production trace [4]. This scale is a parameter of the system model, not a measured property of any one deployment: the commodity edge nodes of Chapter 7, with fast local storage and guest-cold reads, load the same models in 0.1–1.9 s. Two points follow. Every result that depends on the *absolute* scale—absolute latencies, and the hit:miss cost ratio that turns a hit-rate gain into a latency gain—is conditional on this parameter, as this chapter and Chapter 6 already caution. The *relative* ordering of the policies is not: the replay in Chapter 7 re-runs the identical decisions at the true measured load times and FedCache's hit-rate advantage persists and widens, which is the purpose of that validation. The production-scale assumption should thus be read as representative of datacentre-grade storage under load rather than of this particular cluster.

By default, the federation contains $N = 5$ edge nodes. Each node runs the local agent from Chapter 4 on its own request stream, and the asynchronous aggregator runs for $T_{\max} = 1000$ global steps. We use heterogeneous per-client compute delays to induce realistic staleness. In non-IID experiments, each node reads a disjoint shard of the trace. For generalization, we train on an early temporal segment and evaluate on a later held-out segment. Unless otherwise noted, results report the global policy evaluated greedily. Because the request trace is fixed, the environment has no stochastic dynamics, and the deployed policy acts greedily, re-evaluating the deployed policy returns identical metrics; the reported numbers are therefore exactly reproducible, and we report a single deterministic run rather than averaging over evaluation replicas. Table 5.1 lists the main hyperparameters. The PPO

TABLE 5.1: Key training hyperparameters.

Parameter	**Value**
Models M / VRAM capacity C	100 / 15 GB
Queue-window length H	20
Simulation step	1 s ($\approx$1 request)
Load delay $\Delta(m)$	5–15 steps (= load time)
Hit / cold-start latency	50 ms / 5–15 s
Edge nodes N / global steps $T_{\max}$	5 / 1000
Server LR η_s	0.3 (order $1/N$)
Lookahead coefficient α	0.8
Staleness damping	$1/\sqrt{1+\sigma}$
PPO rollout / local epochs	512 / 4
Learning rate / γ / GAE λ	3×10^{-4} / 0.99 / 0.95
PPO clip ϵ / entropy c_e	0.2 / 0.005

settings follow common PPO practice [13, 14]: learning rate 3×10^{-4}, discount factor γ=0.99, GAE λ=0.95, and clipping parameter ϵ=0.2. The federation settings follow the AFedPG design; in particular, η_s is kept on the order of $1/N$ to prevent N concurrent client steps from accumulating into an overly large server update. The queue-window length H=20 is chosen to exceed the maximum load delay, which is the regime needed for catchable prefetches and is supported by the ablation in Figure 6.7(c). That ablation shows hit rate saturating near $H \approx 10$, below the maximum load delay of 15 steps. The two are consistent: only the largest models (full pipelines, 10% of the universe) reach the 15-step delay, while most requests target small adapters and base checkpoints with shorter delays, so a window near 10 already makes the bulk of traffic catchable. We nonetheless set H=20 so that the rare large-delay pipelines remain catchable as well; the ablation confirms this costs no hit rate and only a small increase in thrash.

5.2 Model-Size Parameterization and VRAM Fragmentation

The trace includes request arrivals and component load latencies, but it does not expose the parameter footprint of each anonymized model. Empirically measured representative component footprints were therefore used to parameterize the trace-driven simulation. We measured the VRAM footprints of LoRA adapters, ControlNets, base checkpoints, and full pipelines on the physical cluster of Chapter 7, and used the four measured ranges as size tiers [2, 3]. The ordering of these classes matches the per-class load-latency distributions in the trace [4].

Each of the $M = 100$ trace identifiers is then assigned to a tier, and to a footprint within that tier's measured range, under a fixed seed. The measurements are exact for the components and hardware we ran, but the mapping from anonymized trace identifiers to footprints is synthetic: a different assignment permutes which model receives which size and therefore changes the packing problem. Appendix A.2 documents the tiers, their shares, and the assignment procedure in full.

The resulting sizes span 0.2 to 13.4 GB. Their aggregate footprint is approximately 253 GB against a per-node capacity of $C = 15$ GB, or about 17× oversubscription. Thus, a single full pipeline can nearly fill the cache, while dozens of adapters may fit at once. This heterogeneity creates fragmentation and defines the low-memory regime studied in Chapter 6. Results that depend on absolute packing—the low-memory crossover, the oversubscription ratio, and absolute hit-rate and latency values—should be read as conditional on this parameterization rather than as properties of the trace itself.

5.3 Baselines

We compare FedCache with reactive, learned, and federated baselines, as well as an offline eviction reference. The comparison includes standard cache management and recent approaches relevant to edge GenAI inference. The learned baselines are adapted from other application settings, so they should be read as representative adapted heuristics rather than as tuned state-of-the-art competitors; Appendix A.5 gives their full configurations and states which information each one observes.

- **LRU**: the standard reactive policy. On a miss, it loads the requested model and evicts the least-recently-used models until there is enough space.

- **IceBreaker** [28]: proactive pre-warming based on the dominant period in each model's invocation history, computed through FFT. We adapt it to a single GPU tier and omit its heterogeneous-server cost optimization.

- **XGBoost** [46]: a gradient-boosted next-request predictor that prefetches the highest-scored non-resident model.

- **ACPM** [22]: an adaptive container-provisioning method whose LSTM [47] forecasts near-future request volume from binned utilization history. We adapt its container-provisioning decision to model residency under a fixed VRAM capacity.

- **Synchronous FedAvg** [15]: the same local RL agent trained with synchronous federated averaging, which isolates the effect of asynchronous aggregation.

- **Belady-OPT** [38]: the offline eviction reference. It sees the entire future request sequence, evicts the resident model whose next reference is furthest away, and is granted instantaneous loads. It is optimal among load-on-miss eviction policies, but it cannot prefetch, so it occupies a different information setting from FedCache and is not a general offline optimum for the problem of Chapter 3 (Section 6.3).

We note one asymmetry explicitly. FedCache consumes the pending-request queue, which is intrinsic to its formulation, whereas the learned baselines predict from invocation history in their original designs and are evaluated in that designed information setting. Granting them the same queue would turn them into variants of FedCache's deterministic resolver rather than into their intended methods, so we keep each baseline in its native setting and read the resulting ordering as a comparison of designs under their own assumptions, not as a claim that these methods are inferior given identical inputs.

5.4 Evaluation Metrics

We report a multi-objective profile rather than a single aggregate score:

- **Hit rate**: the fraction of requests served from the available set.
- **Average latency** (ms): mean per-request latency. Because a miss costs roughly 100–300× a hit, this metric is dominated by cold starts.
- **Average VRAM**: mean occupancy fraction u_t, which measures memory pressure.

- **Thrash**: the total number of load and evict actions, measuring prefetch churn and bandwidth cost. This is a coarse aggregate; because it combines loads with evictions and counts only explicitly issued actions, we decompose it into demand loads, prefetch loads, evictions, and wasted prefetches in Section 6.1.1, which is the more informative view of switching cost.

For federated experiments, we also report wall-clock convergence in simulated time. Asynchrony primarily improves the time needed to reach a target hit rate, rather than changing the final hit rate.

The per-capacity sweeps in Figures 6.1–6.2 and the generalization result in Figure 6.3 include all baselines. Consistent with Table 6.1, the learned methods (XGBoost, ACPM, and IceBreaker) match the performance of LRU—a standard reactive policy in production caching—once capacity is ample. However, under tight memory constraints, these predictive models do not surpass LRU's reactive eviction in our setting, which illustrates the difficulty of balancing predictive prefetching against VRAM fragmentation. Meanwhile, FedCache separates above the low-memory crossover. Because the baselines were adapted rather than tuned, and because they observe only request history while FedCache also observes the pending queue (Appendix A.5), this ordering should be read as evidence about these adapted configurations rather than as a general ranking of the underlying methods.

Chapter 6

Results and Evaluation

We evaluate the greedy global FedCache policy against the baselines from Chapter 5. Unless stated otherwise, results have been obtained using a cluster of $N = 5$ nodes, with VRAM capacity of $C = 15\,\text{GB}$ each, and the default hyperparameters in Table 5.1. All values in this chapter come from the trace-driven, queue-aware simulator and are conditional on that configuration and on the model-size parameterization of Chapter 5. FedCache's absolute figures should be read as achievable by a learnable queue-aware policy, not as near-optimal; the distance to the best queue-aware policy is not bounded here (Section 6.3).

6.1 End-to-End Comparison

Table 6.1 summarizes the multi-objective comparison. Among the policies compared, FedCache obtains the highest hit rate (94.78% compared with 90.48% for LRU), the lowest average latency (411 ms compared with 706 ms, a reduction of about 42%), and the numerically lowest thrash (454 versus 476 for LRU). This last figure needs care and is decomposed in Section 6.1.1: the thrash metric counts actions of

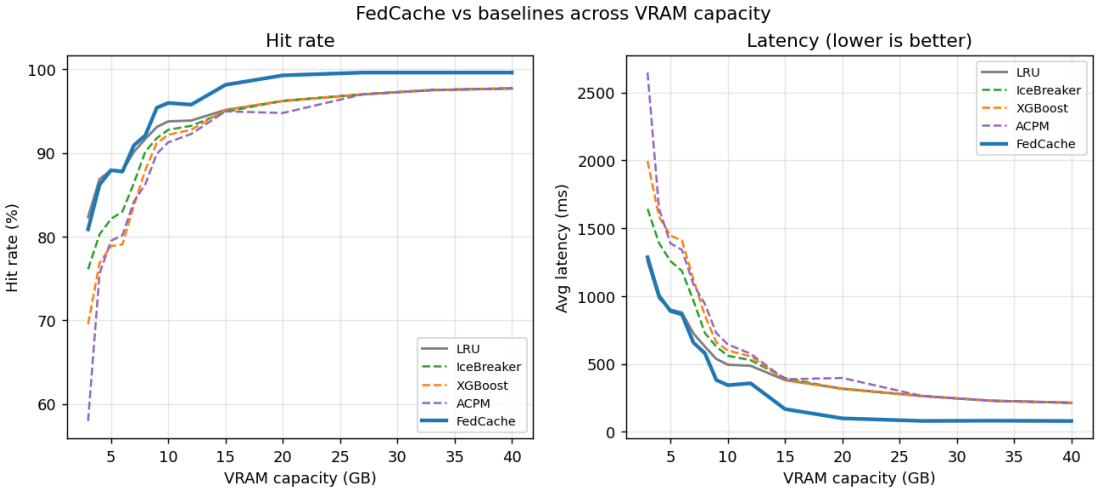

FIGURE 6.1: Hit rate (left) and average latency (right) versus per-node VRAM capacity, for FedCache and all baselines. FedCache is trained once at C=15 GB and evaluated across the range without retraining, so this is a capacity-transfer (robustness) curve rather than a per-capacity optimum. The learned baselines (IceBreaker, XGBoost, ACPM) remain at or below LRU; under tight memory (the thrash regime) FedCache slightly trails LRU while staying above the learned baselines, and it outperforms all baselines above the $\sim$7 GB crossover.

different kinds for different policies, so the raw 454-versus-476 comparison should not be read as FedCache simply churning less. The demand-aware admission gate is important here: it allows FedCache to prefetch when doing so is useful, while avoiding the extra churn that would otherwise make prediction costly.

These end-to-end results have been obtained by a 5000-step evaluation that extends into the more difficult later segment of the trace. The per-configuration analyses in Figures 6.1–6.3 use a shorter 3000-step window from the earlier, more stationary part of the trace; this explains their higher absolute hit rates, around 98% at the same capacity. The training-time curves (Figures 6.4 and 6.5) evaluate on similar early-trace windows, which is why their LRU reference sits above the value

TABLE 6.1: End-to-end multi-objective comparison on GenTD26 (N=5, C=15 GB, 5000-step evaluation); a single deterministic run, identical across evaluation replicas. Lower latency, VRAM, and thrash are better; best per column in bold.

	Hit %	Latency (ms)	VRAM %	Thrash
LRU	90.48	705.6	81.1	476
XGBoost	90.34	713.8	81.2	483
ACPM (LSTM)	89.68	739.7	**80.8**	621
IceBreaker	89.10	793.6	82.0	637
FedCache	**94.78**	**411.4**	83.9	**454**

in Table 6.1. The relative ordering is unchanged. The learned baselines do attempt to prefetch: IceBreaker uses periodicity, whereas XGBoost and ACPM use learned demand prediction. While these adapted baselines establish a strong predictive floor that remains competitive with standard LRU, the specific latency demands of GenAI serving expose their limitations. In this workload, their predictions are not sufficient to stage models within the multi-second load window. Their prefetches, therefore, often arrive too late or add churn without converting misses. FedCache's main cost is a small increase in average VRAM occupancy (83.9% vs. 81.1% for LRU). Since cold starts dominate latency, the observed latency reduction is mainly a consequence of the higher hit rate. Reducing these multi-second events should also improve the latency tail targeted by edge-serving SLAs.

6.1.1 Decomposing the Switching Cost

The single thrash figure in Table 6.1 combines loads and evictions and counts only the action a policy issues explicitly. Table 6.2 decomposes it at the same operating point, separating demand-triggered loads (whose target is the current request) from speculative prefetch loads (whose target is a later request in the

queue), reporting the evictions each policy actually performs, and counting wasted prefetches — models staged but evicted before they serve any request.

Three points follow, and each qualifies the raw thrash comparison. First, the headline figures count actions of different kinds. LRU's 476 is entirely reactive demand loads that a load-on-miss policy cannot avoid, whereas 417 of FedCache's 454 are speculative prefetches; the totals are close but not comparable, so the ordering does not by itself show that FedCache churns less. Second, the metric omits the evictions each policy forces to make room. No policy in this comparison issues an explicit eviction action; evictions occur implicitly when a load displaces a resident model, and the Table 6.1 metric does not count them. Counted, FedCache performs 434 evictions against LRU's 467, so its total cache mutations (876 byte-moving loads and evictions) sit slightly below LRU's (943) while its hit rate is 4.3 points higher; this is the more defensible form of the switching-cost claim. Third, wasted prefetches separate accurate prediction from indiscriminate loading: FedCache wastes 49 of its 417 prefetches (11.8%), whereas the forecasting baselines waste far more (78.6% for IceBreaker, 45.3% for ACPM), which is why their prediction adds churn without a matching hit-rate gain.

6.2 VRAM Contention and the Low-Memory Crossover

Figure 6.1 sweeps the VRAM capacity to measure hit rate and latency. The same policy, trained once at the default C=15 GB, is evaluated across the full capacity range; we do not retrain at each point. This is therefore an out-of-distribution

TABLE 6.2: Thrash decomposed (N=5, C=15 GB, 5000-step evaluation). Demand and prefetch are byte-moving loads whose target is the current or a future request; evictions are those actually performed (all implicit here, to make room for a load); wasted counts prefetches evicted before serving a request. The final column is the load-plus-explicit-eviction "thrash" of Table 6.1, which counts only explicit actions and so reduces to the load count.

	Demand	Prefetch	Evictions	Wasted	Thrash (Table 6.1)
LRU	476	0	467	0	476
XGBoost	475	2	468	0	483
ACPM (LSTM)	448	117	556	53	621
IceBreaker	515	103	611	81	637
FedCache	25	417	434	49	454

capacity-transfer experiment: it measures how a single fixed policy degrades as the operating capacity moves away from the one it was trained at, not the best policy achievable at each capacity. A policy retrained per capacity could do better, particularly at the extremes of the range; the curve should accordingly be read as a robustness result rather than a per-capacity optimum. At very low memory (C=3 GB), FedCache slightly trails LRU (80.9% vs. 82.3%). With so little resident memory, even gated prefetching can sometimes displace a model needed soon, although the gate keeps churn limited (321 load/evict actions rather than the much larger churn of an ungated prefetcher).

FedCache and LRU remain close until roughly $C \approx 6$ GB. Above a crossover near 7 GB, FedCache improves both hit rate and latency, reaching 99.6% hit rate and 80 ms at 40 GB compared with LRU's 97.7% and 215 ms. The latency gap widens as VRAM increases because FedCache can convert more first-touch requests into hits. This identifies FedCache's main operating regime: proactive prefetching is most beneficial once there is enough headroom to stage models without repeatedly

evicting the working set.

6.3 Comparison with an Eviction-Only Belady Reference

Figure 6.2 places FedCache alongside Belady's offline eviction optimum. Belady's algorithm is optimal within a specific setting: it minimizes misses for a *load-on-miss* cache once the request sequence is fixed, by evicting the resident object whose next reference is furthest in the future. It is the natural reference point for eviction, and it is the strongest such reference available.

It is not, however, a general optimum for the problem studied here, and the comparison below should not be read as FedCache surpassing an offline bound. The two settings differ in *both* directions, so neither bounds the other:

- The Belady reference is granted advantages FedCache does not have. It is given the entire future request sequence for its eviction decisions, not a window of H; and its loads are *instantaneous*, so it never pays the size-dependent delay $\Delta(m)$ that governs every other policy in this thesis. In the load-delay model of Chapter 3, this is a substantial concession.

- FedCache is granted an advantage the Belady reference does not have. It observes the H-step pending queue and may act on it by prefetching, so a model can be resident before its first reference. A load-on-miss policy, however well it evicts, must miss that first reference by construction.

What Figure 6.2 therefore measures is a trade between these two concessions: the capacity at which the ability to *act early* becomes worth more than the ability to *load instantly*. It does not establish an ordering between the methods, and it says nothing about how FedCache would compare against an optimal policy in its own setting.

Read that way, FedCache attains a higher hit rate than the eviction-only reference at every capacity above a crossover near 8.5 GB: 98.1% versus 96.5% at C=15 GB, and 99.6% versus 98.5% at 40 GB. Below the crossover the two are close; FedCache leads at the tightest capacity, and the Belady reference is ahead by at most 0.5 percentage points in a narrow 6–8 GB range. The crossover is where prefetching first-touch requests begins to recover more misses than instantaneous loading does, which is consistent with the headroom argument of the previous section: staging a model early only helps once there is room to hold it without displacing the working set.

The comparison that would actually bound FedCache is a proactive oracle in a matched information setting—one that sees the same H-step queue, pays the same load delay $\Delta(m)$, respects the same capacity C, and chooses prefetches and evictions optimally over the horizon. That reference is not constructed here, so the distance between FedCache and the best achievable queue-aware policy remains unquantified. It is the most informative missing comparison in this chapter, and we identify it as future work in Chapter 8.

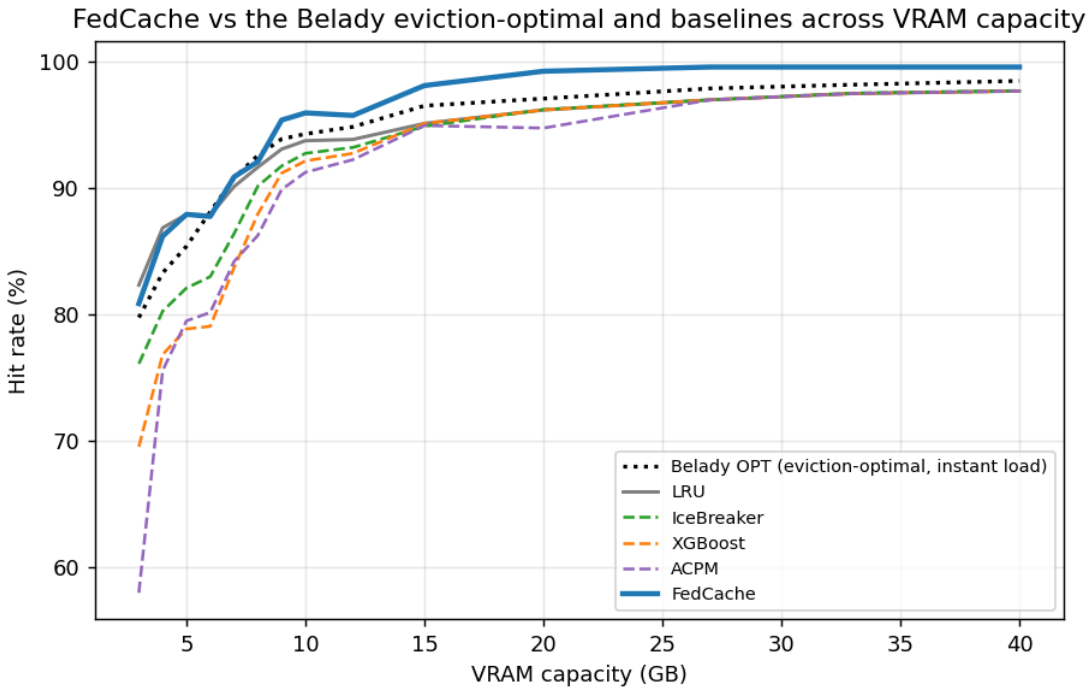

FIGURE 6.2: Hit rate of FedCache, the learned baselines (IceBreaker, XGBoost, ACPM), and LRU against Belady's offline eviction-optimal. *The two settings are not comparable and neither bounds the other*: the Belady reference sees the whole future and loads instantaneously but is eviction-only, whereas FedCache sees an H-step queue and pays the real load delay $\Delta(m)$ but may prefetch. Above a $\sim$8.5 GB crossover FedCache attains the higher hit rate, by staging first-touch requests that a load-on-miss policy must miss; this is not evidence of surpassing an offline optimum. The baselines remain at or below LRU.

6.4 Sensitivity to the Model-Size Mapping

Because the trace does not expose per-model footprints, sizes are assigned synthetically (Chapter 5, Appendix A.2); this is the largest modelling assumption in the evaluation, and results that depend on absolute packing inherit it. To test whether the conclusions depend on the particular draw, we regenerate the mapping under six configurations — the paper draw (seed 42), three reseeds of the same

measured tier distribution, and two altered distributions (a heavier tail with more large pipelines, and a lighter tail with more small adapters) — and for each we retrain FedCache from scratch under the same budget and training seed, so the size mapping is the only factor that changes. Table 6.3 reports hit rate at C=15 GB together with the FedCache-versus-LRU low-memory crossover.

Three conclusions are robust to the mapping. First, across the four draws of the measured tier distribution (seed 42 and the three reseeds), the FedCache advantage over LRU is stable at +2.6 to +3.1 percentage points, even though oversubscription ranges from 16.9× to 22.1× and absolute hit rates move by about three points; the margin is a property of the method, not of the particular draw. Second, FedCache exceeds all three learned baselines in every configuration without exception, including the adversarial heavy-tailed one. Third, the low-memory crossover is present in every configuration, but its location scales with oversubscription, from 3 GB for the lightly loaded mapping (10.7×) to 20 GB for the heavy-tailed one (28.4×). The specific $\sim$7 GB crossover reported earlier is therefore parameterization-dependent, as Chapter 5 cautions, while the qualitative structure — trailing LRU under tight memory and leading once there is headroom — is invariant.

The single configuration in which FedCache trails LRU at C=15 is the heavy-tailed distribution (−1.2 points). This is consistent rather than contradictory: with a quarter of the models drawn from the 8–14 GB pipeline tier, the workload is 28.4× oversubscribed and its crossover moves to 20 GB, so C=15 sits in the tight-memory regime below the crossover — exactly where Figure 6.1 already shows gated prefetching can trail reactive eviction. Even there, FedCache stays above

TABLE 6.3: Sensitivity to the synthetic model-size mapping (C=15 GB, 3000-step evaluation, FedCache retrained from scratch under each mapping). Each row is a different assignment of the 100 models to footprints; "best learned" is the strongest of IceBreaker, XGBoost, and ACPM. The crossover is the smallest capacity at or above which FedCache stays $\geq$ LRU. FedCache leads every learned baseline in all six mappings; it trails LRU only in the heavy-tailed mapping, whose crossover (20 GB) places C=15 in the tight-memory regime.

Mapping	Oversub.	FedCache	LRU	Best learned	ΔLRU	Crossover
Paper (seed 42)	16.9×	98.1	95.1	95.1	+3.0	5 GB
Reseed 0	21.6×	95.5	92.6	91.4	+2.9	9 GB
Reseed 1	17.6×	96.5	93.9	94.0	+2.6	7 GB
Reseed 2	22.1×	97.1	94.0	93.8	+3.1	7 GB
Heavy tail	28.4×	89.5	90.7	88.9	−1.2	20 GB
Light tail	10.7×	99.5	97.3	97.3	+2.2	3 GB

every learned baseline. The analysis thus supports the qualitative claims while confirming that the crossover capacity and the absolute hit-rate and latency values should be read as conditional on the size parameterization.

6.5 Generalization to Held-Out Workloads

To test whether FedCache learns transferable structure rather than memorizing the training stream, we train on an early temporal segment and evaluate on a later held-out segment (Figure 6.3). All policies drop on the held-out period, indicating that the later segment is harder, likely because it contains new adapters or checkpoints that were not present in the earlier phase. FedCache nevertheless keeps its advantage under this distribution shift, and the margin increases: +3.0 percentage points on the training split (98.1% vs. 95.1%) and +5.3 percentage points on the test split (83.7% vs. 78.4%).

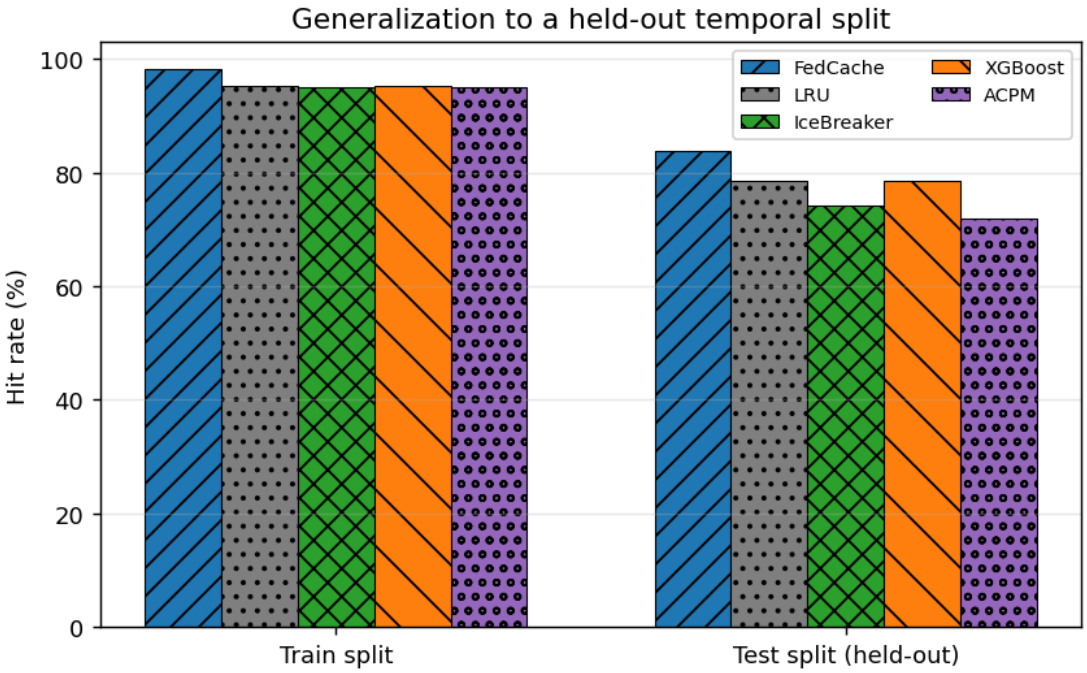

FIGURE 6.3: Hit rate on the training split versus a held-out later temporal split, for FedCache and all baselines. FedCache maintains a larger margin over LRU and the learned baselines on unseen traffic.

6.6 Asynchronous Scalability and Straggler Tolerance

Figure 6.4 reports the wall-clock time needed to reach 95% hit rate as the number of edge nodes increases. The number of global updates required to reach the final hit rate remains roughly constant, while wall-clock time decreases because more clients contribute updates in parallel. The time to reach 95% hit rate drops from 10.0 at N=1 to 5.2, 3.1, 2.5, and 2.3 at N=2, 5, 10, 20, respectively. The scaling is close to linear for small N and then saturates as the fixed per-update compute floor becomes the bottleneck.

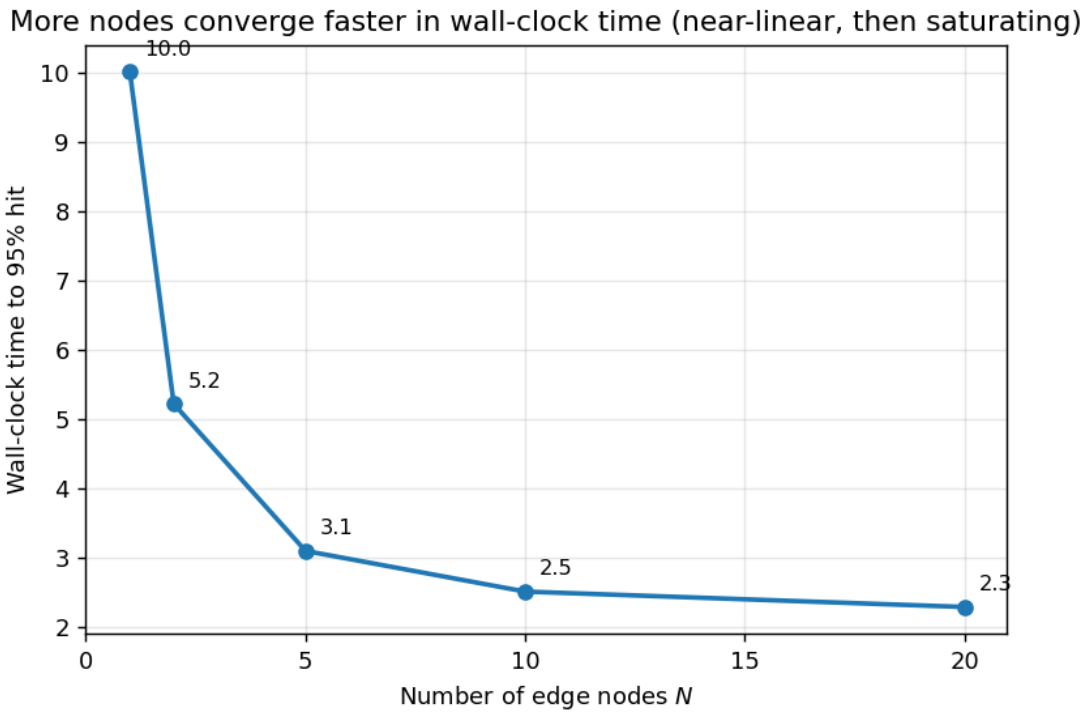

FIGURE 6.4: Wall-clock time (in simulator time units) to reach 95% hit rate versus the number of nodes N. Convergence time falls steeply and then saturates. Scaling is near-linear for small N before flattening as the per-update compute floor dominates.

We also compare the asynchronous aggregator with a synchronous FedAvg variant under heterogeneous node speeds, with per-update delays drawn from $[1, 20]$ in simulator time units. Synchronous training can advance only when the slowest worker in a round has finished. It reaches a steady-state hit rate of 97.3% at wall-clock time $t{\approx}33$, compared with $t{\approx}15$ for the asynchronous aggregator. The resulting 2.1× wall-clock speedup comes from applying updates as they arrive rather than waiting at a global barrier.

6.7 Training-Scheme Comparison

The preceding sections evaluate asynchronous federated training but do not isolate what federating contributes relative to not federating at all. We therefore compare five training schemes, matched to the same number of local PPO updates (150) so that they differ only in data access and aggregation, and all evaluated on the same full request stream. Table 6.4 reports the result.

For policy quality the outcome is uniformly negative: every scheme reaches the same final hit rate of 98.13%, and all five reach the 95% target after the same number of updates (10). Local-only training on a single shard covering 29% of the trace matches centralized training on the pooled trace, and both match synchronous and asynchronous federation.

This is not an artifact of the schemes collapsing onto one model. Two independently trained policies, local-only and asynchronous, differ in their parameters (ℓ_2 distance 1.52 over $\sim$169,000 weights) yet produce the same greedy hit rate. The macro-action policy saturates in *behaviour* rather than in weights: because the deterministic resolver and admission gate fix the concrete target and infeasible macro-actions are masked, the learned component only decides *when* to act, and that decision converges quickly and from little data. The same ceiling is reached by the 1000-step deployed checkpoint and by every retrained policy in Section 6.4. It is also consistent with the learning curve of Figure 6.5, where an untrained policy is already competitive.

What federation does buy is wall-clock convergence. A single node needs 36.7 simulated time units to reach 95%, whereas five nodes reach it in 10.1 (synchronous)

and 9.1 (asynchronous): the large factor is parallelism, and the smaller remaining one is removal of the round barrier. Communication volume, however, is identical for the two federated schemes (193 MB for 150 updates of a 660 KB policy), because each local update costs one upload and one download either way. Asynchrony therefore improves latency, not bandwidth. Centralized training needs no model traffic at all, but requires the raw traces to be pooled — precisely the cost the federated setting exists to avoid.

Finally, the IID and non-IID asynchronous runs are indistinguishable. This should not be read as robustness to heterogeneity. The non-IID partition assigns disjoint chunks of a single short trace (7 chunks over 5 nodes, an uneven 2/2/1/1/1 split), so the shards differ in which requests they sample but not in tenant population; such a partition does not represent geographically distinct sites with distinct model mixes, and testing that would require multiple traces.

These results support a narrower claim than federated learning is often given. On this workload, asynchronous federated training is justified as a way to reach the policy *faster* without centralizing request traces — not as a way to obtain a *better* policy, since the policy is the same however it is trained. We report it on that basis.

TABLE 6.4: Training-scheme comparison, budget-matched at 150 local PPO updates (N=5, C=15 GB, policy 660 KB). All schemes are evaluated on the same full request stream; only training-time data access and aggregation differ. Wall-clock is simulated time to reach a 95% hit rate. Final quality is identical throughout; federation buys convergence time, and asynchrony buys a further reduction over the synchronous barrier at equal communication volume. Centralized training moves no model weights, but requires the raw request traces to be pooled, which is the cost federation exists to avoid.

Scheme	Training data	Final hit %	Time to 95%	Comm. (MB)
Local-only	own 1/5 shard	98.13	36.7	0
Centralized	pooled full trace	98.13	36.7	0 (raw traces)
Sync FedAvg (non-IID)	disjoint 1/5 shards	98.13	10.1	193
Async AFedPG (non-IID)	disjoint 1/5 shards	98.13	**9.1**	193
Async AFedPG (IID)	full trace, all nodes	98.13	**9.1**	193

6.8 Ablation Studies

6.8.1 Learning from Scratch

Figure 6.5 shows that, with the macro-action formulation, reinforcement learning reaches its final hit rate from a random initialization and without expert demonstrations. The policy converges within roughly 10–20 global steps. The key factor is the reduction of the action space: once the target model is selected deterministically, the agent only needs to learn when prefetching or eviction is useful. This also explains why even the untrained policy is competitive at step 0. The deterministic target resolution and admission gate already encode the domain structure, while training refines *when* to act and closes the remaining gap to the $\sim 98\%$ ceiling. The policy quickly learns to stage the imminent working set and remain idle when no beneficial action is available, thereby keeping the cold-start rate low.

Two properties of the evaluated regime bound how much the learned component

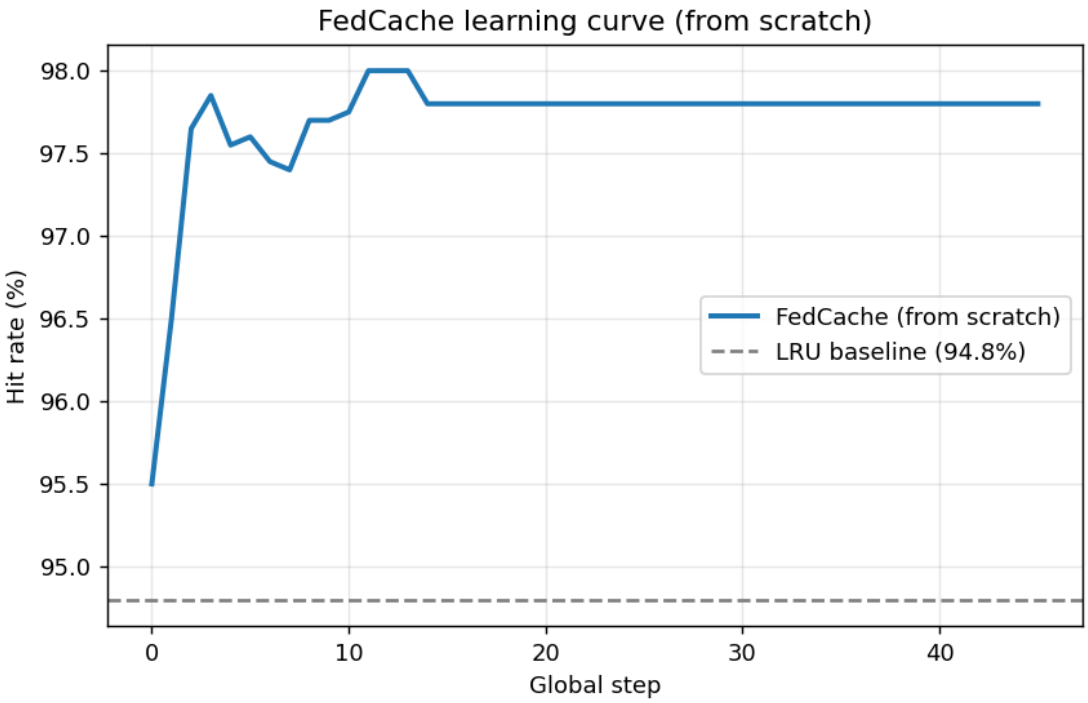

FIGURE 6.5: From-scratch FedCache learning curve. With no expert demonstrations, the macro-action policy converges to a $\sim 98\%$ steady-state hit rate within ~ 10–20 global updates, well above the LRU reference.

can add here, so the from-scratch result is not over-read: the fixed request rate keeps the pending queue deep—the regime in which a near-constant prefetching rate is closest to optimal—and the deterministic resolver and admission gate already encode most of the domain structure. The claim supported by this section is therefore that the macro-action formulation makes the policy learnable and stable from a random start, not that learning is the dominant source of the serving gains; the regimes in which a learned policy should separate more clearly (variable or light load, and workload drift) are discussed in Chapter 8.

6.8.2 Demand-Aware Admission Gate

The demand-aware gate determines whether prediction becomes useful prefetching or excessive churn. Figure 6.6 replays the deployed policy with the gate disabled, so every catchable prefetch is admitted regardless of the demand for the model it would displace. Without the gate, load/evict actions more than double (984 vs. 454), and the hit rate falls to 85.8%, below LRU's 90.5%. With the gate, the same policy reaches 94.8% hit rate while keeping thrash below LRU.

The switching-cost decomposition of Section 6.1.1 makes the mechanism precise. Disabling the gate raises prefetch loads from 417 to 775, but the extra loads are mostly speculative noise: wasted prefetches rise from 49 to 337, so the fraction of prefetches evicted before use climbs from 11.8% to 43.5%, and the make-room evictions they force rise from 434 to 843. The gate's specific function is thus to suppress speculative loads that would be displaced before serving a request; it converts aggressive prefetching from a source of wasted churn into the low-thrash, high-hit-rate operating point reported in Table 6.1.

6.8.3 Robustness to Federation Hyperparameters

FedCache is not highly sensitive to the asynchronous federation hyperparameters. Varying the delay-adaptive lookahead coefficient from α=1.0 (no lookahead) to 0.2 does not materially change convergence. Similarly, enabling or disabling the $1/\sqrt{1+\sigma}$ staleness damping has little effect under the induced staleness in our experiments. The deployed policy also remains near 97% hit rate across two orders of magnitude in server learning rate, $\eta_s \in [0.1, 30]$, both with and without damping. This stability comes from two design choices: each client returns a

single clipped PPO step, and the server keeps the highest-hit-rate checkpoint to absorb transient drift. That these mechanisms are near-inert here is expected rather than contradictory: the induced staleness in the default configuration is mild (N=5, client delays drawn from $[1, 6]$), so there is little for delay-adaptive lookahead or staleness damping to correct. They are retained because staleness grows with the number of concurrent clients and the spread of their delays; the asynchronous-versus-synchronous comparison (delays widened to $[1, 20]$) is the regime in which asynchrony's benefit becomes visible, and heavier fleets would stress these corrections further.

6.8.4 Proximal regularization

We sweep the FedProx coefficient μ under non-IID clients in Figure 6.7(a). Small values of μ have little effect because the macro policy is largely identity-invariant across clients. Large values ($\mu \gtrsim 50$) over-constrain local learning and slow convergence. In this setting, FedCache does not require a proximal anchor because AFedPG's corrections already mitigate client drift.

6.8.5 Reward shaping

Scaling the hit and miss rewards does not change the optimal policy, so we vary the per-prefetch cost in Figure 6.7(b). Performance is flat across the tested range. Because the demand-aware gate already restricts prefetching to beneficial admissions, increasing the nominal prefetch cost neither changes hit rate nor causes the policy to collapse to idling.

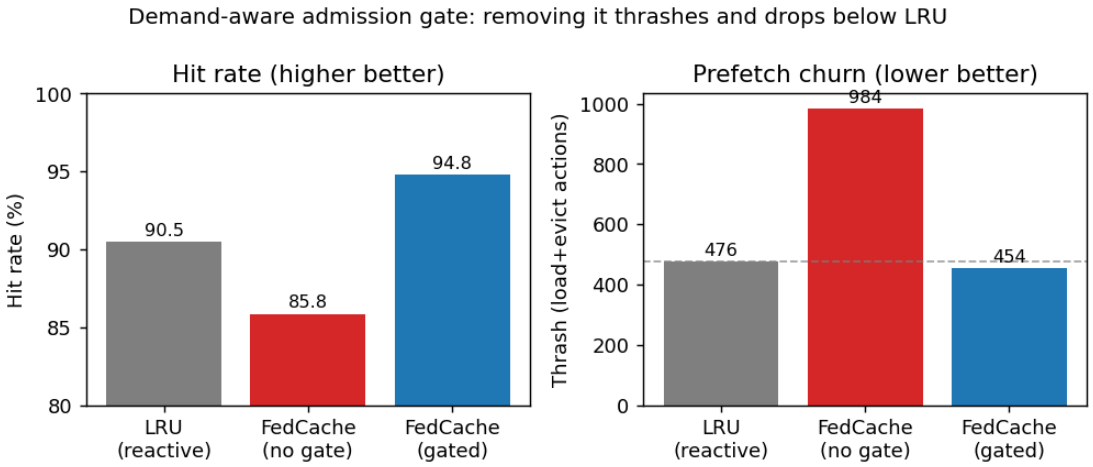

FIGURE 6.6: Demand-aware admission gate ablation (5000-step evaluation). Disabling the gate more than doubles prefetch churn ($454 \rightarrow 984$) and drops the hit rate below LRU; the gate secures FedCache's high hit rate at thrash below LRU.

6.8.6 Queue window

The queue window must be long enough to cover the load delay. With $\Delta(m)$ between 5 and 15 steps, windows of H=3–5 provide no catchable prefetch targets and the policy achieves nearly 0% hit rate. As H exceeds the load delay, performance rises quickly: about 35% at H=6, about 95% at H=8, and a $\sim$98% hit rate by $H \approx 10$ (Figure 6.7(c)). Saturation at $H \approx 10$, below the 15-step maximum load delay, reflects the size distribution: only the largest pipelines reach a 15-step delay, so once the window covers the shorter delays of the many small adapters and base checkpoints, the bulk of catchable traffic is already served. Beyond the load-delay range, hit rate remains flat while thrash increases with H, so a window slightly beyond the maximum load time ($H \approx$ 10–20) is preferable; the deployed H=20 sits in this range and keeps even the largest-delay pipelines catchable.

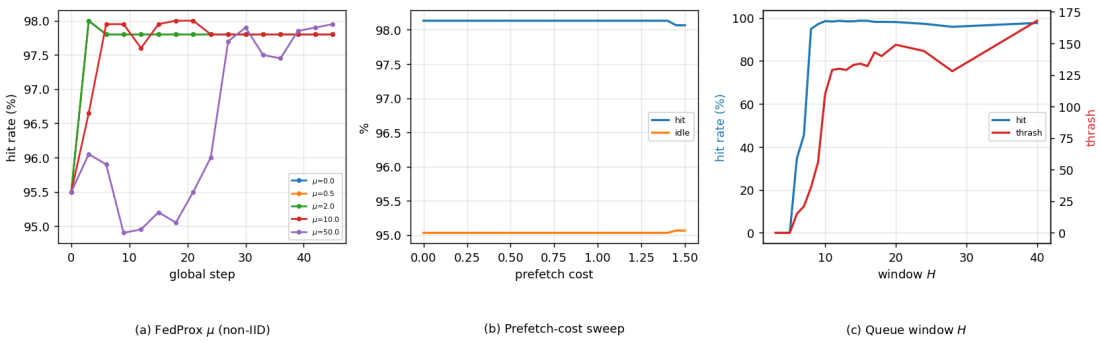

FIGURE 6.7: Ablations. (a) FedProx μ under non-IID clients: large μ over-regularizes; small μ has no effect. (b) Prefetch-cost sweep: performance remains flat because the demand-aware gate keeps prefetching beneficial. (c) Queue window H: hit is $\sim$0 below the load delay, climbs sharply as H clears it, then is flat while thrash grows.

Chapter 7

Deployment on Real Edge GPU Clusters

The simulation results use a discrete-event model in which loads have a fixed delay and do not contend for a real storage-to-GPU path. To test whether the predictive advantage remains under real byte movement, we replay the same policy decisions on a physical cluster. This chapter therefore serves as an experimental validation of the trace-driven simulation: if the simulator's rankings were artifacts of its contention-free load model, they would not survive measured, serialized IO on real GPUs.

7.1 Deployment Setup

We deploy on a **k3s** cluster with four primary nodes. Each node has an NVIDIA H100 GPU partitioned with Multi-Instance GPU (MIG) [48] using the `1g.24gb` profile, giving 24 GB of usable GPU memory. A replay harness applies each policy's decisions to real device memory. Each load is a cold disk read, with the OS page

TABLE 7.1: Experimental evaluation on the four-node matched (fast) H100 cluster (mean of four MIG slices, 5000 requests). Simulation hit rates (Table 6.1) are shown for reference; experimental latency is reported with the demand-priority IO scheduler off and on.

	Sim hit %	Exp. hit %	Exp. lat off	Exp. lat on
LRU	90.48	86.75	371	377
XGBoost	90.34	86.72	373	377
ACPM (LSTM)	89.68	86.67	379	376
IceBreaker	89.10	85.31	441	430
FedCache	**94.78**	**93.83**	412	**225**

cache evicted, staged through a pinned host buffer, and copied over PCIe into VRAM. All loads on a node serialize on the single disk→PCIe path, a physical constraint not captured by the constant-Δ simulator. The four primary nodes have matched storage configurations, and their Tier-0 load-time calibrations agree within a few percent. To bound behaviour on under-provisioned storage, we additionally retain two slower nodes from a heterogeneous configuration whose disks are about 25–30% slower. We run all five policies for 5000 requests using the same deterministic replay configuration and report the mean over the four matched MIG slices using the same evaluators as in simulation.

We also evaluate a demand-priority IO scheduler, which is separate from the policy's admission gate. If a request is waiting for a load, that demand load is prioritized ahead of queued speculative prefetches; similarly, a request for a warming model expedites that load. The scheduler does not drop any loads, so the prefetch hit rate is preserved.

7.2 Experimental Evaluation

On real GPUs, FedCache reaches about 93.8% hit rate (four-node mean), while the other policies are around 85–87% (Table 7.1). This +7-percentage-point margin is wider than the +4.3-percentage-point gap predicted by simulation and is consistent across all four nodes (hit rates spread by ≤ 0.4 percentage points), with or without the IO scheduler. The simulator overestimates hit rate for the reactive and learned baselines by about 3–4 percentage points because it gives credit for loads without modelling congestion, whereas FedCache's simulated hit rate transfers nearly unchanged (a drop of about 1 point). FedCache therefore remains the strongest policy in this replay, and its relative advantage increases. The other predictive baselines (XGBoost, ACPM, and IceBreaker) do not outperform LRU in the experimental evaluation, suggesting that not all forms of prefetching remain useful when loads take real time.

7.3 Latency with the Demand-Priority Scheduler

Without the scheduler, FedCache's denser prefetch IO shares the single disk→PCIe path with demand loads, so its mean latency (412 ms) is slightly above LRU (371 ms). The real experiments expose a prefetch-IO cost that the simulator does not model. The demand-priority scheduler removes this penalty on the matched fast fleet. It lowers FedCache latency from 412 to 225 ms (about −45%), up to 40% below the baselines (LRU: 377 ms), so on this matched fast fleet FedCache then leads every baseline on hit rate *and* latency at once; it also removes node-to-node variance, bringing FedCache to 220–232 ms on all four nodes (Table 7.2). It has little effect on the baseline policies, which issue fewer loads and do not substantially

TABLE 7.2: Per-node FedCache latency (ms), demand-priority IO scheduler off vs. on. On the four matched fast nodes, the scheduler drops FedCache below LRU and removes node-to-node variance; on the two slow nodes (kept for contrast), it increases latency because the IO path is already saturated.

Node	Disk	LRU	FedCache off	FedCache on
afrl-1	fast	400	532	**232**
worker-1	fast	395	490	**224**
worker-2	fast	344	315	**224**
worker-3	fast	344	311	**220**
slow-1	slow	702	827	1387
slow-2	slow	640	565	1224

congest the IO path. The gain is gated on spare IO bandwidth; however, on the two slow, IO-saturated nodes, the same scheduler increases latency ($827 \rightarrow 1387$ and $565 \rightarrow 1224$ ms), because prioritizing demand loads only reorders a backlog that cannot drain.

This distinction matters for interpreting the trace. GenTD26 was collected from Alibaba's production GenAI serving infrastructure, where datacentre-grade storage is not represented by the slow disk bottleneck on our commodity edge nodes. The matched fast nodes are closer to the trace's native environment and show that FedCache improves both hit rate and latency there, while the two slow nodes are bound by the behaviour of under-provisioned edge hardware.

7.4 Validity Bounds

Two caveats qualify the experimental latency results. First, the replay emulates the byte movement of model loads but omits multi-second diffusion inference on both hit and miss paths. Adding full inference would compress relative latency

differences, so the most robust experimental claim is a higher hit rate rather than lower end-to-end latency. Second, the benefit of prediction depends on the load rate. It is most useful under sustained overload, when the pending queue is at least as deep as the queue-window length, and may be neutral or harmful at light load. The single-GPU-per-node result holds at the design request rate, while federated training across more than four physical nodes and high-rate multi-GPU settings remain untested on this cluster. The latency results are also measured in replay rather than in a live serving system, and on four nodes measured once each.

Taken together, the defensible summary of this chapter is the following. Fed-Cache consistently improves cache-hit rate in the replay deployment. Its latency improvement is conditional on sufficient storage and IO headroom and on demand-priority IO scheduling, and it does not hold on nodes whose IO path is already saturated. The demand-priority scheduler is therefore not an incidental implementation detail: on this hardware it is the component that makes the latency benefit materialize at all, and it should be treated as part of the deployed system rather than as a tuning option. Appendix A.7 specifies its behaviour, including the fact that it reorders only queued loads and never cancels a speculative prefetch.

Chapter 8

Discussion and Conclusion

This chapter concludes the thesis. It first summarizes what was built and found, then reflects on where FedCache's gains come from and which design choices matter most, states the assumptions and threats that bound the conclusions of this study, and finally outlines directions for future research.

8.1 Summary

We presented FedCache, a system for reducing GenAI cold starts on edge GPU clusters by learning to prefetch models into VRAM before they are requested. The work is based on the observation that reactive replacement cannot hide the cold-start latency of the request that reveals a miss. FedCache therefore treats VRAM management as proactive prefetching. To make the policy learnable, it uses a compact macro-action space over a demand-aware state, and it trains across nodes from scratch with asynchronous federated policy gradients that exchange gradients rather than private traces. Delay-adaptive lookahead and staleness damping make the training procedure more robust to heterogeneous client speeds.

Under the trace-driven, queue-aware simulation of the Alibaba GenTD26 trace, FedCache improves the hit rate over reactive and learned baselines. With memory headroom, it attains a higher hit rate than an eviction-only Belady reference, by prefetching first-touch requests that a load-on-miss policy must miss. That reference is granted the whole future and instantaneous loads but cannot prefetch, so the two occupy different information settings and neither bounds the other; the comparison is not evidence of surpassing a general offline optimum. FedCache also retains its advantage on held-out traffic and, through asynchronous federation, reduces wall-clock convergence time as more edge nodes participate. A real deployment on H100 edge nodes reproduces the hit-rate advantage (+7 percentage points over the baselines); with a demand-priority IO scheduler, FedCache also achieves up to 40% lower latency than the baselines on matched fast storage. This latency benefit does not extend to slow IO-saturated nodes (Chapter 7). The demand-aware admission gate keeps prefetch churn below reactive LRU, so under the simulated conditions FedCache improves hit rate, latency, and switching cost together rather than trading one objective for another.

8.2 Discussion

8.2.1 Sources of the Gains

The gains summarized above come from two parts of the system: the serving policy and the training scheme. On the serving side, FedCache achieves the highest hit rate, lowest latency, and lowest thrash among the methods reported in Table 6.1, and it stays above the eviction-only Belady reference once there is enough memory headroom, in the qualified sense of Section 6.3. These values come

from a deterministic evaluation, identical across replicas. On the training side, asynchronous federation reduces wall-clock convergence time as nodes are added and avoids the straggler delays that limit synchronous training, while sharing gradients instead of transmitting raw traces. The robustness results indicate that these gains do not require extensive tuning: bounded local PPO steps, staleness handling, and best-checkpoint selection keep the aggregate update stable.

8.2.2 Key Design Factors

The main design choice is the formulation of predictive prefetching as a macro-action MDP. This reduces a large combinatorial model-selection problem to a small decision about whether to prefetch, evict, or idle. It also makes delayed prefetch rewards easier to learn from, because the state determines the concrete model target. Asynchronous federated policy gradients then distribute learning across nodes without transmitting request traces. Together, these choices produce a policy that improves serving quality while training efficiently across the federation and keeping raw traces on the node that produced them. We note that exchanging gradients limits what is transmitted but does not by itself constitute a privacy guarantee; establishing one would require a threat model and a formal mechanism, neither of which is provided here.

The training-scheme comparison in Section 6.7 bounds the federated claim further, and we state the bound explicitly because it is easy to overread. Local-only, centralized, synchronous and asynchronous training all reach the same final hit rate on this workload; federation changes how quickly the policy is reached, not how good it is. The reason is the same design choice that makes the problem learnable:

with the target resolver and admission gate fixing the concrete action, the learned component only decides when to act, so it saturates quickly and from little data — two independently trained policies differ in their weights while behaving identically. Asynchronous federation is therefore best understood as a mechanism for training the policy quickly without pooling traces, and should be reported as a separate contribution from the queue-aware caching itself rather than as a source of the serving gains.

Two considerations bear on where the learned component earns its place, and both point beyond the operating conditions evaluated here. The first is node saturation. All results in this thesis are obtained at a fixed request rate, so the pending queue is consistently deep and the cost of a speculative load is low; this is precisely the regime in which the right amount of prefetching is close to constant. Under lighter or fluctuating load the queue is shallower and a prefetch is more likely to displace something useful before it is needed, so the appropriate degree of selectivity should vary with saturation rather than being fixed at design time. A learned policy can condition that decision on the demand-aware state, and we would expect the benefit of learning — in final quality, and plausibly in how quickly training converges under heterogeneous loads — to grow as node saturation varies across the fleet. We have not measured this, since every configuration reported here runs at the design rate; it is the most direct extension of the present evaluation.

The second is that a learned policy continues to change after deployment. The workload is not stationary: new checkpoints and adapters enter circulation, popularity shifts, and the held-out results in Section 6 already show a measurable distribution shift between early and later parts of the trace. A policy trained

online can track that drift, and the federated aggregator gives it a way to do so continuously and across sites, whereas a rule fixed at design time cannot adapt without being re-tuned by hand. This is an argument for the learning machinery as an operational property rather than as a source of the static gains reported in Chapter 6, and evaluating it would require a longer or evolving trace than the one used here.

8.3 Limitations

Three assumptions bound the conclusions of this study: the model-size parameterization, the fidelity of the discrete-event simulator, and the scope of the workloads evaluated.

8.3.1 Model-Size Parameterization

The trace does not expose per-model parameter footprints, so we parameterized the system model with representative component-class footprints measured on the physical cluster. The measurements are exact for the components and hardware we ran, but the assignment of anonymized trace identifiers to those footprints is synthetic (Appendix A.2). Results that depend on absolute packing, such as the low-memory crossover and the $\sim$17$\times$ oversubscription estimate, inherit this assumption, and the crossover capacity may shift under a different size distribution or a different identifier-to-tier assignment. The sensitivity analysis in Section 6.4 tests this directly: retraining FedCache under six different mappings, including two altered distributions, leaves the qualitative conclusions intact. FedCache exceeds every learned baseline in all six, and its margin over LRU is stable at +2.6 to +3.1

points across draws of the measured distribution, while oversubscription varies from 16.9× to 22.1×. As expected, the absolute values and the crossover capacity are conditional on the mapping: the crossover ranges from 3 to 20 GB and scales with oversubscription, and under an adversarial heavy-tailed distribution (28.4×) C=15 falls below the crossover, so FedCache trails LRU there while still leading the learned baselines. What remains untested is popularity skew and grounding the sizes in more detailed measured load-latency tables, which would further tighten the system model.

8.3.2 Simulation Fidelity

Most results are obtained with a discrete-event simulator, while Chapter 7 adds a real experimentation replay that captures physical timing not present in the simulator. Several modelling choices remain: size-scaled cold-start latency, size-dependent load delay $\Delta(m)$, and the abstraction of a full pipeline as a single cacheable unit. The dataset's component-level structure, including separately loadable base models, LoRA adapters, and ControlNets, is not exploited. The experimental replay shows that PCIe and storage-bandwidth contention on a serialized IO path is the main omitted effect and can gate FedCache's latency benefit. Streaming or partial weight loads, request batching, and full diffusion inference remain unmodelled. In addition, asynchronous federation is simulated with an event queue and synthetic heterogeneous client delays; the real experiments replay trained policies but do not yet run federated training across the physical cluster.

8.3.3 Scope and Threats to Validity

Every reported number comes from a deterministic, trace-driven evaluation and is identical across evaluation replicas. That determinism, however, removes evaluation noise only, not the variance a different training seed could introduce; we do not characterize training-seed sensitivity. The policies retrained across mappings in Section 6.4 reach the same behavioural ceiling, which is suggestive of stability, but a systematic seed sweep with confidence intervals is left to future work. The main remaining threats are workload coverage and scale. We use one production trace, a fixed model count (M=100), a modest simulated fleet ($N \leq 20$), and a small held-out segment because the trace is short. The generalization result should therefore be read as indicative rather than conclusive; evaluation across multiple traces remains future work. We also assume one GPU per node. Sharded or multi-GPU models and cross-node model migration are outside the scope of this study. The four-node real deployment reproduces the hit-rate result on real GPUs and exposes the storage-bandwidth dependence of the latency result, but fleet-scale validation of asynchronous federated training remains open.

8.4 Future Work

The limitations identified above point to several extensions. A proactive oracle in a matched information setting—one that observes the same H-step queue, pays the same load delay, and respects the same capacity, but selects prefetches and evictions optimally over the horizon—would bound how much of the achievable gain the learned policy actually captures. The eviction-only Belady reference of Section 6.3 cannot serve this role, since it occupies a different information setting, so the

gap between FedCache and the best queue-aware policy is currently unmeasured. Grounding model sizes and cold-start latencies in more detailed measured data would tighten the system model. Modelling pipelines at the component level, with separate caching decisions for base checkpoints, LoRA adapters, and ControlNets, would expose finer-grained prefetching opportunities than the current single-unit abstraction. Such component-level management could also extend to multi-tier memory hierarchies, where small adapters remain in fast VRAM while larger base models are staged from slower memory tiers. On the system side, experimental replay already captures real IO contention; adding full diffusion inference, streaming, and batched loads, and live fleet-scale federated training would bring the evaluation closer to production. More broadly, our results suggest that hiding GenAI cold starts requires acting ahead of demand rather than reacting to it, and that the policy governing when to act can be learned collaboratively across edge sites without transmitting raw request traces.

Bibliography

[1] S. S. Bhuyan, V. Sateesh, N. Mukul, A. Galvankar, A. Mahmood, M. Nauman, A. Rai, K. Bordoloi, U. Basu, and J. Samuel, "Generative artificial intelligence use in healthcare: Opportunities for clinical excellence and administrative efficiency," *Journal of Medical Systems*, vol. 49, p. 10, 2025.

[2] R. Rombach, A. Blattmann, D. Lorenz, P. Esser, and B. Ommer, "High-resolution image synthesis with latent diffusion models," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, (New Orleans, LA, USA), pp. 10684–10695, IEEE, 2022.

[3] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," in *International Conference on Learning Representations (ICLR)*, (Virtual Event), OpenReview.net, 2022.

[4] Y. Lin, S. Wu, S. Luo, H. Xu, H. Shen, C. Ma, M. Shen, L. Chen, C. Xu, L. Qu, and K. Ye, "Understanding diffusion model serving in production: A top-down analysis of workload, scheduling, and resource efficiency," in *Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC '25)*, (New York, NY, USA), pp. 1–15, ACM, 2025.

[5] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[6] T. Pfandzelter and D. Bermbach, "Komet: A serverless platform for low-earth orbit edge services," in *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC '24)*, (New York, NY, USA), pp. 866–882, ACM, 2024.

[7] R. Das and S. Sidhanta, "Live migration of containers in the edge," in *Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER)*, (Setúbal, Portugal), pp. 51–62, SCITEPRESS, 2022.

[8] E. Z. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in *International Conference on Machine Learning (ICML)*, (Virtual Event), pp. 6237–6247, PMLR, 2020.

[9] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan, "Driving cache replacement with ML-based LeCaR," in *USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, (USA), USENIX Association, 2018.

[10] Y. Sui, H. Yu, Y. Hu, J. Li, and H. Wang, "Pre-warming is not enough: Accelerating serverless inference with opportunistic pre-loading," in *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC '24)*, (New York, NY, USA), pp. 178–195, ACM, 2024.

[11] C. Segarra, I. Durev, and P. R. Pietzuch, "Is it time to put cold starts in the deep freeze?," in *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC '24)*, (New York, NY, USA), pp. 259–268, ACM, 2024.

[12] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 2nd ed., 2018.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017. arXiv preprint arXiv:1707.06347.

[14] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," in *International Conference on Learning Representations (ICLR)*, (San Juan, Puerto Rico), 2016.

[15] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, (Fort Lauderdale, FL, USA), pp. 1273–1282, PMLR, 2017.

[16] C. Xie, S. Koyejo, and I. Gupta, "Asynchronous federated optimization," 2019. arXiv preprint arXiv:1903.03934.

[17] J. Nguyen, K. Malik, H. Zhan, A. Yousefpour, M. Rabbat, M. Malek, and D. Huba, "Federated learning with buffered asynchronous aggregation," in *International Conference on Artificial Intelligence and Statistics (AISTATS)*, (Virtual Event), pp. 3581–3607, PMLR, 2022.

[18] J. Qi, Q. Zhou, L. Lei, and K. Zheng, "Federated reinforcement learning: Techniques, applications, and open challenges," *Intelligence & Robotics*, vol. 1, no. 1, pp. 18–57, 2021.

[19] G. Lan, D.-J. Han, A. Hashemi, V. Aggarwal, and C. G. Brinton, "Asynchronous federated reinforcement learning with policy gradient updates: Algorithm design and convergence analysis," in *International Conference on Learning Representations (ICLR)*, (Singapore), OpenReview.net, 2025.

[20] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. Darlow, J. Wang, and A. Barker, "How does it function?: Characterizing long-term trends in production serverless workloads," in *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23)*, (New York, NY, USA), pp. 443–458, ACM, 2023.

[21] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX Annual Technical Conference (USENIX ATC)*, (USA), pp. 205–218, USENIX Association, 2020.

[22] A. Kumari and B. Sahoo, "ACPM: Adaptive container provisioning model to mitigate serverless cold-start," *Cluster Computing*, vol. 27, pp. 1333–1360, 2024.

[23] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (New York, NY, USA), pp. 467–481, ACM, 2020.

[24] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," in *USENIX Annual Technical Conference (USENIX ATC)*, (USA), pp. 57–70, USENIX Association, 2018.

[25] Y. Lan, X. Peng, and Y. Wang, "Snapipeline: Accelerating snapshot startup for FaaS containers," in *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC '24)*, (New York, NY, USA), pp. 144–159, ACM, 2024.

[26] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at Alibaba cloud function compute," in *USENIX Annual Technical Conference (USENIX ATC)*, (USA), pp. 443–457, USENIX Association, 2021.

[27] F. Romero, G. I. Chaudhry, Í. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$T: A transparent auto-scaling cache for

serverless applications," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, (New York, NY, USA), pp. 122–137, ACM, 2021.

[28] R. B. Roy, T. Patel, and D. Tiwari, "IceBreaker: Warming serverless functions better with heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, (New York, NY, USA), pp. 753–767, ACM, 2022.

[29] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "ServerlessLLM: Low-latency serverless inference for large language models," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (USA), pp. 135–153, USENIX Association, 2024.

[30] Y. Sheng, S. Cao, D. Li, C. Hooper, N. Lee, S. Yang, C. Chou, B. Zhu, L. Zheng, K. Keutzer, J. E. Gonzalez, and I. Stoica, "S-LoRA: Serving thousands of concurrent lora adapters," in *Proceedings of Machine Learning and Systems (MLSys)*, (Santa Clara, CA, USA), mlsys.org, 2024.

[31] L. Chen, Z. Ye, Y. Wu, D. Zhuo, L. Ceze, and A. Krishnamurthy, "Punica: Multi-tenant lora serving," in *Proceedings of Machine Learning and Systems (MLSys)*, (Santa Clara, CA, USA), pp. 1–13, mlsys.org, 2024.

[32] B. Wu, R. Zhu, Z. Zhang, P. Sun, X. Liu, and X. Jin, "dLoRA: Dynamically orchestrating requests and adapters for lora llm serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (USA), pp. 911–927, USENIX Association, 2024.

[33] 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)*, (USA), pp. 443–462, USENIX Association, 2020.

[34] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. E. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with PagedAttention," in *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, (New York, NY, USA), pp. 611–626, ACM, 2023.

[35] R. Li, S. Pal, V. N. Pullu, P. Sinha, J. Ryoo, L. K. John, and N. J. Yadwadkar, "Oneiros: KV cache optimization through parameter remapping for multi-tenant LLM serving," in *Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC '25)*, (New York, NY, USA), pp. 88–101, ACM, 2025.

[36] S. Zhang, L. Diao, Z. Meng, S. Wang, W. Lin, and C. Wu, "Dyorc: Efficient serving of dynamic machine learning workflows," in *Proceedings of the 2025 ACM Symposium on Cloud Computing (SoCC '25)*, (New York, NY, USA), pp. 111–124, ACM, 2025.

[37] Y. Lin, S. Peng, C. Lu, C. Xu, and K. Ye, "Flexpipe: Adapting dynamic llm serving through inflight pipeline refactoring in fragmented serverless clusters," in *European Conference on Computer Systems (EuroSys '26)*, (New York, NY, USA), pp. 1–17, ACM, 2026.

[38] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[39] Z. Song, D. S. Berger, K. Li, and W. Lloyd, "Learning relaxed Belady for content distribution network caching," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, (USA), pp. 529–544, USENIX Association, 2020.

[40] M. Hashemi, K. Swersky, J. A. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning memory access patterns," in *International Conference on Machine Learning (ICML)*, (Stockholm, Sweden), pp. 1919–1928, PMLR, 2018.

[41] C. Zhong, M. C. Gursoy, and S. Velipasalar, "A deep reinforcement learning-based framework for content caching," in *Conference on Information Sciences and Systems (CISS)*, (Princeton, NJ, USA), pp. 1–6, IEEE, 2018.

[42] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM SIGCOMM Conference*, (New York, NY, USA), pp. 270–288, ACM, 2019.

[43] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," in *Proceedings of Machine Learning and Systems (MLSys)*, (Austin, TX, USA), pp. 429–450, mlsys.org, 2020.

[44] Q. Wu, W. Wang, P. Fan, Q. Fan, H. Zhu, and K. B. Letaief, "Cooperative edge caching based on elastic federated and multi-agent deep reinforcement learning in next-generation networks," *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, pp. 4179–4196, 2024.

[45] J. Wang, J. Hu, J. Mills, G. Min, and M. Xia, "Federated ensemble model-based reinforcement learning in edge computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, 2023.

[46] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, (New York, NY, USA), pp. 785–794, ACM, 2016.

[47] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[48] NVIDIA Corporation, "NVIDIA multi-instance GPU user guide." `https://docs.nvidia.com/datacenter/tesla/mig-user-guide/`, 2026. Accessed: 2026-07-08.

Appendix A

Reproducibility Details

This appendix documents the configuration behind every number reported in Chapters 6 and 7, at the level of detail needed to re-run the experiments or to judge how far the results depend on a particular parameterization. It covers trace preprocessing and splits, the model universe and its size parameterization, the exact event ordering of the simulator, the policy network and training budget, the baseline configurations, the physical cluster, and the demand-priority IO scheduler. All referenced scripts are in the public repository.[1]

A.1 Trace Preprocessing and Splits

The request-level GenTD26 log [4] is read in chronological chunks of 10,000 requests. Three columns are used: the anonymized target identifier `checkpoint_model_version_id`, and the request features `prompt_length` and `num_inference_steps`. Rows with missing values in these columns are dropped. The two features are normalized to `prompt_length`/256 and

[1] `https://github.com/pacslab/fedCache`

`num_inference_steps`/100; the latter is the completion weight $\hat{c}(m_t)$ that appears in the reward (4.5). Requests are consumed in trace order, one per simulation step; no resampling, reordering, or rate scaling is applied.

Splits. The generalization experiment (Figure 6.3) splits the trace by chunk index: the first 60% of chunks form the training segment and the remaining 40% the held-out segment. The split is temporal, so the held-out segment contains model identifiers that never appear during training. The non-IID federated experiments instead give client k the chunks whose index satisfies index mod $N = k$, so the N clients read disjoint shards of the same trace.

The evaluation protocol has no separate validation split. Model selection is the best-by-hit-rate checkpoint described in Section A.4, which is scored on the training stream every 50 global steps. Checkpoint selection and training therefore share data, and the held-out segment is used only once, for the result in Figure 6.3. A three-way train/validation/test protocol on a longer trace would be a stronger design and is left to future work.

Evaluation windows. The end-to-end comparison (Table 6.1), the gate ablation (Figure 6.6), and the physical replay (Table 7.1) use 5000 requests. The per-configuration sweeps (Figures 6.1–6.3) use 3000 requests from the earlier, more stationary part of the trace, and the training-time curves (Figures 6.4 and 6.5) use 1500-request evaluation windows so that the curve can be sampled frequently. This is the source of the absolute differences in hit rate between figures noted in Chapter 6; the ordering of the policies is unaffected.

A.2 Model Universe and Size Parameterization

Model-ID selection. Identifiers are ranked by their total request count over the whole trace, and the $M = 100$ most frequently requested models form the cacheable universe. The ranking is computed once, globally, so a given model receives the same index in every chunk; the tail beyond rank 100 (four identifiers, about 0.01% of requests) is never cacheable and is always served as a miss. The map is built by `validation/build_model_id_map.py` and stored as `data/model_id_map.csv`.

Size-tier assignment. The trace records per-request load latencies but does not expose the parameter footprint of each anonymized model, so footprints must be supplied by the system model. Four component classes were measured on the physical cluster of Section A.6 — LoRA adapters, ControlNets, base checkpoints, and full composed pipelines — and the measured ranges define the four size tiers in Table A.1. Each of the $M = 100$ identifiers is then assigned a tier by sampling from the tier-share column, and a footprint by sampling uniformly within that tier's range, under a fixed seed (42) so that the assignment is reproducible. Identifiers for which a directly measured footprint is available take that measured value instead of a sampled one. The generator is `generate_meta.py` and its output is `data/model_meta.csv`.

This is the most consequential modelling assumption in the thesis, and it should be read as a parameterization rather than as ground truth. The measured footprints are exact for the components and hardware of Section A.6, but the mapping from anonymized trace identifiers to those footprints is synthetic: a different draw permutes which model receives which size, and therefore changes the packing

TABLE A.1: Size tiers used to parameterize the $M = 100$ model universe. The ranges are the measured footprints of the corresponding component classes on the physical cluster; the share column is the probability with which an anonymized trace identifier is assigned to that tier.

Component class	Share	Footprint range (GB)
LoRA adapter	0.45	0.20–0.80
ControlNet	0.20	1.50–2.50
Base checkpoint	0.25	2.50–7.00
Full pipeline	0.10	8.00–14.00

problem. The resulting universe spans 0.2–13.4 GB with an aggregate footprint of about 253 GB, or roughly 17× the default per-node capacity of $C = 15$ GB. Results that depend on absolute packing — the low-memory crossover, the oversubscription ratio, and absolute hit-rate and latency values — are conditional on this draw. Section 6.4 evaluates robustness to this choice by retraining FedCache under six mappings, including reseeds of this tier distribution and two altered distributions; the qualitative conclusions hold, while the crossover capacity and absolute values shift with the mapping as expected.

A.3 Simulator Semantics

Constants. Table A.2 lists the simulator constants. One step corresponds to 1 s and to exactly one request, so the arrival rate is fixed at one request per second by construction; the simulator does not model an arrival process, service times, or a queueing discipline separately from the trace order.

Event ordering within a step. Each simulation step executes the following operations in this order.

TABLE A.2: Simulator constants. Sizes are normalized internally by a fixed 80 GB scale factor.

Quantity	Value	Note
Step duration	1000 ms	one request per step
Chunk size	10,000 requests	streaming granularity
Hit latency ℓ_{hit}	50 ms	
Load latency	$5000 + 770\, s(m)$ ms	$s(m)$ in GB
Load delay $\Delta(m)$	max(1, round(load ms/1000))	5–15 steps
Recency decay τ	5.0	in (4.2)
Size normalizer	80 GB	

1. **Warming completion.** Every model in W_t has its remaining delay decremented by one; models that reach zero move from W_t to A_t. Completion is therefore checked *before* the current request is served, so a load that finishes on step t can serve the step-t request.

2. **Macro-action.** For PREFETCH, resident models are evicted in least-recently-used order until the target fits, and the target then enters W_t and reserves its full footprint immediately. If it still does not fit, the action is a no-op and incurs the capacity penalty. For EVICT, the target is removed from A_t and its footprint released. IDLE does nothing.

3. **Service.** The request is a hit if its model is in A_t (50 ms). If the model is in W_t, the request waits only for the *remaining* delay and the model is then promoted to A_t. Otherwise it is a full cold start charged at the load latency of Table A.2; a cold-start load is transient and the model is not retained.

4. **Accounting.** Recency counters advance, the occupancy penalty $\lambda_{\text{v}} u_t$ is applied, the request index advances, and the next chunk is fetched if the current one is exhausted.

Note the asymmetry in step 2 and step 3: a prefetch may evict to make room, whereas a demand miss is served transiently and never displaces a resident model. This is what makes prefetching the only mechanism that changes cache membership ahead of demand, and it is also why the admission gate is load-bearing.

Queue formation and window. The "pending queue" is the next H entries of the trace in file order, positions $t \ldots t{+}H{-}1$ of the current chunk, truncated at the chunk boundary and zero-padded. It is not produced by a modelled arrival or service process. The queue window is therefore *exact* over the next H requests rather than forecast, and the reported gains hold under the assumption that a backlog of at least H requests exists — the deployment condition discussed in Chapters 7 and 8. From this window the environment computes the recency-weighted demand d_t with $\tau = 5$, normalized by its peak entry, and the unmet-demand vector $\tilde{d}_t$ of (4.2).

The prefetch target is the earliest position $j \geq 1$ in the window whose model is neither available nor warming, fits within C, satisfies the catchability test $j \geq \Delta(m_{t+j})$, and passes the admission gate. The gate admits the candidate if it fits in free space, or if its demand strictly exceeds the demand of the least-recently-used resident model it would displace. The eviction target is the resident model minimizing $(d_t[m], -\text{age}(m))$, that is, lowest demand with ties broken towards the older model.

State vector. The state has $3 + 4M + 3H = 463$ dimensions at the default M=100, H=20: three occupancy scalars; four length-M blocks holding cache status (available 1, warming $\frac{1}{2}$, absent 0), clipped recency, d_t, and $\tilde{d}_t$; and the

raw $H \times 3$ queue window of (model index, normalized prompt length, normalized completion weight).

Reward constants. $R_{\text{hit}} = 1.0$; completion bonus $w_{\text{c}} = 0.5$; cold-start penalty $\kappa = 0.2$ per second of measured latency; occupancy weight $\lambda_{\text{v}} = 0.05$; switching penalties of 0.1 per load and 0.05 per eviction; capacity-violation penalty 1.0.

A.4 Policy Network and Training

Architecture. A shared backbone maps the 463-dimensional state through Linear(463→256), LayerNorm, ReLU, Linear(256→128), ReLU. Two heads follow: an actor Linear(128→64), ReLU, Linear(64→3) over the macro-actions, and a critic Linear(128→64), ReLU, Linear(64→1). Infeasible macro-actions are masked by setting their logits to -10^9 before the categorical distribution is formed.

Local PPO. Optimizer Adam at learning rate 3×10^{-4}. Each local update collects a 512-step rollout and takes 4 full-batch epochs over it; there is no minibatching within an epoch, and the reference log-probabilities are taken at the start of the first epoch. Advantages use GAE with γ=0.99 and λ=0.95 and are standardized to zero mean and unit variance. The objective is the clipped surrogate at ϵ=0.2, plus 0.5 times the value MSE, minus 0.005 times the policy entropy. Gradients are clipped to a global norm of 0.5.

Federated aggregation. $N = 5$ clients by default and $T_{\max} = 1000$ global steps; server learning rate $\eta_s = 0.3$; delay-adaptive lookahead coefficient $\alpha = 0.8$; staleness damping $1/\sqrt{1+\sigma}$. Client compute delays are drawn uniformly from $[1, 6]$

simulator time units, except in the asynchronous-versus-synchronous comparison, which widens them to $[1, 20]$ to induce stragglers. The FedProx coefficient is $\mu = 0$ unless swept. Updates containing non-finite values are discarded rather than applied. Every 50 global steps the server evaluates the global policy greedily for 3000 steps and retains the highest-hit-rate snapshot; that snapshot is the deployed policy.

Budgets and seeds. The reported policy is trained with the budget in Table 5.1. The per-figure experiments use a reduced budget of 150 global steps with the same 512-step rollouts, which is sufficient because the macro-action policy converges within 10–20 global updates (Figure 6.5). Python, NumPy, and PyTorch are seeded from a single value, 0, for both training and evaluation.

A.5 Baseline Configurations

All baselines run inside the same simulator, against the same request stream, the same capacity C, the same size parameterization, and the same load-delay model, and are scored by the same evaluator. Each is a *load-on-miss* policy augmented with its own prefetch rule: when the current request misses, the baseline loads the requested model; when the current request hits, it may issue one speculative prefetch chosen by its own predictor.

LRU. On a miss, load the requested model and evict in least-recently-used order until it fits. No prefetching. No hyperparameters.

IceBreaker [28]. Per-model invocation history, with the dominant period estimated by a real FFT over a 256-step binary occupancy signal built from the last 64 recorded invocations; the largest non-DC magnitude bin gives the period, and the fallback when the spectrum is flat is the mean inter-arrival time of the last 10 invocations. A model is a pre-warm candidate once it has at least 4 recorded invocations and its predicted next arrival falls within $\Delta(m) + 2$ steps. The earliest such candidate is prefetched. The heterogeneous-server cost optimization of the original system is omitted, since the simulated fleet has a single GPU tier.

XGBoost [46]. An `XGBClassifier` with 50 trees, maximum depth 4, learning rate 0.1, and log-loss objective. Features are the last 5 binary occupancy indicators for a model; the label is whether that model is requested next. Training examples are collected online, retaining all positive windows and 5% of all-zero windows to limit imbalance; the model is refitted every 500 steps once at least 100 examples have accumulated, after which the buffer is cleared. At decision time the highest-probability non-resident model is prefetched, but only if its probability is at least twice the mean probability across candidates.

ACPM [22]. A single-layer LSTM [47] with hidden size 32 over a scalar input, followed by Linear(32→16), ReLU, Linear(16→1). The input sequence is the last 12 bins of per-model request counts, each bin covering 10 simulation steps. Trained online with Adam at learning rate 0.01 and MSE loss, refitted every 500 steps for 2 epochs at batch size 64. The non-resident model with the highest forecast count is prefetched when that forecast is at least 0.5 requests in the next bin. This adapts the original container-provisioning decision to model residency under a fixed VRAM capacity.

Synchronous FedAvg [15]. The identical local agent, network, and PPO configuration, aggregated at a round barrier instead of asynchronously. A round advances at the slowest client's delay draw, which is the effect the comparison isolates.

Belady-OPT [38]. Offline eviction reference computed over the same request sequence, the same capacity, and the same size parameterization, but with instantaneous loads. On a miss it evicts the resident model whose next reference is furthest in the future until the requested model fits, then admits it. It is a load-on-miss policy and cannot prefetch, so it necessarily misses the first reference to every model; this is the asymmetry discussed in Section 6.3.

Two implementation details bound the strength of this reference. The next-use distance is computed over a 4000-request horizon: any model not referenced within that horizon is treated as having equal, maximal distance, so eviction among such models is arbitrary rather than optimal. The reference is therefore a bounded-horizon approximation of Belady's rule, not exact global optimality, and it is scored on the same 3000-request window as the other policies in Figure 6.2. Both details make it a slightly weaker reference than true Belady, which is a further reason not to read the comparison as surpassing an offline optimum.

A.6 Physical Cluster

Hardware. Four primary nodes in a **k3s** cluster, each exposing one NVIDIA H100 partitioned with MIG [48] at the `1g.24gb` profile, giving 24 GB of usable GPU memory advertised to Kubernetes as `nvidia.com/gpu: 1`. Two additional

TABLE A.3: Per-node cold-load calibration on the four matched nodes.

Node	Affine cold-load fit	13.4 GB cold / warm
afrl-1	$455 + 87\,s$ ms	1.39 s / 0.96 s
worker-1	$459 + 91\,s$ ms	1.45 s / 0.96 s
worker-2	$464 + 98\,s$ ms	1.48 s / 0.96 s
worker-3	$471 + 97\,s$ ms	1.50 s / 0.96 s

nodes with disks roughly 25–30% slower are retained to bound behaviour on under-provisioned storage. Each GPU replay job was scheduled with 2–4 vCPUs and 8–24 GiB of host memory (`validation/k8s/40-gpu-b1-job.yaml`); the agent and aggregation code run on CPU in a shared virtual environment.

Storage throughput. Each node was calibrated by timing page-cache-evicted cold loads across the 0.2–13.4 GB size range with five repetitions per size, and fitting an affine model $a + b\,s(m)$ in milliseconds. The four matched nodes agree within a few percent (Table A.3). The host-to-device leg is uniform and linear at 25.6 GB/s on every node, so the disk read dominates the cold path; the disk leg is sub-linear in size because of a fixed per-load overhead, which is why the affine fits have moderate R^2 (0.68–0.72).

Two measurement caveats apply. The guest cannot evict the hypervisor's page cache, so a "cold" read here is guest-cold and a physically cold read may be slower. Conversely, steady-state popular models are cache-warm, so these figures may also overstate production cold-start costs. Note that these measured load times are substantially faster than the 5–15 s cold starts assumed by the simulator's constants (Table A.2); the simulator's parameterization reflects the load latencies reported in the production trace rather than the storage of this particular cluster.

Replay harness. A resident model is a CUDA tensor of the model's exact byte size on the MIG slice; a warming model is an in-flight background load thread; a load is a cold disk read with `posix_fadvise(DONTNEED)`, staged through a 1 GiB pinned host buffer, then copied host-to-device over PCIe. Because the 100-model universe totals about 253 GB and does not fit a node's disk, each node holds one shared 14 GB blob file and a load of model m reads $s(m)$ GB from a per-model offset, so the disk and PCIe work matches a dedicated file of that size. All loads on a node serialize behind a single IO lock, matching the single disk-to-PCIe path of a single-GPU node.

Two deviations from a literal serving stack are deliberate. Reactive miss loads are transient in the simulator, so they stream through a fixed GPU scratch buffer instead of materializing a full tensor — a 13.4 GB transient on top of a 15 GB resident set would exceed the 24 GB slice. The bytes moved, and therefore the timing, are unchanged. Second, diffusion inference itself is not executed: the harness emulates byte movement on both the hit and miss paths. Adding real inference would compress relative latency differences, which is why the hit-rate result is the more robust experimental claim.

Software. A shared Python virtual environment provides PyTorch (CPU wheels for the agent and aggregation code), pandas, XGBoost, and scikit-learn. Cluster orchestration is k3s; monitoring is kube-prometheus-stack at a 5 s scrape interval plus `dcgm-exporter`, which requires `runtimeClassName: nvidia`. The NVIDIA device plugin requires `migStrategy=single` under this vGPU profile, and the GPU jobs run with `PYTORCH_NO_CUDA_MEMORY_CACHING=1` because the restricted

MIG profile blocks the caching allocator's NVML query. On this profile, MIG-scoped DCGM profiling counters are available but PCIe and power counters are not, so prefetch bandwidth is measured from application-level load timings. The GPU replay jobs ran in the `pytorch/pytorch:2.5.1-cuda12.4-cudnn9-runtime` container image (PyTorch 2.5.1, CUDA 12.4, cuDNN 9), with pandas, scikit-learn, and XGBoost installed at job startup.

A.7 Demand-Priority IO Scheduler

The scheduler is a property of the IO path in the replay harness, not of the learned policy, and it is the single toggle between the "off" and "on" columns of Table 7.1. Each node has one service position: at most one transfer is in flight and the rest wait.

Off (FIFO). Every load, demand or speculative, is served first-come-first-served on the single disk-to-PCIe path. This is the baseline configuration.

On (demand priority). Loads are classified into two classes. A load is **demand** if some request is actually waiting for it: the load for the current request, a full-miss serve load, or a warming model whose request has just arrived. A load is **speculative** if it is a prefetch for some future model. When the scheduler is on, the waiting set is ordered demand-first, FIFO within each class, so a demand load cuts ahead of queued speculative prefetches. An arriving request for a still-warming model reclassifies that in-flight load as demand, which expedites it.

Two properties matter for interpreting the results. First, the transfer already in flight is never preempted: reordering applies only to the waiting set, so a demand load can still be delayed by up to one in-progress speculative transfer. Second, nothing is cancelled or dropped — speculative loads still complete, so the prefetch hit rate is preserved and only the cost per miss changes. An earlier design that applied backpressure by dropping speculative loads was rejected in favour of this no-drop reordering.

Consequently the scheduler helps exactly when the IO path has spare capacity and cannot help when it does not. On the four matched fast nodes it lowers FedCache latency from 412 to 225 ms and removes node-to-node variance; on the two slow, IO-saturated nodes it increases latency, because reordering a backlog that cannot drain moves the wait rather than removing it (Table 7.2).

A.8 Mapping from Results to Scripts

Table A.4 maps each reported result to the script that produces it. Each experiment script writes a JSON record under `experiments/results/`, and the physical-cluster jobs write a `summary.json` plus per-request event logs, so that reported values can be traced back to raw output.

TABLE A.4: Reported results and the scripts that generate them.

Result	Script
Table 6.1 (end-to-end)	`evaluate.py`
Table 6.2 (thrash breakdown)	`experiments/exp_thrash_breakdown.py`
Table 6.3 (size sensitivity)	`experiments/exp_size_sensitivity.py`
Table 6.4 (training schemes)	`experiments/exp_federated_comparison.py`
Figure 6.1 (VRAM sweep)	`experiments/exp_vram_sweep.py`
Figure 6.2 (Belady)	`experiments/exp_belady.py`
Figure 6.3 (generalization)	`experiments/exp_generalization.py`
Figure 6.4 (client scaling)	`experiments/exp_client_scaling.py`
Figure 6.5 (learning curve)	`experiments/exp_learning_curve.py`
Figure 6.6 (admission gate)	`experiments/exp_gate_ablation.py`
Figure 6.7(a) (FedProx μ)	`experiments/exp_fedprox_mu.py`
Figure 6.7(b) (prefetch cost)	`experiments/exp_reward_sweep.py`
Figure 6.7(c) (queue window H)	`experiments/exp_queue_window.py`
Async vs. sync aggregation	`experiments/exp_async_vs_sync.py`
Server learning-rate robustness	`experiments/exp_server_lr.py`
Lookahead coefficient α	`experiments/exp_lookahead.py`
Staleness damping	`experiments/exp_staleness.py`
Tables 7.1 and 7.2 (cluster)	`validation/edge/run_gpu_b1.py`
Load-time calibration (Table A.3)	`validation/calibrate_loads.py`