Cost-Efficient Storage and Caching in Public Clouds

Hojin Park

CMU-CS-25-133
September 2025

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work is supported in part by US Army Research Lab under Contract No. W911NF-20-F-0025 and Korea Foundation for Advanced Studies. We thank the members and companies of the PDL Consortium (Amazon, Bloomberg, Datadog, Google, Honda, Intel, Jane Street, LayerZero Research, Meta, Microsoft, Oracle, Pure Storage, Salesforce, Samsung, Western Digital, Hewlett Packard Enterprise, Hitachi, IBM, NetApp, Seagate, Two Sigma, Alibaba, Datrium, Micron). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Cloud Storage, Public Cloud Resource Provisioning, Cross-Cloud/Cross-Region Data Access, Caching, Cache Prefetching, System Auto-Configuration

Dedicated to my wife, Youli, and to all of my family, for their love and support.

Abstract

As modern data-intensive workloads increasingly migrate to the public cloud, managing the resulting costs has emerged as a pressing challenge despite the operational simplicity and elasticity that cloud environments offer. Although many efforts in cost optimization have focused on computation, storage-related costs have received comparatively less attention despite being a significant portion of total cloud spending. In particular, two categories dominate storage-related costs in public cloud: the cost of deploying and operating storage clusters in the cloud, and the cost of accessing data across geographically distributed regions or clouds. These challenges cannot be effectively addressed by existing optimization techniques developed for on-premise environments, since they often overlook the unique characteristics of public clouds, including elastic resource provisioning, diverse cost-performance trade-offs, and dynamic and unique access patterns found in cloud object storage workloads.

This dissertation addresses these challenges by proposing a cost-efficient approach to designing storage and caching systems that are cloud-aware, elastic, and adaptive to workload behavior. It introduces three systems that target key aspects of cloud storage cost optimization. First, Mimir reduces the cost of the deployment of storage clusters by automatically selecting cost-effective combinations of virtual machines and block storage types, based on profiling workload characteristics and benchmarking available resource options. Second, Macaron reduces cross-region and cross-cloud data access costs by auto-configuring a cache with a tiered storage architecture that leverages low-cost object storage and dynamically resizes the cache based on workload changes. Third, Macaron+ builds on Macaron by introducing a cost-aware prefetching technique that analyzes object-level access patterns to reduce latency in workloads with high cold miss ratios, while preserving cost-efficiency. Together, these systems demonstrate that by tailoring automated resource selection, adaptive configuration, and predictive techniques to the characteristics of the public cloud, it is possible to significantly reduce the cost of storing and accessing data.

Acknowledgments

Completing this dissertation has been a long and deeply collaborative journey. The work presented here would not have been possible without the guidance, encouragement, and support of many mentors, colleagues, collaborators, friends, and family. I offer the following acknowledgements with deep gratitude.

First and foremost, I am deeply grateful to my advisors, Greg Ganger and George Amvrosiadis. From the very beginning, they taught me how to find important research problems, how to start and carry out research, how to present results clearly, and how to work effectively with colleagues in industry. Greg and George were always available to answer questions and provide guidance at every stage of my PhD. Their example of thoughtful questioning, feedback, and calm leadership has shaped the way I approach research and collaboration.

I am grateful to my thesis committee members, Jignesh Patel and Carlo Curino, for their time and incisive feedback. Jignesh contributed constructive guidance throughout my thesis work and at multiple PDL retreats, and he supported my professional development through the speaking-skills committee. Carlo, despite his busy schedule, showed strong interest in my work and offered valuable new perspectives with new insights that improved several of my papers.

I am grateful to all the students and colleagues in the Parallel Data Lab. In particular, Ziyue Qiu was an excellent collaborator on our caching work and asked many constructive questions that advanced our ideas. Thanks also to everyone who collaborated on my research projects, including Saileshwar Karthik, Mohit Gaggar, Fulun Ma, Somansh Satish, Anurag Choudhary, Midhush Manohar, Thevendria Karthic, Shalini Shukla, and Hao Yang Lu. I appreciate each of you and the many ways you contributed to this research. For timely advice and helpful discussions during my PhD, I am thankful to Junwoo Park, Juncheng Yang, Sara McAllister, and Daniel Wong.

My sincere thanks go to the PDL staff who kept everything running. Jason Boles, Bill Courtright, Chad Dougherty, and Mitch Franzos were always ready to help when there were issues with large-scale servers or urgent hardware needs. Joan Digney handled posters and web pages with great care, and Karen Lindenfelser kept PDL operations moving smoothly. I also thank the CSD staff, including Deb Cavlovich, Jenn Landefeld, Matthew Stewart, and Charlotte Yano, for their indispensable administrative support.

I am grateful to our industry collaborators and partners. Thank you to Ishai Menache at Microsoft Research for opportunities to work on VM packing, to Adriana Szekeres at VMware for helping collect data for Macaron, and to Jing Zhao at Uber for enabling data collection and supporting hybrid cloud research. I also appreciate the helpful input and feedback from Nat Wyatt, Pat Helland, and Vaibhav Arora at Salesforce, Carlos Costa at IBM, Jeff Butler at Microsoft, and Michael

Abd-El-Malek at Snowflake.

To my close friends and labmates, thank you for keeping me balanced and motivated. I am especially grateful to Jekyeom Jeon and Hyungjoo Kim for being steady companions for meals, board games, and the support that made finishing this degree possible. I also thank George's PhD student group, including Ankush Jain, Sheng Jiang, Nj Mukherjee, Jekyeom Jeon, and Ziyue Qiu, for many memorable moments and helpful conversations.

Finally, and most importantly, I thank my family. My parents, Jooyeon and Seongjun, have supported me unconditionally from childhood through to the completion of this PhD; their sacrifices and encouragement made everything here possible. My sister Hojung and my brother-in-law Jason helped me settle in the United States and provided care and steadiness when it was most needed. Above all, I owe my deepest gratitude to my wife, Youli. You believed in me, made the difficult move to the U.S., took on so much so that I could focus on my work, and supported me every day. I could not have completed this journey without you.

To everyone named here and to many more who have helped me in ways both large and small, thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As enterprises increasingly migrate to public clouds for their data-intensive workloads [2, 3, 4], cloud infrastructure has emerged as the dominant platform for modern computing. This transition offers unprecedented elasticity, operational simplicity, and access to a vast ecosystem of managed services. However, it also presents substantial cost challenges. Recent industry reports show that optimizing cloud costs has remained one of the top organizational priorities for the past seven years [2], and 69% of IT leaders report public cloud cost overruns that negatively impact other budgets [5]. As illustrated in Figure 1.1, Gartner forecasts [6, 7, 8, 9, 10, 11, 12] indicate that end-user spending on public cloud services has increased at an average annual rate of 23% from 2018 to 2024. Reducing this growing cost burden is becoming an increasingly critical concern for cloud users.

While prior research [13, 14, 15, 16, 17] has extensively studied cost-optimized virtual machine selection for computational workloads in the public cloud, storage selection has received comparatively less attention. However, recent industry findings underscore the growing importance of addressing cloud storage costs. More than two-thirds of organizations allocate more than a quarter of their cloud budgets to storage services [18], and 66% of them report exceeding their cloud storage budgets [19].

Although organizations aim to reduce cloud storage costs, selecting cost-efficient storage strategies remains challenging. This difficulty comes from the complex landscape of resource options, each with varying pricing and performance characteristics. In addition, storage performance often depends on workload access patterns, further complicating decision making. As a result, organizations frequently adopt suboptimal deployment strategies, such as lift-and-shift migrations from on-premises systems or the uniform selection of storage configurations across all virtual machines. These strategies often rely on predefined "storage-optimized" instances without tailoring the configurations to specific workload needs.

These challenges are further amplified in today's cloud landscape, where applications are increasingly geo-distributed across multiple regions or cloud providers. Accessing data across these boundaries has become common, driven by needs such as fault tolerance, data locality, and regulatory compliance. However, multi-region and multi-cloud deployments introduce significant costs, particularly due to high data egress fees and variability in performance guarantees. In practice, organizations replicate entire data lakes across regions or clouds to meet latency requirements or access remote data directly for simplicity. Both approaches result in increased

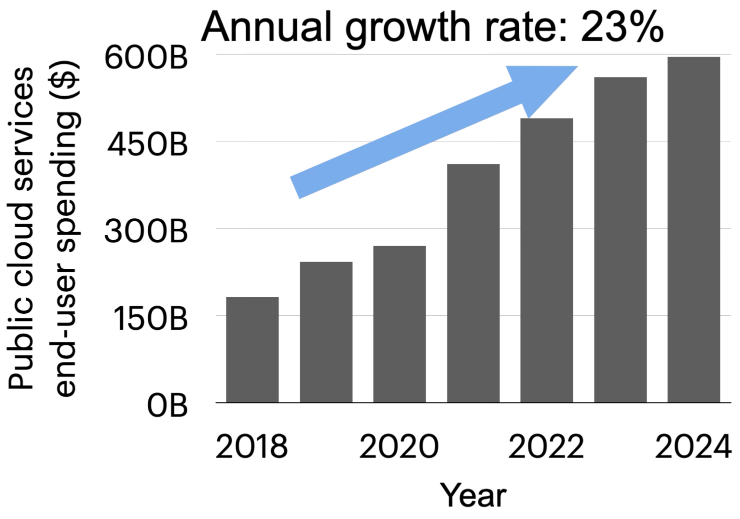


Figure 1.1: Annual end-user spending on public cloud services has grown by an average of 23% per year from 2018 to 2024, according to Gartner reports. This consistent growth highlights that controlling cloud expenses is already a major concern for users and is expected to become even more critical in the coming years.

operational costs and degraded performance as a result of higher latencies.

Although many research has explored the design and optimization of storage systems in on-premises and research datacenter settings, these techniques do not transfer directly to public cloud environments. Cloud deployments are similarly difficult, but fundamentally different, because public cloud providers enforce strict service level agreements (SLAs) and pricing models that limit traditional tuning and resource control. In this context, cost is not simply a function of storage capacity or usage volume. Instead, it relies on configuration decisions that interact with dynamic workload behavior and billing structures.

This dissertation addresses these emerging challenges by rethinking how to design and operate storage systems in the public cloud. It focuses on improving cost-efficiency of storing and accessing data in public clouds, by exploiting the diversity of cloud resources and incorporating runtime workload monitoring. The dissertation presents systems that automatically select heterogeneous combinations of compute and storage types, adjust cache capacity dynamically to reduce remote data access costs, and apply predictive techniques such as prefetching to reduce latency without incurring unnecessary data transfer expenses. Through these contributions, the dissertation provides new insights into building cloud-native storage systems that balance performance with cost-awareness.

1.1 Thesis statement

This dissertation focuses on reducing the following two major categories of storage-related costs while maintaining the performance requirements: (1) storage cluster deployment costs, which includes the costs of provisioning virtual machines and storage volumes among storage types with differing cost-performance trade-offs, and (2) cross-cloud/region data access costs, which arise from accessing data across geographically distant regions or across different cloud providers, and are often dominated by data egress charges and high-latency penalties.

To address these challenges, we identify three public cloud-aware design principles that guide cost-efficient storage system design in public clouds:

- **Exploiting resource diversity:** Public clouds offer a wide variety of VM and storage options with diverse cost-performance characteristics. Systems can achieve cost-efficiency by automatically selecting and combining heterogeneous resources.
- **Leveraging elasticity:** Public clouds allow on-demand provisioning and releasing of resources. Systems can benefit from dynamically adjusting system capacity based on current resource availability and budget constraints.
- **Exploiting workload-awareness:** Cloud workloads often exhibit time-varying data access patterns. Systems should incorporate runtime monitoring and adapt to changing workload behavior to maintain efficiency.

Grounded in these principles, this dissertation presents our works that collectively demonstrate and support the following thesis statement:

Thesis Statement: By leveraging the elasticity and diversity of public cloud storage resources in combination with real-time workload monitoring, it is possible to reduce storage deployment costs and cross-region/cloud data access costs.

We demonstrate this thesis through the design and evaluation of three systems, each addressing a key challenge in public cloud storage and caching, using automated heterogeneous resource selection, adaptive caching, and cost-aware prefetching.

(i) ***Mimir: finding cost-efficient storage configurations in the public cloud*** **(Chapter 2])** Provisioning cost-efficient storage clusters in public clouds is challenging due to the wide range of available block storage options (e.g., remote / local SSDs, HDDs) with varying cost-performance trade-offs and performance SLAs. Organizations often resort to naive shift and lift deployments that ignore this diversity, leading to inefficient spending. Mimir is a system that automatically selects cost-efficient storage configurations by leveraging both workload profiling and detailed resource benchmarking. Specifically, it configures virtual storage clusters with a mix of storage types and provisioning parameters. Mimir uses a two-stage pipeline: first, it profiles workloads to understand access behavior and latency sensitivity; then, it benchmarks cloud storage types to estimate cost and performance under those behaviors. By combining this information, Mimir optimizes cluster configurations to meet performance goals while minimizing costs. Evaluation of Apache BookKeeper storage systems shows that Mimir reduces storage deployment costs by up to 81% compared to state-of-the-art approaches.

(ii) ***Reducing cross-cloud/region costs with the auto-configuring Macaron cache*** **(Chapter 3])** Cross-cloud and cross-region workloads incur significant egress costs and access latency, yet current solutions either replicate all data (expensive) or access it remotely (slow). While cloud providers offer caching services, these require manual configuration

and often rely on expensive DRAM. This work introduces Macaron, an auto-configuring cache system that minimizes cost for remote object storage access in geo-distributed workloads. Macaron is built around a key insight: cache size in the cloud is cost-constrained, not hardware-limited. As such, Macaron dynamically adjusts both cache size and storage tiering (DRAM vs. object storage) in response to access patterns. A lightweight miniature simulation estimates cache miss rates and latency, enabling cost-performance tradeoff decisions. Macaron addresses three core challenges derived from real-world traces from IBM, Uber, and VMware: (1) bridging between full replication and naive remote access with an auto-tuned cache; (2) supporting large cache sizes using cheap storage to offset high egress cost; and (3) dynamically reconfiguring the cache to match evolving access patterns. Compared to alternatives, Macaron achieves up to 81% cost reduction versus full replication, 66% over provider caching services, and only 9% higher cost than oracular caching (with perfect future knowledge).

(iii) Macaron+: Cost-aware cross-region cache prefetching **(Chapter 4)** Macaron+ extends the Macaron caching system to improve latency in geo-distributed workloads by prefetching blocks based on access pattern similarity between objects. Existing prefetching techniques are ineffective for cloud object storage, which lacks a global address space and exhibits high cold miss ratios. To overcome this, Macaron+ leverages the insight that early access patterns can predict future behavior, using a sparse random projection to embed per-object access patterns and a lightweight MLP model to forecast future embeddings. It then uses approximate nearest neighbor search to identify similar objects and selects blocks to prefetch based on aggregate access frequencies. Evaluation on Uber's Presto cluster traces shows that Macaron+ reduces average latency by 28% while increasing cost by 103% compared to Macaron, outperforming other existing online prefetchers. Ablation studies show further improvements by constraining prefetching to well-observed objects or those in similar directory paths. Compared to Macaron+ without prediction, Macaron+ offers a more cost-efficient latency reduction and approaches the performance of an oracular offline version with perfect knowledge.

1.2 Summary of contributions

The core contributions of this dissertation are as follows:

Mimir:

- Show that finding cost-optimal virtual storage cluster configurations requires considering diverse storage volume types and configurations.
- Describe the architecture and algorithms that allow Mimir to find cost-effective virtual storage cluster configurations for a distributed storage backend.
- Demonstrate that Mimir can effectively explore AWS's diverse block storage offerings, reducing cost by up to 81% relative to state-of-the-art approaches.
- Demonstrate that Mimir can be used as part of a dynamic reconfiguration system to reduce cost by 74% for diurnal workloads.

Macaron:

- Show that finding cost-optimal virtual storage cluster configurations requires considering diverse storage volume types and configurations.
- Describe the architecture and algorithms that allow Mimir to find cost-effective virtual storage cluster configurations for a distributed storage backend.
- Demonstrate that Mimir can effectively explore AWS's diverse block storage offerings, reducing cost by up to 81% relative to state-of-the-art approaches.
- Demonstrate that Mimir can be used as part of a dynamic reconfiguration system to reduce cost by 74% for diurnal workloads.

Macaron+:

- Analyze real-world cloud storage workloads and derive key design rationales for prefetching data blocks.
- Propose a cost-efficient prefetching technique that leverages access pattern similarity between objects to guide block prefetching.
- Explore multiple variants of the Macaron+ prefetcher, characterizing the cost–latency trade-off space.
- Demonstrate that Macaron+ achieves 13–61% cost reduction for the same latency target compared to existing approaches, and identify opportunities for further improvement.

1.3 Outline

This dissertation is organized as follows. Chapter 2 presents Mimir [20], a system for automatically selecting cost-efficient storage configurations for cloud-based storage clusters by analyzing workload characteristics and resource trade-offs. Chapter 3 introduces Macaron [21], an auto-configuring cache that reduces cross-region/cloud access costs by dynamically adapting to workload behavior and cloud resource pricing. Chapter 4 extends this work with Macaron+, a cost-aware prefetcher that leverages block-level access pattern similarity and runtime prediction to proactively reduce access latency and egress costs. Finally, Chapter 5 concludes with a summary of key findings and a discussion of future research directions.

Chapter 2

Mimir: Finding cost-efficient storage configurations in the public cloud

Companies are increasingly moving data-heavy applications to the cloud, often replicating on-premises implementations of integrated data processing and storage backend systems on cloud instances. While researchers have introduced and studied effective approaches for auto-selecting cost-optimized VM instances for computation work [13, 14, 15, 16, 17], less attention has been paid to storage selection. For cold storage, there is usually a clear option (e.g., S3 in AWS or Blob Storage in Azure). For performant storage needs, however, the set of block storage volume types is increasingly diverse in storage characteristics, SLAs and cost structures. Selecting the most cost-effective virtual storage cluster (VSC) configuration for a given data-heavy application deployment is likely beyond all but the most expert user.

Commonly, storage backends (e.g., distributed file systems, key-value stores) are built for use with block storage volumes providing traditional SSD or HDD interfaces. Selecting storage hardware for on-premises deployments is challenging [22, 23, 24], given the wealth of options. The challenge in cloud deployments is similarly difficult, but differently so because of cloud SLA and cost structures. Using AWS as a concrete example, there are three block storage volume types: local-SSD associated with a compute instance, remote-SSD that can be attached to any VM instance, and remote-HDD that can be attached to any compute instance. Making matters worse, each type has multiple options with different costs and different SLA structures regarding cost as a function of performance and capacity required. For example, options include charging per-GB with a fixed budget of IOPS per-GB, providing a specific capacity and performance for a given cost, or charging for a performance budget of MiB/s per-TB. Each customer is best served by a different option, and the most cost-effective may be a mix of options.

Figure 2.1 illustrates the need to consider many volume types and configurations in selecting a VSC configuration. For each of the three workloads on a distributed storage backend, it shows the cost for the best VSC configuration choice under each of three constraints: considering only local-SSD volume types, only remote storage (EBS) volume types, and arbitrary mixes of both volume types. The most cost-effective configuration is used in each case. We note that: (1) the best single-type choice differs across workloads, and (2) cost is sometimes minimized by mixing volume types.

In this chapter, we present **Mimir**, a tool for finding a cost-effective set of instances, vol-

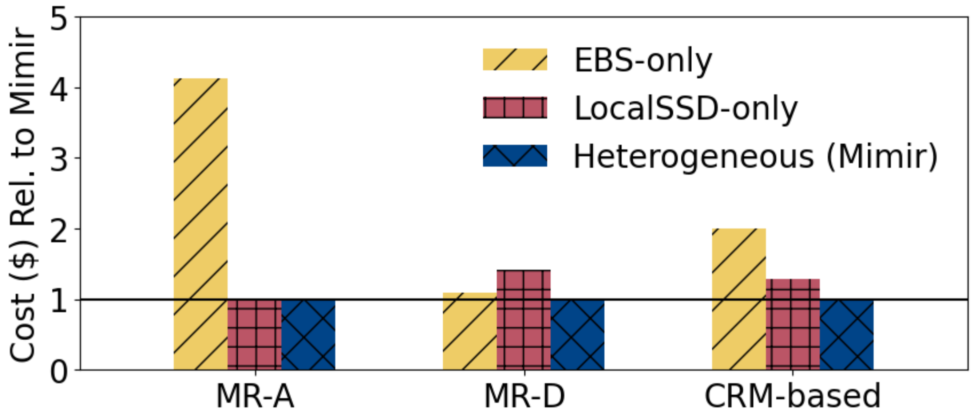


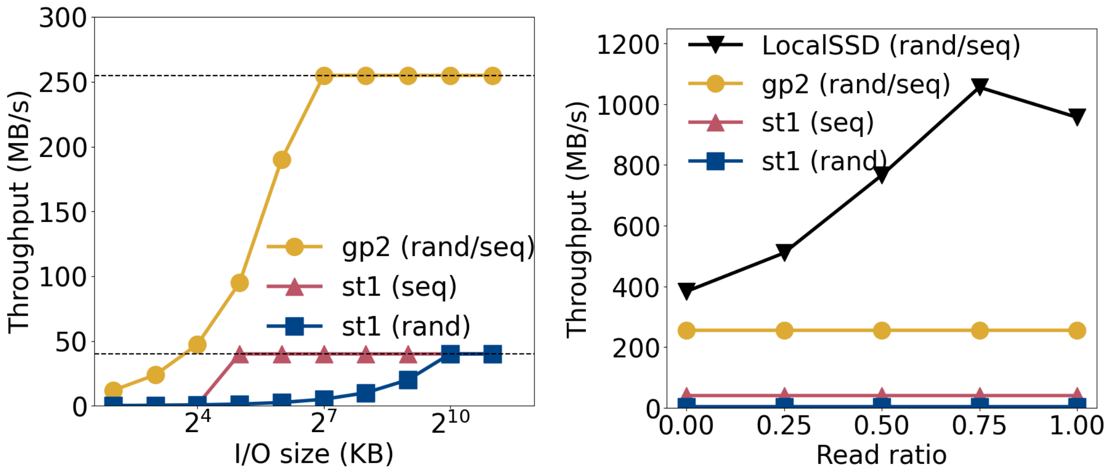
Figure 2.1: No volume type is most cost-efficient for every workload, and a mix of volume types may be the most cost-effective option. MR-A, MR-D, and CRM-based workloads respectively represent high-throughput, low-throughput, and mixed workloads, and the detailed workload characteristics are described in §2.4.

ume types and volume configurations for a distributed storage backend used by a data-heavy application workload. Given high-level workload specifications and performance requirements, as might be produced by profiling an operational version of the system (whether on-premises or using an over-provisioned manual configuration), Mimir considers potentially heterogeneous VSC configurations as shown in Figure 2.1.

Mimir casts VSC configuration selection as an optimization problem, like most prior tools for automated resource selection. Central to how Mimir achieves its goals is predicting the resources required for the given workload, including both the I/O throughput of the access pattern and the compute and memory needs of the storage software. Using predicted resource requirements and analytically-formulated price-performance cost models of public cloud resources, Mimir determines cost-efficient VSC configurations using dynamic programming. This is in contrast to predicting workload performance for a specific instance type like previous works [17, 25, 26], which allows Mimir to explore heterogeneous VSC configuration options, and find good VSC configurations for workloads composed of multiple access patterns.

Mimir focuses on cost-efficient resource selection for given workload characteristics and requirements, and shows that such resource selection must consider diverse volume types and configurations to minimize costs. In some cases, the workload characteristics and requirements can be determined just once for a stable workload or when provisioning for peak requirements. In other cases, adjusting allocated resources dynamically to match observed variations in the workload can bring further cost benefits. For such cases, Mimir can be used as the resource selection component (replacing less-effective traditional selection components) in a system that monitors the workload variations, intermittently invokes Mimir to suggest new VSC configurations, and enacts configuration changes (and data movement) if the project savings exceeds the cost of changing.

To evaluate Mimir, we used Apache BookKeeper as the distributed storage backend driven by each of two key-value workloads based on discussions with engineers of a top customer relationship management (CRM) service and six workloads based on key-value workloads described by Meta [1]. Our results show significant cost savings arising from Mimir's approach and its



(a) by I/O unit size (b) by read ratio

Figure 2.2: Performance characteristics of public cloud storage volume types by (a) I/O unit size and (b) workload read ratio. In (a), both volume types have throughput limits defined by AWS (horizontal lines), and we used a read-only workload.

ability to consider diverse volume types. For example, compared to a state-of-the-art approach considering only EBS volume type and configurations, Mimir reduces cost by up to 81%. More generally, Mimir consistently and quickly finds cost-effective VSC configurations.

Contributions. We make four primary contributions: (1) We show that finding cost-optimal VSC configurations requires considering diverse volume types and configurations. (2) We describe the architecture and algorithms that allow Mimir to find cost-effective VSC configurations for a distributed storage backend. (3) We demonstrate that Mimir can effectively explore AWS's diverse block storage offerings, reducing cost by up to 81% relative to state-of-the-art approaches. (4) We experimentally demonstrate that Mimir can be used as part of a dynamic reconfiguration system to reduce cost by 74% for diurnal workloads.

2.1 Cloud storage configuration challenges

This section motivates the need for tools like Mimir that automate the configuration of virtual storage clusters in public clouds. First, we examine the diversity in performance and cost characteristics of different cloud storage volume types, which complicate manual configurations (§2.1.1). Second, we examine how the characteristics of cloud storage volume types affect the cost of deploying a scalable storage service in the public cloud (§2.1.2).

2.1.1 Public cloud storage characteristics

It is crucial to understand the characteristics of public cloud storage types in order to configure storage systems atop virtual storage cluster in a cost-efficient manner. Mimir formulates the price and performance cost model with the analyzed storage characteristics in this section.

One of the public cloud storage types we use to build volumes in this work is *block storage*, such as AWS Elastic Block Store [27], Azure Disk Storage [28], and GCE Persistent Disk [29]. On AWS, there are five different block storage types: local NVMe SSD, remote SSD (gp2, io1), and remote HDD (st1, sc1). Local SSD is served as an SSD locally attached to some instance types, such as i3, c5d, and m5d. It delivers high performance with low latency, but the attached volume capacity is fixed, and it can be an expensive option for data that does not require high throughput. Unlike local SSD, users can attach remote storage volumes (EBS) to the machines they need. The performance of remote storage types is defined as SLAs by the public cloud providers. Though AWS has recently introduced support for EBS Multi-Attach, allowing a single io1 volume to be attached to multiple instances, we assume that a single EBS volume can only be attached to one instance since this service is currently available in a limited number of regions and works only for io1 volumes. For instance, AWS currently offers gp2 volumes at 3 IOPS per GiB of provisioned capacity, while it provides 40 MiB/s per TiB of provisioned capacity for st1 volumes.

Figure 2.2 illustrates the characteristics of 1 TiB of gp2 volume and 1 TiB of st1 volume, in which the performance of each volume is 3000 IOPS and 40 MiB/s, respectively, and local SSD attached at i3.xlarge. We generated the test workloads with the `fio` benchmark [30] varying the access pattern (random/sequential), read ratio, and I/O unit size. For Figure 2.2(a), we used a read-only workload to evaluate the performance characteristics.

Figure 2.2(a) shows *how the performance characteristics according to the I/O unit size and access pattern are different for each storage type.* Because gp2 performance is defined in IOPS, as the I/O unit size increases, the throughput of gp2 also increases up to 250 MiB/s, which is the maximum single gp2 volume throughput limited by SLA. In the case of st1, performance is defined in MiB/s, but shows lower throughput for the workloads with random access patterns and I/O units less than 1 MiB [31]. st1 has a throughput limit at 40 MiB/s for 1 TiB st1 volume, in which the limit can be up to 500 MiB/s for the larger st1 volume. The performance of gp2 is the same for both random and sequential data access patterns, while st1 shows better performance for sequential data access than random access.

Figure 2.2(b) shows *how read ratio affects each volume type's throughput differently.* Throughput of EBS volumes is not affected by the read ratio of the storage workload, as the read ratio is not included in their performance SLAs. The local SSD, however, shows much higher throughput than EBS, and the throughput is affected by the read ratio, while it is not affected by the I/O unit size for requests larger than 32KB.

We have measured the local SSD performance of all the machines we used as candidates of the cost-optimal VSC. The local SSD performance profiling is not a consuming process in terms of time or cost because profiling needs only be performed once. There are other volume types (`io1`, `sc1`) we also considered, but we omit them for brevity.

2.1.2 Apache BookKeeper use case

Next, we give a motivating example demonstrating the potential savings of careful machine configuration for an application. Inspired by discussions with engineers from a large customer relationship management (CRM) company shifting from on-premises to cloud, we look at Apache BookKeeper. Apache BookKeeper [32] is a storage system designed for high scalability, fault-tolerance, and low latency. It stores data as streams of log entries in sequences called ledgers,

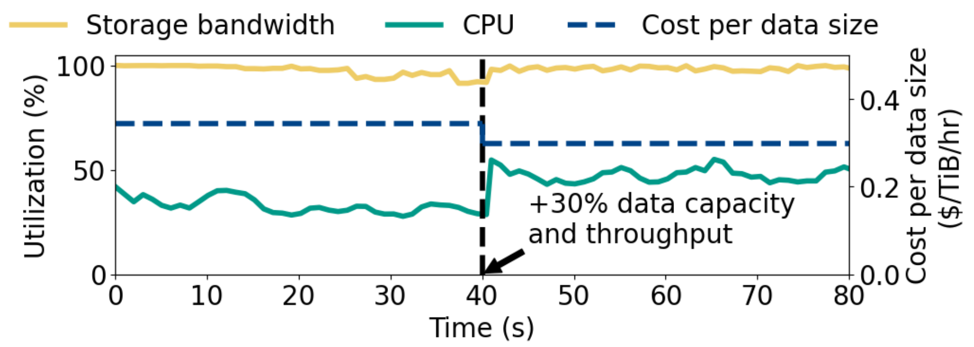


Figure 2.3: Reducing the cost per data size by exploiting heterogeneous machine allocation. When a storage server uses only local SSD, CPU is underutilized. Attaching an EBS volume can store 30% more data, paying only 12% additional cost.

and the ledgers become immutable once the ledger is closed. The primary data access pattern of Apache BookKeeper's storage server is sequential writes and random reads.

We can reduce cost by exploiting heterogeneous resource allocations. Figure 2.3 shows resource utilization of Apache BookKeeper's storage server running on i3.2xlarge, with a 1.9 TiB local SSD. The workload is write-only and requires 1.8 TiB of data capacity and 360 MiB/s of write throughput at the beginning. After 40 seconds, we increase both requirements by 30%, so the workload's required throughput per TiB remains the same.

For the first 40 seconds, 67% of CPU is idle on average while the storage bandwidth of the local SSD is fully utilized. After 40 seconds, the simplest way to satisfy both increased requirements is to provision another i3.2xlarge which doubles the cost. As Figure 2.3 shows, however, attaching a 600GiB EBS volume to the original instance instead of provisioning a new instance allows us to store 30% more data while paying only 12% additional cost. It also reduces the cost per data size by 15%, and this heterogeneous allocation allows the workload to utilize 15% more idle computing power.

Therefore, it is crucial to accurately predict how many resources (e.g., CPU, memory, storage bandwidth, etc.) are required for the given workload characteristics to configure cost-efficient heterogeneous virtual storage clusters (§2.3.3). Also, though we restrict to a single instance type and one workload characteristic in this example, if we consider more instance types and workload characteristics, the gain from the heterogeneity compared to the homogeneous allocation increases (§2.4.4).

2.2 Prior work

In this section, we discuss previous research on the automatic provisioning of public cloud resources and predicting application performance on virtual machines in public clouds.

Configuring storage and VMs in public cloud. Many previous works [16, 33, 34, 35, 36] aim to optimize virtual cluster configuration in public clouds for various workloads. Some studies [17, 25, 37] find near-optimal cloud storage and VM configuration for data analytics workloads, guaranteeing performance and minimizing cost. However, the workloads we target have different nature from the data analytics workloads, e.g., workloads are long-running rather than transient and cannot classify data into input/output and intermediate data, which are common in data analytics applications. Therefore, our research cannot be solved in the same way as previous studies. For example, in the case of data analytics workloads, to reduce the overall cost, the trade-off between using expensive resources for a short duration or simply using cheaper resources should be considered, but our problem does not have this nature.

OptimusCloud [26] jointly optimizes database and VM configurations to find cost-efficient VSC configurations for distributed databases. We consider OptimusCloud as the state-of-the-art to compare with Mimir, but OptimusCloud only considers the EBS volume type, which we show could be a costly configuration compared to a VSC using both local SSD and EBS volume types (§2.4.4).

Performance prediction on VMs. Numerous previous systems [38, 39, 40, 41, 42, 43, 44, 45] studied the method of predicting workload performance on VMs. PARIS [46] uses hybrid offline/online data collection and trains a random forest model to predict the workload performance on VMs. Ernest [47] predicts the performance of large-scale data analytics workloads using statistical modeling. Auto-configuration systems [25, 26] also predict the workloads' performance on VMs using machine learning techniques, such as collaborative filtering and gradient boosting tree.

In contrast, our approach predicts resources required for the given workload performance instead of predicting workload performance on VMs. By predicting resource requirements and knowing the performance SLAs given by the cloud providers, we mathematically formulate a linear programming problem to find the cheapest VSC configuration that has the necessary resources. Still, we can use similar data profiling techniques and prediction approaches that previous works proposed, such as gradient boosting tree.

2.3 Mimir design

Figure 2.4 shows the workflow of Mimir. First, Mimir takes as input information about multiple workloads' characteristics (§2.3.1). Storage systems can store data for different workloads, and each workload can have a different data access pattern, such as the data request rate, data access locality, and read/write request ratio. Then, our *Resource profiler* profiles each workload and collects data on how many resources are required to run each workload cost-efficiently on the machines in the cloud (§2.3.2). Using the collected data, the *Resource predictor* learns how to convert each workload specification into the right size of the container to run (§2.3.3). As demonstrated in §2.1.2, the heterogeneous single machine configuration is important for the entire VSC's cost-efficiency, and in Mimir, to leverage the heterogeneous single machine configuration, we run multiple storage servers on a single machine by deploying each server in a Docker container, which guarantees the isolation of allocated resources for each storage server. Lastly, the *VSC Cost optimizer* uses the *Resource predictor* and the cost model of public cloud resources to find the cost-efficient VSC configuration of the distributed storage system (§2.3.4). We assume that the workload characteristics can be profiled (or are known) by Mimir users, and Mimir uses the profiled information as input to its optimization process. One advantage of this design is that any storage system capable of measuring the necessary resource utilization that

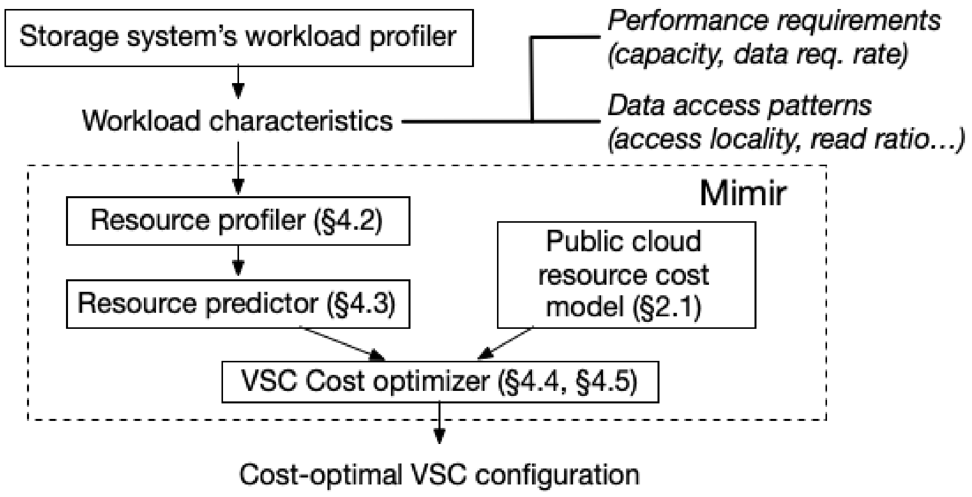

Storage system's workload profiler
Workload characteristics
Performance requirements (capacity, data req. rate)
Data access patterns (access locality, read ratio...)
Mimir
Resource profiler (§4.2)
Resource predictor (§4.3)
Public cloud resource cost model (§2.1)
VSC Cost optimizer (§4.4, §4.5)
Cost-optimal VSC configuration

Figure 2.4: Mimir's workflow for optimizing the price of public cloud resources. Mimir profiles the given workloads and learn how many resources (e.g., CPU, memory) are required. The VSC Cost optimizer uses this trained module and cost model of public cloud resources to find the cost-efficient VSC configuration.

Mimir uses for optimization can employ Mimir as a resource auto-selector. In our evaluation, we evaluate Mimir only on Apache BookKeeper, but we left extending our evaluation to other storage systems as a future work.

2.3.1 Input: profiled workload characteristics

Mimir takes workload specifications as input. Table 2.1 shows the five attributes we use to describe workload characteristics in Mimir. They are divided into two categories: performance requirements and data access patterns.

Data capacity and data request rate are the attributes of the performance requirements that should be satisfied for the given workloads. Performance requirements are also used as profiling knobs and are proportional to the size of *workload fraction*, i.e., a subset of data and data accesses to the subset of data. For example, if a user defines a workload with 1 TiB of data capacity and 10K QPS (queries per second) of data request rate, we expect 3K QPS is required for the 300 GiB of the given workload's data. This can be achieved by load balancing for a clustered storage system, which has been extensively studied [48, 49, 50].

The attributes of the data access pattern describe the behavior of the workloads: temporal/spatial data access locality, read/write ratio, and distribution of object size stored in the storage backend. Unlike performance requirements, Mimir expects the attributes to remain the same for *workload fractions* and uses this assumption in *Resource profiler* to generate a set of *workload fractions* to profile. As future work, Mimir will monitor the actual characteristics of the *workload fraction* on runtime and give feedback to these assumptions.

Several studies [26, 51] have supported the elastic rightsizing of cloud resources by predicting workload characteristics or in a reactive manner. But elasticity is orthogonal to our work. Instead, we focus on finding the potentially heterogeneous cost-efficient VSC configuration for

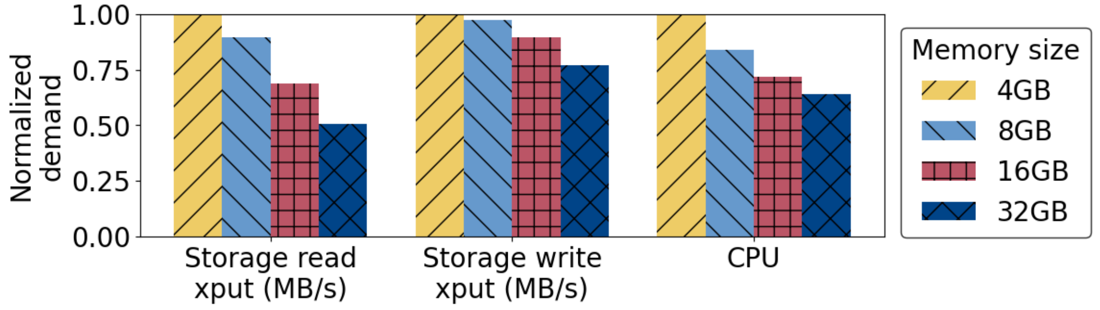


Figure 2.5: Cost-efficient container sizes for the same workload with different memory sizes. The resource profiler profiles each workload with different memory sizes to collect different resource/performance demands.

the mixture of static workloads with different characteristics. Still, we show the extensibility of our tool for dynamic workloads in §2.4.6.

Type	**Attribute**	**Description**	**Units**
Performance requirement (Profiling knobs)	data capacity	total size of data stored in the storage system	GB
	data request rate	rate of read and write requests arrive at the storage system	QPS
Data access pattern	data request size	mean or distribution of the requested data size	Byte
	read/write ratio	ratio of the read and write request rates	
	access locality	pattern of data access locality	

Table 2.1: The workload characteristic attributes. The performance requirement attributes are the knobs used by Mimir in profiling to get multiple profile data points, while the data access pattern attributes are not changed.

2.3.2 Resource profiler

Mimir's goal is to allocate sufficient resources per storage server container (*ContainerSpec*) to satisfy the provided workload performance requirements, while remaining frugal to reduce costs. However, many factors make it challenging to compute the right size of *ContainerSpec* for the arbitrary workload specification analytically. Read/write amplification inherent in the storage servers depends on the implementation and data access pattern; memory size of the storage servers and read/write ratio of workloads affect the necessary storage throughput and computing power to meet the performance requirements. None of these factors can be precisely formulated without the storage system experts and should be reformulated for every storage system to be used. Instead of formulating the cost-efficient size of *ContainerSpec*, Mimir, therefore, profiles and collects data using the *Resource profiler* and predicts the optimal size of containers using the *Resource predictor* trained with the collected data.

Algorithm 1 Profiling logic of the *Resource profiler*

```
1:  $W$: Input workload characteristic
2:  $BM$: Benchmark machine
3:  procedure PROFILE($W$)
4:      $p \leftarrow$ MEASUREMAXPERFORMANCE($W$, $BM$)
5:      $S \leftarrow$ WORKLOADFRACTIONSETTOPROFILE($W$, $p$)
6:      $D \leftarrow \{\}$
7:      for $wf$ in $S$ do
8:          $u \leftarrow$ MEASURERESOURCEUTILIZATION($wf$, $BM$)
9:          while $\neg$ ISCONTAINERRIGHTSIZE($wf$, $u$) do
10:             $u \leftarrow$ UPDATECONTAINERSPEC($wf$, $u$)
11:         end while
12:         $D[wf] \leftarrow u$
13:     end for
14:     return $D$
15: end procedure
```

Resource profiler **overview.** The *Resource profiler* runs a storage workload with the given workload characteristics on a benchmark machine to collect the data of the cost-efficient size of *ContainerSpecs*. When profiling, Mimir uses the performance requirement attributes as knobs to get multiple data points. The attributes of *ContainerSpec* we use are: the number of CPUs, memory size, storage bandwidth, storage capacity, and network bandwidth. To get enough profiling data, we chose i3.4xlarge of AWS as the benchmark machine, which has the local SSD with the highest single-storage performance (1900GB NVMe SSD), and sufficient memory and computing resources to profile our evaluation workloads.

There are multiple suitable *ContainerSpecs* for a single workload specification. For example, a read-intensive workload with a high degree of data access locality requires less storage volume performance and computing power with a larger memory size because of memory caching. Figure 2.5 shows how different the required resources are according to the memory size, even for the same workload specification. Thus, the *Resource profiler* tests different memory sizes to account for multiple *ContainerSpecs* during optimization.

Resource profiler **logic.** The *Resource profiler* first measures the maximum performance p of the workload on the benchmark machine with the given data access pattern (Algorithm 1, Line 4). Then, it generates a set of N different *workload fractions* (i.e., workloads with $1/N * p, 2/N * p, ..., N/N * p$ performance requirements and given data access pattern) to be profiled (Algorithm 1, Line 5). We used $N = 10$ in our experiments. For each *workload fraction*, *Resource profiler* finds the right container size (Algorithm 1, Line 7-13). It first measures the average resource utilization while running the *workload fraction* on the benchmark machine. However, the container allocated with the average resource utilization may not meet the performance requirements, or it may have been allocated more resources than necessary. So, it iteratively updates the candidate container size by measuring the storage server performance and resource utilization in the container until it finds the cost-efficient container size. Detailed rules for this iterative updates are described below.

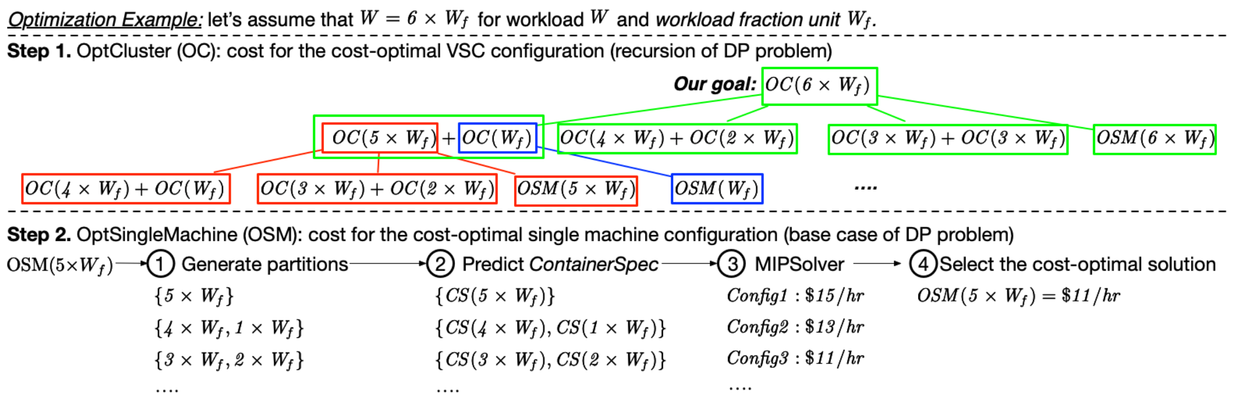

Optimization Example: let's assume that $W = 6 \times W_f$ for workload W and workload fraction unit W_f.
Step 1. OptCluster (OC): cost for the cost-optimal VSC configuration (recursion of DP problem)
Our goal: $OC(6 \times W_f)$
$OC(5 \times W_f) + OC(W_f)$
$OC(4 \times W_f) + OC(2 \times W_f)$
$OC(3 \times W_f) + OC(3 \times W_f)$
$OSM(6 \times W_f)$
$OC(4 \times W_f) + OC(W_f)$
$OC(3 \times W_f) + OC(2 \times W_f)$
$OSM(5 \times W_f)$
$OSM(W_f)$
....
Step 2. OptSingleMachine (OSM): cost for the cost-optimal single machine configuration (base case of DP problem)
$OSM(5 \times W_f)$
1 Generate partitions
2 Predict ContainerSpec
3 MIPSolver
4 Select the cost-optimal solution
$\{5 \times W_f\}$
$\{4 \times W_f, 1 \times W_f\}$
$\{3 \times W_f, 2 \times W_f\}$
....
$\{CS(5 \times W_f)\}$
$\{CS(4 \times W_f), CS(1 \times W_f)\}$
$\{CS(3 \times W_f), CS(2 \times W_f)\}$
....
Config1 : \$15/hr
Config2 : \$13/hr
Config3 : \$11/hr
....
$OSM(5 \times W_f) = \$11/hr$

Figure 2.6: Example of the *VSC Cost optimizer*'s optimization algorithm. It uses dynamic programming to break the optimization problem into smaller problems (OPTCLUSTER), and mixed-integer programming to solve the base cases (OPTSINGLEMACHINE).

Rules for updating *ContainerSpec*. If the current container size satisfies the workload requirements and the average utilization values of some resources are less than the over-provisioning threshold (we used 80%), it is considered those resources are allocated more than necessary. So the profiler reduces their resource allocations. If it does not satisfy the workload requirements, Mimir increases the allocation of the resources with the average utilization higher than the under-provisioning threshold (we used 90%), judging them as bottleneck resources. We used *docker stats*, *iostat*, and *sysfs network interface statistics* to measure CPU, storage/network bandwidth utilizations.

The thresholds we use in this logic can control the cost-efficiency of the container sizes that Mimir selects for the storage servers. For example, if we use a small over-provisioning threshold, Mimir is likely to allocate more resources to the storage servers than they actually require, and vice versa. If we use a smaller under-provisioning threshold, the storage server configuration is more tolerant to a slight increase in workload performance requirements.

2.3.3 Resource predictor

Based on the data profiled by the *Resource profiler*, Mimir predicts the cost-efficient size of containers for the given workload characteristic. Currently, Mimir provides an implementation using interpolation, but other prediction models, such as a gradient boosting tree [52], could be used instead.

The *Resource profiler* profiled N different *workload fractions*, in which each requires the performance of the $i/N \times p$ (where $i = 1 \cdots N$ and p is the maximum performance on the benchmark machine). Thus, the *ContainerSpecs* for the workload requiring performance less than p can be computed using interpolation. As we noted, we use the large enough instance types as a benchmark machine (i.e., the machine that can profile up to large p) in order to use the interpolation for the *workload fractions* that require high performance.

The interpolation approach allows accurate prediction of the right size of the *ContainerSpecs* as the *Resource profiler* profiles enough data. However, this approach requires a profiling step

Algorithm 2 Optimization algorithm of *VSC Cost optimizer*

```
1:  $W$: Profiled workload characteristics
2:  procedure OPTCLUSTER($W$)
3:      $S \leftarrow$ WORKLOADFRACTIONPAIRS($W$)
4:      $c \leftarrow \infty$
5:      for Pair¡$W_1$, $W_2$¿ in $S$ do
6:          $t \leftarrow$ OPTCLUSTER($W_1$) + OPTCLUSTER($W_2$)
7:          $c \leftarrow \min(t, c)$
8:      end for
9:      return $\min$(c, OPTSINGLEMACHINE($W$))
10: end procedure
11:
12: procedure OPTSINGLEMACHINE($F$)
13:     $D \leftarrow$ WORKLOADFRACTIONPARTITIONS($F$)
14:     $c \leftarrow \infty$
15:     for $d$ in $D$ do
16:         $S \leftarrow$ RESOURCEPREDICTOR($d$)
17:         $c \leftarrow$ min(MIPSOLVER($S$), $c$)
18:     end for
19:     return $c$
20: end procedure
```

when the new workload comes in, which requires additional profiling time and cost, although it is cheaper than the cost savings by our tool as we evaluate in §2.4.7.

2.3.4 VSC Cost optimizer

Mimir uses dynamic programming (DP) to minimize the cost of the virtual storage cluster while satisfying the performance requirements. Figure 2.6 shows an example of how we use recursion in the DP problem (OPTCLUSTER) and solve the base cases (OPTSINGLEMACHINE) using a mixed-integer programming. First, the OPTCLUSTER breaks the problem of finding the cost-efficient VSC configuration that can run the entire workload into the smaller problems of finding the ones that can run the *workload fractions*. In order to solve the base cases of the DP problem, the OPTSINGLEMACHINE searches for the cost-efficient resource configuration of a single machine that can execute each *workload fraction*. Algorithm 2 is the pseudocode of the *VSC Cost Optimizer*. We first explain it using a single workload W as input and then expand to using a mixture of workloads as input (§2.3.5).

Recursion: OptCluster. Mimir first defines the *workload fraction unit* (W_f) of the given workload W, the smallest unit of the workload data stored in the same storage volume. Multiple *workload fraction units* can be stored in the same volume, but a single unit cannot be split. The size of W_f provides the trade-off between the search space size and the optimality of the solution. We empirically evaluated the trade-off and found that using the data size between 50-100 GiB for W_f is generally good in our experiments, e.g., Mimir uses 100 GiB of data size and 1K QPS as W_f for the workload requires 3 TiB of storage capacity and 30K QPS.

The *VSC Cost optimizer* uses DP because the optimization problem has optimal substructure property and overlapping subproblems: *if we found the cost-optimal VSC configuration, then any subcluster of the VSC must have the cost-optimal VSC configuration for the workload fraction running on that subcluster*.

Based on this property, we can argue that the cost-optimal VSC configuration for W is the one that is the cheapest combination of two clusters that one cluster is the cost-optimal for certain amount of *workload fraction* and the other cluster is again the cost-optimal for the remaining *workload fraction* (Algorithm 2, Line 5-9):

$$\begin{aligned}\textsc{OptCluster}(W) &= \textsc{OptCluster}(N \times W_f)\\ &= \min\left(\{\textsc{OptCluster}(i \times W_f) + \textsc{OptCluster}((N-i) \times W_f)\}_{i=1}^{\lfloor N/2 \rfloor},\right.\\ &\qquad\left.\textsc{OptSingleMachine}(N \times W_f)\right)\end{aligned}$$

OPTCLUSTER function can be called recursively, until the input of OPTCLUSTER becomes W_f, and Mimir uses memoization for the computational efficiency (Figure 2.6, Step 1). Note that if the cost-optimal VSC configuration has a single machine, we should find the single machine configuration (OPTSINGLEMACHINE), which is explained below.

Base case: OptSingleMachine. To compute the base cases of the dynamic programming problem, OPTSINGLEMACHINE finds the cost-efficient configuration of a single machine for the given *workload fraction* ($k \times W_f$). ① Within a single machine, there are $\textsc{Partition}(k)$ different ways of distributing the data in storage volumes, where $\textsc{Partition}(n)$ equals the number of possible partitions of *n* (Figure 2.6, Step 2-1). For each partition, ② Mimir generates a set of *ContainerSpecs* by predicting them for each *workload fraction* in the partition using the *Resource predictor* (Figure 2.6, Step 2-2). As each set has the resource requirements of the *workload fractions*, ③ Mimir uses mixed-integer programming (MIPSOLVER) to minimize the price of a single machine under the resource and the performance requirement constraints (Figure 2.6, Step 2-3):

$$\begin{aligned}\underset{Machine,Storage}{\text{minimize}}\quad & \text{Machine[Price]} + \sum_i \text{Storage[i][Price]}\\ \text{subject to}\quad & \sum_{CS} \text{CS[CPU,Mem,...]} \leq \text{Machine[CPU,Mem,...]}\\ & \sum_{CS \in Storage[i]} \text{CS[Storage BW]} \leq \text{Storage[i][BW]}\end{aligned}$$

Lastly, ④ OPTSINGLEMACHINE selects the partition with the smallest return value of MIPSOLVER as the cost-efficient configuration of a single machine (Figure 2.6, Step 2-4).

2.3.5 VSC Cost optimizer: multiple workloads

When the input has more than one workload, running the optimization algorithm using all workloads as input at once can find more (or at least the same) cost-efficient results than using separate virtual storage clusters together after finding the cost-efficient VSC configuration for each, because the search space of the former is the superset of one of the latter. For multiple workloads as input, Mimir can use the same optimization algorithm described in §2.3.4. However, as the number of workloads increases, the complexity of the search space becomes infeasible.

The time complexity (TC) of the recursive loop for the set of workloads $\{W_i\}$, where each workload W_i can be divided into $N_i \times W_{f_i}$, is proportional to the multiplication of N_i.

$$\text{TC of the recursive loop } \propto \prod_i N_i$$

The time complexity of finding the solution for the base cases is proportional to the multiplication of two values: the number of possible partitions of *k*, which is proportional to the exponential function of the square root of *k* [53], and the optimization time of the MIPSOLVER.

$$\text{TC of solving base case } \propto a^{\sqrt{k}} \times T_{\text{MIPSOLVER}} \text{ (a¿1)}$$

Since the total time complexity is the product of these two TCs, it increases exponentially with the number of workloads considered, in which it becomes impractical to use the optimizer even when only three workloads are given as input to the optimizer. To make the time complexity feasible, we use *systematic sampling* and *pairwise workload optimization*.

Systematic sampling: In OPTSINGLEMACHINE, instead of computing MIPSOLVER for all the possible partitions, Mimir samples some of the partitions and find the minimum among them. We used systematic sampling rather than random sampling because the order of the partitions we generated has a property that the adjacent partitions tend to have similar configurations. So selecting every nth partition allows the Mimir to explore various configurations.

Pairwise workload optimization: As the complexity of the search space increases exponentially with the number of workloads, Mimir runs the optimization algorithm for up to two workloads at once for all pairwise workload combinations. For example, if there are six different workloads as input, rather than giving six of them at once to the optimization function, run pairwise optimization $\binom{6}{2}$ times and find the total cost-efficient VSC configuration using them.

Both approaches provide the trade-off between the optimization execution time and the solution's optimality. We could not directly evaluate the trade-off because the search space is infeasible without these approaches. But, we show that Mimir can find cheaper VSC configuration using these approaches when multiple workloads are considered as an optimization input (§2.4.4). We also evaluate how fast our approach finds a cost-efficient VSC configuration compared to the naive search algorithm (§2.4.7).

2.4 Evaluation

This section evaluates Mimir using a CRM-based benchmark and Meta's RocksDB key-value workloads [1]. We first describe our experimental setup (§2.4.1), evaluation benchmarks (§2.4.2), and baselines to which we compare Mimir (§2.4.3). We evaluate Mimir to answer the following questions:

- Can Mimir find a cost-efficient VSC to satisfy the requirements of different workloads? (§2.4.4)
- How effective are key Mimir aspects, including how closely it fits containers to workloads and how important its data partitioning search is? (§2.4.5)
- Can Mimir be used as part of dynamic resizing? (§2.4.6)

Benchmark	**Workload**	**Capacity**	**Req. rate (QPS)**	**Req. size**	**Read req. ratio**	**Access locality**
CRM-based benchmark	H (*High-xput* workload)	3 TiB	9600	64 KB	1.0	Random access
	L (*Low-xput* workload)	3 TiB	2400	64 KB	1.0	
Meta RocksDB benchmark (MR)	A (*Object* in paper)	3 TiB	40K	120 B	0.86	Same as described in paper
	B (*Object_2ry* in paper)	200 GiB	20K	3 B	0.0	
	C (*Assoc* in paper)	600 GiB	80K	17 B	0.81	
	D (*Assoc_2ry* in paper)	400 GiB	40K	5 B	0.0	
	E (*Assoc_count* in paper)	800 GiB	100K	20 B	0.29	
	F (*Non_SG* in paper)	800 GiB	160K	19 B	0.14	

Table 2.2: Benchmarks used to evaluate Mimir. CRM-based benchmark consists of a throughput-intensive (`CRM-H`) and a capacity-intensive workload (`CRM-L`). Meta RocksDB benchmark consists of 6 real-world workloads [1].

- How significant are Mimir's overheads? (§2.4.7)

2.4.1 Experimental setup

We evaluated Mimir in AWS EC2 US-East-1. We used 55 different instance types for the candidate instance types of a VSC configuration. The candidate instance types include all categories of AWS instance types (except ones with GPUs): "general purpose" instances (m5, m5d) and those "optimized" for compute (c4, c5, c5d), memory (r5, r5d), and storage (i3). For the candidate storage types, we used local SSD (i.e., the SSD in i3, c5d, r5d) and EBS volume types (gp2, io1, st1, sc1). We ran our optimization algorithm on a Xeon E5-2670 2.60GHz CPU with 64 GiB DDR3 RAM, using Gurobi 9.0.1 [54]. We used Apache BookKeeper 4.11.0 as the storage backend where our key-value workloads were run.

2.4.2 Benchmarks

We evaluated Mimir using two benchmarks (Table 2.2) on top of Apache BookKeeper: a CRM-based benchmark and a set of workloads similar to the Meta RocksDB key-value workloads described in the paper [1].

Our CRM-based benchmark (**CRM**) is comprised of two workloads: high-throughput workload (`CRM-H`) and low-throughput workload (`CRM-L`). We synthetically generated the CRM-based benchmark based on the discussion with engineers from one of the CRM companies. We used 64 KB of entry size (i.e., data request size) and 2 MB of ledger size, which are the average values the company uses in its BookKeeper storage cluster. Most of their workloads are read-heavy, so the workloads we generated are read-only workloads that read data randomly. For the performance requirements, `CRM-H` and `CRM-L` require 200 MiB/s and 50 MiB/s per TiB of data capacity, respectively, and both have 3 TiB of data. We selected these performance requirements to evaluate how well Mimir finds the cost-efficient VSC configuration for the workloads that have different throughput requirements. The CRM-based benchmark also represents key-value workloads that have large value sizes which are common in real-world [55, 56, 57, 58].

Meta presented detailed characteristics of key-value workloads [1] in their storage cluster, which uses RocksDB as their backend storage engine. Among the three production use cases they described, we selected UDB to evaluate Mimir. Because UDB has six workloads that have different characteristics to each other, we can evaluate Mimir for a complicated realistic benchmark. To evaluate UDB-like workload on Apache BookKeeper, we implemented our own benchmark (**MR**) on BookKeeper that has similar characteristics as Meta described. Our benchmark has the same data size distribution, data access locality and count distribution, and average Put/Get request ratio. We used the same distributions presented by Meta, which are General Pareto Distribution [59] for value size distribution and a power model for access count distribution. We implemented only `Put` and `Get` operations, as semantics of other RocksDB operations deviate significantly from available operations in BookKeeper.

2.4.3 Resource selection baselines

We compare Mimir to three baselines. Whereas Mimir considers all instance and storage types listed in §2.4.1, when selecting VSC resources, each baseline considers only a subset, and the comparisons show the significance of considering heterogeneous VSC configurations for cost-efficiency.

i3.xlarge-only. The simplest way to configure a VSC on the public cloud is to select one instance type and decide the number of instances to provision from measured storage server performance on the selected instance type. However, since this approach has only a single dimension (i.e., the number of machines) in the search space, it ignores too many potential solutions. In our evaluation, we used `i3.xlarge`, because it is categorized as a storage-optimized instance and provides high-performance local SSD.

Mimir-LocalOnly. Another way to configure the VSC is to use only instance types that have local SSDs, including some compute or memory optimized instance types, such as `m5d`, `c5d` and `r5d`. Local SSD provides high storage performance, but can be costly and may provision more IOPS than necessary. We call this constraint `Mimir-LocalOnly`, because we apply all the Mimir's optimizations to finding the cost-efficient VSC configuration (including mixes of instance types) but limit it to considering only local SSD volumes.

Mimir-EBSonly/OptimusCloud-like. Here, VSCs can only use EBS volumes. EBS volumes can persist data independently from the instance status, and users can provision the volume capacity as much as they need. However, if the workload requires high-performance, it can be more expensive than local SSD. As we explained in Section 2.2, OptimusCloud [26] restricts the volume type to EBS volumes because of their persistent nature, but our results show that this approach is often much more costly. Like `Mimir-LocalOnly`, this baseline uses all of Mimir's optimizations while only considering EBS volumes. We use the terms `Mimir-EBSonly` and `OptimusCloud-like` interchangeably.

2.4.4 Cost-efficiency analysis of Mimir

Observation 1: *Mimir finds configurations that are 2–5.3× more cost-efficient than the `OptimusCloud-like` approach, as different workloads benefit from different storage types.*

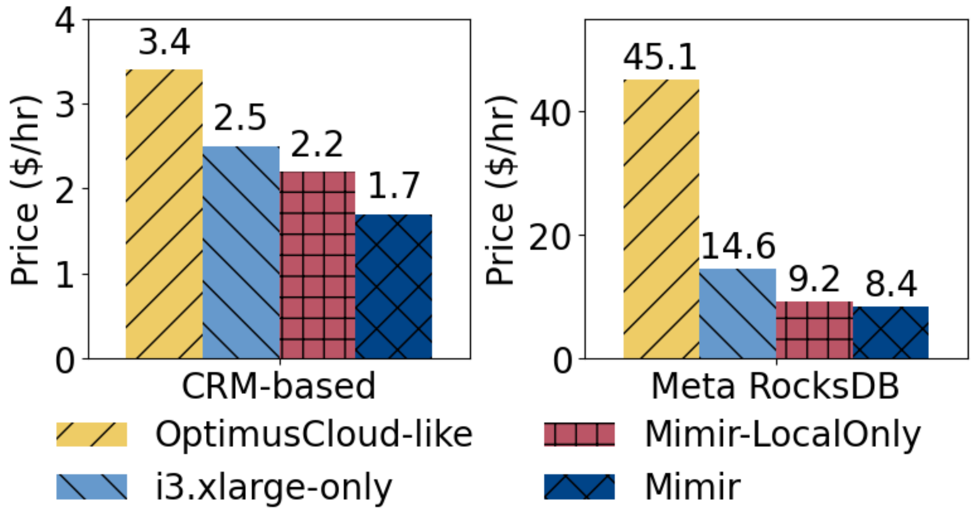


Figure 2.7: The cost-efficiency analysis of the optimization results of the two benchmarks, `CRM` and `MR`. Mimir finds the most cost-efficient VSC configuration compared to the other baselines.

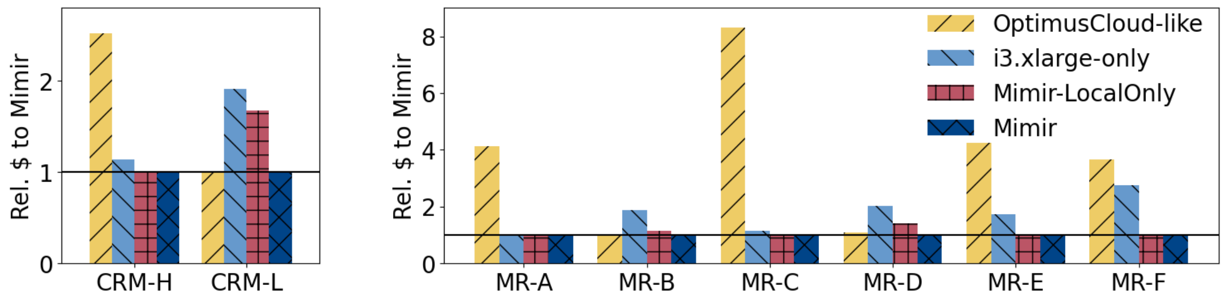


Figure 2.8: The cost-efficiency analysis of the optimization results of the workloads of the two benchmarks, `CRM` and `MR`. Throughput-intensive workloads (e.g., `CRM-H`, `MR-A,C,E,F`) prefer local SSD as its storage type. In contrast, other workloads (e.g., `CRM-L`, `MR-B,D`) that do not require high throughput prefer EBS volume to local SSD. An `i3` instance type is a costly option for some workloads (e.g., `MR-B,D,E,F`), even if it is categorized as storage optimized instance.

Figure 2.7 shows price comparison of the optimization results with Mimir and other baselines for `CRM` and `MR`, in which each benchmark is a mix of two and six distinct workloads respectively. Mimir finds cheaper VSC than the other baselines and it is up to 5.3 $\times$ cheaper than the `OptimusCloud-like`. In both benchmarks, `Mimir-LocalOnly` finds more cost-efficient VSC configurations than `OptimusCloud-like`. However, it does not mean that every workload data in the benchmarks is more cost-efficient to be stored in local SSD than EBS volume.

Figure 2.8 shows the storage preference of each workload of `CRM` and `MR`. First, `CRM-H` is the workload that requires high-performance of the storage system. Thus, Mimir finds the VSC configuration that only uses local SSD for the cost-efficient solution. On the other hand, if only EBS volumes are used to store data that requires high storage performance, it should provision much larger storage capacity than it needs to get enough volume IOPS. For example, the cost-efficient VSC configuration of `CRM-H` searched by `Mimir-EBSonly` uses 15 TiB of

Optimal Cost/Hour	W_1	W_2	$W_1 + W_2$	Gain
W_1 = MR-A, W_2 = MR-D	\$1.86	\$0.46	\$2.32	0%
W_1 = MR-B, W_2 = MR-E	\$0.33	\$1.8	\$1.91	10.3%
W_1 = MR-C, W_2 = MR-F	\$2.25	\$2.09	\$4.21	3%

Table 2.3: Pairwise workload optimization of `MR` benchmark. Mimir finds 10.3% cheaper VSC configuration when it optimizes for both workloads at once.

`gp2` volumes to store only 3 TiB of data to get enough IOPS. It costs 2.5× higher price compared to the result of `Mimir-LocalOnly` or Mimir with no resource constraint.

In contrast, `CRM-L` is the workload that does not require high-performance (i.e., capacity-intensive workload). So local SSD is an expensive storage type to store data of `CRM-L`, as it underutilizes storage bandwidth of local SSD. The throughput of `gp2` (i.e., 3 IOPS per provisioned GiB) is enough to support the workload. Figure 2.8 shows that the cost-efficient VSC configuration optimized by `Mimir-LocalOnly` costs 1.7 × higher price than the one with `Mimir-EBSonly`.

`MR` workloads with different characteristics also show different preferences on the volume type. `MR-A,C,E,F` require 4.13 ×, 8.3 ×, 4.2 ×, 3.6 × higher price with `Mimir-EBSonly` than `Mimir-LocalOnly`, respectively, while `MR-B,D` require 1.1 ×, 1.3 × higher price with `Mimir-LocalOnly`. As Table 2.2 indicates, `MR-B,D` need lower data request rate and smaller data request size than the other workloads, which makes both workloads well suited to EBS volume type.

Observation 2: *Considering diverse instance types and heterogeneous VSC configurations is crucial for cost-efficiency.*

Not only the volume type, but also the instance type is an important factor that affects the price of the virtual storage cluster. For example, `MR-F` workload requires the second highest storage system throughput per GiB of data among the workloads of `MR`, and `Mimir-LocalOnly` finds more cost-efficient VSC configuration compared to the `Mimir-EBSonly`. However, `i3.xlarge`, a storage-optimized instance type, is a costly option for the `MR-F` workload when using the `i3.xlarge-only` configuration. Instead, Mimir and `Mimir-LocalOnly` find the cost-efficient VSC configuration that uses `c5d` instance type, in which `c5d` is a compute-optimized instance type that has a small capacity of SSD. This is because the storage server for `MR-F` needs high computing power (i.e., CPU-intensive) as the workload requires high data request rate. Similarly, `MR-B,D,E` prefer `m5d` or `c5d` to `i3` instance type.

Observation 3: *Considering two workloads together in the optimization algorithm can save cost up to 10.3% compared to using two clusters optimized for each.*

Lastly, we evaluate the pairwise workload optimization of the `MR`'s workloads. We ran the *VSC Cost optimizer* for the $\binom{6}{2}$ number of pairwise combinations of the `MR`'s workloads. Table 2.3 shows the selected combinations of the workloads that minimize the total price of the virtual storage cluster when the cluster should support all the workloads. (`MR-B`, `MR-E`) pair yields the highest financial gains, 10.3% lower price, when Mimir considers them together to optimize the VSC configuration. `MR-B` and `MR-E` are cost-efficient when data is stored in EBS volume and local SSD, respectively. Thus, Mimir finds the VSC configuration that remote EBS volumes for MR-B are attached to the machines for MR-E that have local SSDs. In this way, the cost of provisioning instances for MR-B could be saved. (MR-A, MR-D) pair also have the same property (i.e., they prefer different volume types), but there is no financial gain as no computing power left in the machines of MR-A to support additional workloads in the same machine. By the pairwise workload optimization, Mimir could save total 4% additional cost compared to using six individual VSCs optimized for each workload.

Despite the large search space for the resource heterogeneity and numerous factors to consider (e.g., complex workload and storage characteristic, many price and performance SLAs), Mimir could find the cost-efficient VSC configuration.

2.4.5 Deep dive into Mimir component effectiveness

In this section, we first evaluate two components of Mimir: *Resource profiler* and *Resource predictor*. And then, we demonstrate how important finding good data partitioning is in finding cost-efficient VSC configurations.

Observation 4: *Mimir selects a cost-efficient container size to run the storage server for the given workload characteristics. Workloads tested utilize at least 83% of allocated resources.*

We evaluate how the *ContainerSpec* profiled by the *Resource profiler* fits for the given workload. Figure 2.9 shows the example of how the container with the size of profiled *ContainerSpec* works for the *workload fraction* of MR-A. Data access pattern of the *workload fraction* we tested is the same as the one of the original workload, and the performance requirements in this example are 6K QPS of data request rate and 450 GiB of data capacity. The *ContainerSpec* profiled for this *workload fraction* is 4.1 vCPU and 166 MB/s read throughput of the storage volume. To evaluate, we ran a docker container with the profiled amount of resources and measured the CPU and storage throughput of the storage server running on the docker container. As Figure 2.9 shows, the storage server utilizes 85% of both allocated computing power and storage throughput. We confirmed that the storage server running on the same container (i.e., container with 4.1 vCPU and 166 MB/s read throughput) satisfies the workload requirements, but the storage server on the next smaller container (i.e., container with 3.7 vCPU and 150 MB/s read throughput) following our container size update algorithm (§2.3.2) cannot meet the performance requirements. We also checked that even a container lack of a single resource failed to satisfy the performance requirements. We ran the same experiment on 300 *ContainerSpecs* we profiled and all the results showed the resource utilization higher than 83% of the resources allocated according to the profiled *ContainerSpecs*.

Observation 5: *Mimir can predict the cost-efficient container size using interpolation with less than 13% error.*

Next, we evaluate our *Resource predictor* using interpolation to see how accurately it predicts the right size of *ContainerSpec*. As a dataset, we use the *ContainerSpecs* that are profiled by the *Resource profiler* for the six workloads of MR.

As Mimir profiles multiple data points with different performance requirements for each workload, we used the profiled data as a test dataset to evaluate. For instance, the maximum data request rate we measured for the MR-A workload on i3.4xlarge with memory-to-data ratio

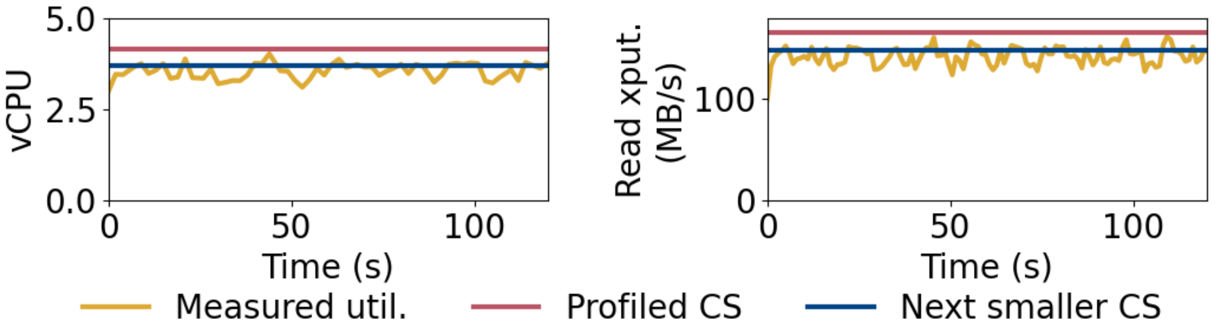


Figure 2.9: The resource utilization of `MR-A`'s *workload fraction*. The storage server utilizes 85% of the vCPU and storage read throughput (green line) of the allocated resources (red line). If any resource allocation reduces to the next smaller *ContainerSpec* (blue line), the server cannot satisfy the performance requirements.

Workload	Avg % error of the interpolation predictor			
	CPU	Read xput.	Write xput.	Net
MR-A	4.0%	1.1%	7.4%	2.0%
MR-B	2.4%	6.1%	8.4%	1.8%
MR-C	3.1%	0.8%	4.8%	1.1%
MR-D	5.9%	7.7%	1.4%	1.1%
MR-E	2.5%	0.6%	2.7%	1.0%
MR-F	12.9%	2.5%	4.5%	5.1%

Table 2.4: Percent error of the *ContainerSpec* prediction using interpolation. The interpolation predicts the cost-efficient container size with small percent errors.

of 1:16 is 20K QPS. For the measured maximum data request rate and $N = 10$ (in §2.3.2) we used, the *Resource profiler* profiled the right size of 10 different *ContainerSpecs* for the *workload fractions* of `MR-A` with the performance requirements of 2K QPS, 4K QPS, ..., 20K QPS. So we evaluate how close the profiled *ContainerSpec* for 4K QPS to the interpolation result of two *ContainerSpecs* for 2K QPS and 6K QPS. Table 2.4 shows at most 12.9% error for predicting the cost-efficient container size of `MR`'s six workloads.

Observation 6: *The cost-efficient VSC configuration varies greatly depending on how data is distributed. Only 2.4% of data partitions Mimir explored could result in a VSC configuration cheaper than 1.3 $\times$ of the minimum cost.*

Exploring data partitioning options is an important aspect of Mimir's success. Here data partitioning means how we split the data to be stored in the storage cluster, in which each split is stored in a single storage. For example, consider a workload W that defines its *workload fraction unit* (W_f) as 1/4 of the original size. Then, there are five different partitions of W, which are $\{4W_F\}$, $\{3W_F, W_F\}$, $\{2W_F, 2W_F\}$, $\{2W_F, W_F, W_F\}$, and $\{W_F, W_F, W_F, W_F\}$. Any data partitioning can be used, but we show that only tiny percentage of the possible data partitions can lead to cost-efficient VSC configuration. In this evaluation, we fixed the number of machines to use and ran the optimization algorithm of Mimir for each data partition. So we could compare

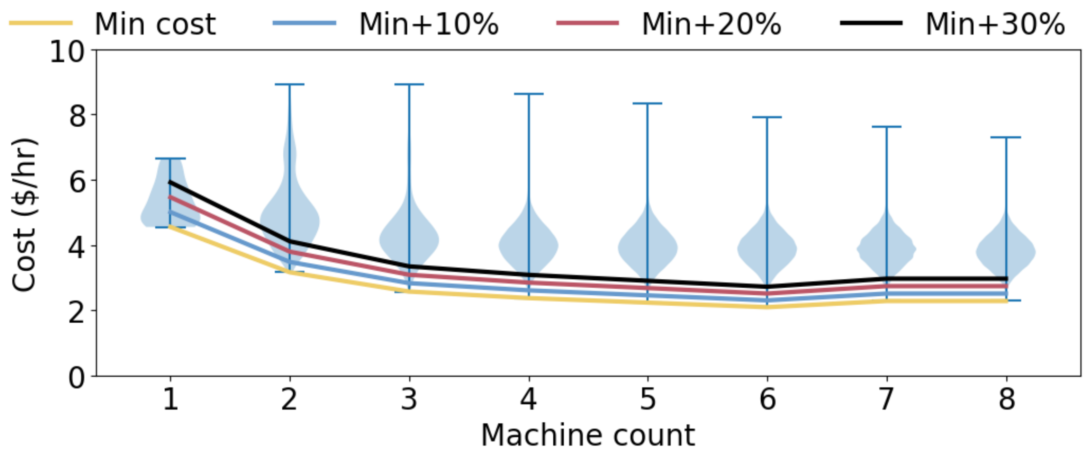


Figure 2.10: Violin plot of `MR-F` showing the distribution of the cost-efficient VSC configuration price for all possible data partitions. A tiny portion of data partitions can find the near cost-efficient VSC configuration.

the minimum price of VSC configuration of each data partition in that the algorithm finds the cheapest VSC configuration of each data partition.

Figure 2.10 is a violin plot of `MR-F` showing the distribution of the cost-efficient VSC configuration price for each data partition with a fixed number of machines. Mimir finds the most cost-efficient VSC configuration with 6 nodes at the price of 2.09$/hr. For 6 nodes, out of 6043 possible data partitions, only two data partitions could be used for the VSC configuration cheaper than 1.1 $\times$ of the minimum price, which is 2.3$/hr, i.e., if we distribute data using one of the remaining 6041 data partitions, we cannot find any VSC configuration that is cheaper than 1.1 $\times$ of the minimum price. Even for 1.2 $\times$ and 1.3 $\times$ of the minimum price, only 24 and 144 data partitions can be used for the VSC configuration cheaper than the respective prices. In other words, only 2.4% of all possible data partitions can unearth the VSC configuration cheaper than 1.3 $\times$ of the minimum cost.

As we demonstrated, although there are many data partitions and only a few of them can find near cost-efficient VSC configurations, Mimir successfully finds the most cost-efficient one using its optimization algorithm.

2.4.6 Mimir in dynamic workloads

Observation 7: *If Mimir is used by an auto-scaler at changepoints in a dynamic workload, it can reduce daily VSC cost by 74% compared to the* `OptimusCloud-like` *resource selection.*

In this evaluation, we describe how Mimir can be used by an auto-scaling system for dynamic workloads, showing that heterogeneity is still important to find cost-efficient VSC configurations. We used the `MR` benchmark for the evaluation, but unlike our other experiments, each workload has the diurnal pattern of data request rate described in the corresponding paper [1]. As described earlier, Mimir is a resource selector, not an adaptive auto-scaler, so it would fit as a component of an auto-scaling system that identifies changepoints and dynamically reconfigures as indicated by the selector. Here, we simulate the change of VSC price computed by Mimir's *VSC cost*

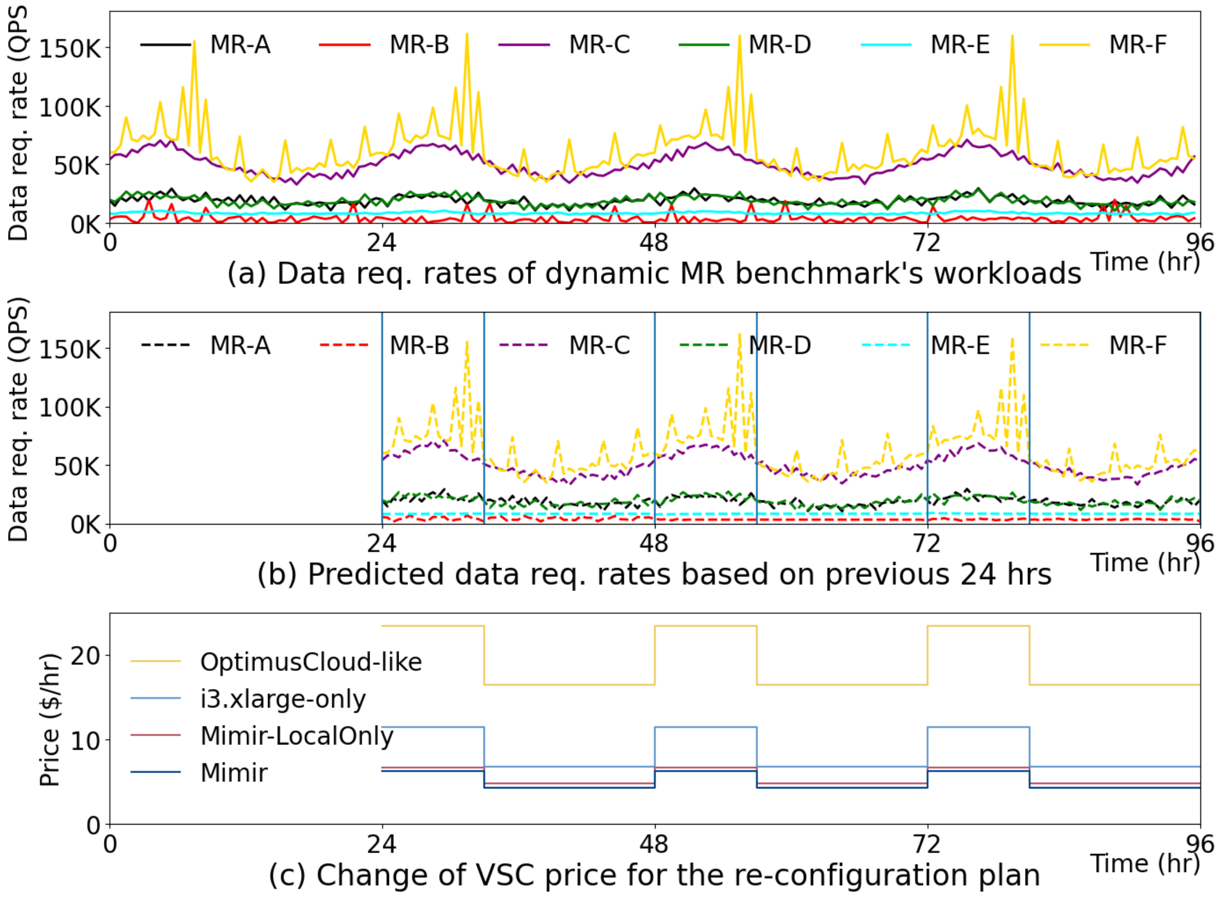


Figure 2.11: Using Mimir for dynamic MR benchmark. With dynamic re-configuration, between 8-24hr each day, one can save 32% price compared to using static VSC configuration for the peak performance requirements.

optimizer were used in such an auto-scaler.

The solid lines in Figure 2.11(a) illustrate each workload's data request rate over time generated by the dynamic MR benchmark. MR−A, MR−C, MR−D, and MR−F have strong diurnal patterns, and MR−B and MR−E are static and bursty, respectively. The dashed lines in Figure 2.11(b) are the predicted data request rates learned from the previous day's historical data – the workload behavior of *day*(i + 1) is predicted based on that of *day*(i) – using the SARIMAX [60] forecasting model. We focus on evaluating the cost-efficiency of Mimir's optimization results for the predicted workload characteristics in dynamic workloads and showing that such optimization must consider diverse storage types and configurations at any phase of dynamic workloads to minimize costs. So evaluating the accuracy of the workload prediction model is out of the scope of this study, and other prediction models [26, 61, 62, 63] can be used instead of SARIMAX.

Based on the predicted workload behavior, we can plan ahead for the dynamic change of the VSC configuration. We spot 0 and 8 o'clock every day as the change points and used the maximum data request rates between 0-8 hr and 8-24 hr as the performance requirement constraints for each period during the three days we predict the workload behavior.

Figure 2.11(c) shows the change of VSC price if VSC configurations are adaptively reconfigured using Mimir's resource selections. Using Mimir as a resource auto-selector, the optimal

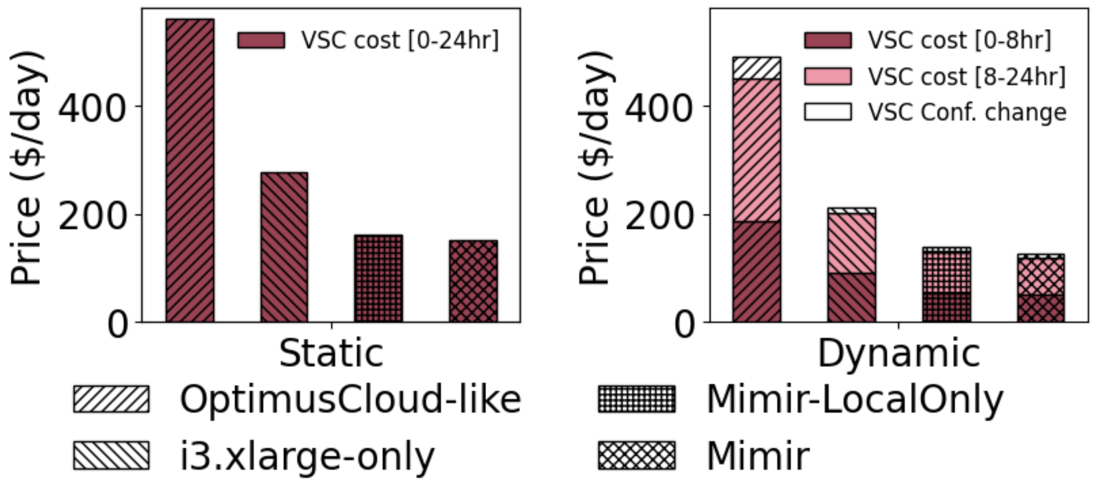


Figure 2.12: Daily VSC prices for running clusters with static and dynamic configurations. Even if the cost for changing VSC configuration is considered, Mimir is effective as a resource auto-selector for dynamic VSC reconfiguration: Mimir can save 21% daily cost.

VSC price reduces by 32% during 8-24 hr compared to the one during 0-8 hr. One reason for the cost reduction is that some workloads, such as `MR-C`, that prefer local SSDs during 0-8 hr do not require high throughput storage volumes during 8-24 hr and use EBS volumes instead, which are cheaper storage volumes. Another reason is that the computing power required by some workloads, such as `MR-F`, decreases during 8-24 hr, so fewer machines are used.

We evaluate the cost-benefit analysis by comparing the financial benefit gained by using a cheaper configuration during 8-24 hr with the cost need to be paid for the extra resources during the reconfiguration, i.e., both old and new storage clusters should be running while transferring the data. Figure 2.12 shows the daily cost comparison between running a cluster with a static configuration that satisfies peak performance requirements for a day and running a cluster that changes its configuration at 0 and 8 o'clock. Using Mimir, the data transfer takes 48 and 38 minutes for the reconfiguration at 0 and 8 o'clock, respectively, which incurs $7.7 additional costs for running the extra cluster while transferring data. Even considering such cost, there is still a 16% gain for changing the VSC configuration based on the Mimir's optimization results compared to running a static cluster.

Mimir finds more cost-efficient VSC configurations than baselines, saving 8-74% of daily VSC price compared to using the baselines to select auto-scaling system resources.

2.4.7 Profiling and optimization overheads

Observation 8: Total time and cost overheads incurred by Mimir for `MR` benchmark are 4.5 hours and $14.1, which are one-time costs for each optimization round, and time overhead can be further shortened if necessary.

We measured how much time and cost overhead Mimir incurs for a single optimization round of the `MR` benchmark. Profiling step of the *Resource profiler* and optimization step of the *VSC Cost optimizer* mainly induce the overheads.

To profile the six workloads of the `MR` benchmark, for each workload, we used one i3.4xlarge instance for a storage server and two c5.2xlarge instances for a workload generator. Each workload took 2.8 hours on average, and it costs \$14.1 by exploiting the low price of the spot instances. Also, users can further reduce the total profiling time as much as they want for no extra cost just by using more machines because Mimir can profile different *workload fractions* simultaneously as those profiling jobs are independent to each other, i.e., the profiling processes can be parallelized.

To evaluate the time overhead of our optimization algorithm of the *VSC Cost optimizer*, we used three local machines described in §2.4.1. We ran $\binom{6}{2}$ number of pairwise workload optimizations and it took 1.7 hours in total to find a VSC configuration that costs \$8.4/hr. To evaluate how fast our algorithm can find the cost-efficient VSC configuration, we compared it with an algorithm that does not use dynamic programming but use only a mixed-integer programming solver, i.e., instead of breaking the optimization problem into smaller problems as we described in §2.3.4. We used the same three machines and pairwise workload optimization in this case as well. Without dynamic programming, it failed to find a feasible solution until 30 hours; after 30 hours, the cost-efficient VSC configuration it found costs \$10.1/hr; even after 50 hours, the VSC configuration it found costs \$9.6/hr, which is still 14% more expensive than Mimir's approach.

2.5 Summary

Mimir finds cost-efficient virtual storage cluster (VSC) configurations for distributed storage backends. Given workload information and performance requirements, Mimir predicts resource requirements and explores the complex, heterogeneous set of block storage offerings to identify the lowest-cost VSC configuration that satisfies the customer's need. Experiments show that no single allocation type is best for all workloads and that a mix of allocation types is the best choice for some workloads. Compared to a state-of-the-art approach, Mimir finds VSC configurations that satisfy requirements at up to 81% lower cost for static workloads and 74% lower daily VSC price for dynamic workloads.

Chapter 3

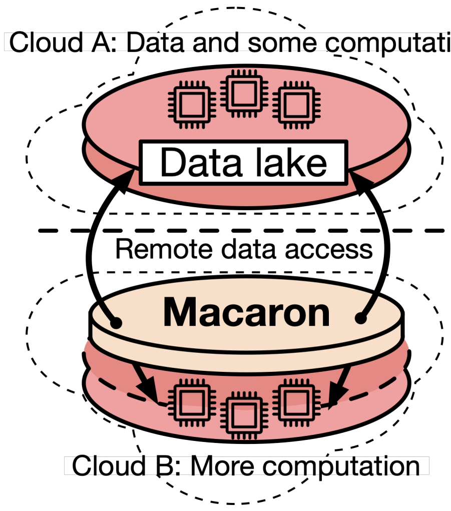
Reducing cross-cloud/region costs with the auto-configuring Macaron cache

Demand for multi-cloud is surging [64, 65, 66], driven by factors including: disparities among cloud provider features [67, 68], the desire to avoid vendor lock-in [69, 70, 71], and evolving organizational structures, such as consolidations [72]. Similarly, multi-region solutions (within a cloud provider) are becoming more popular due to data sovereignty requirements [73, 74, 75], service latency reduction [76, 77], and availability.

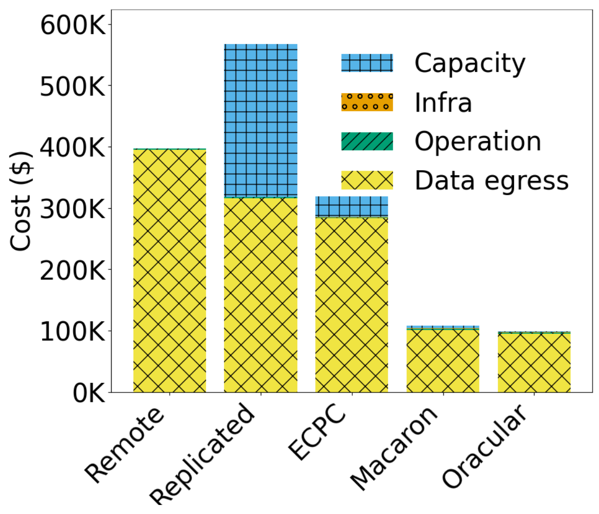
Despite many efforts to optimize resource use within a cloud or region [20, 25, 26, 37, 71, 78, 79, 80, 81], achieving cost-efficiency across clouds and regions remains a prominent challenge [82, 83, 84, 85, 86] hindering the adoption of multi-cloud/region strategies. Current cross-cloud/region data access solutions often lead to substantial increases in overall costs. Organizations typically resort to direct data access across clouds or regions [87, 88, 89], which results in prohibitively high data egress cost for frequently accessed data and high data access latency. While many organizations address the latency issue through data replication [90, 91, 92, 93, 94], the costs associated with maintaining replicas and synchronization data egress remain substantial. Public cloud providers and third parties offer caching services that can be used to keep only hot data locally [95, 96, 97, 98, 99, 100], but those rely on manual configuration of the cache size.

We analyzed object storage traces from Uber, VMware, and the IBM cloud [101], and derived three insights for cache design. First, we find that manual selection of cache capacity and storage type can lead to cost-inefficient decisions, whereas strategies that support re-configuration of the cache size and type can optimize cost and performance. Second, the high data egress costs of workloads skewed toward lower accesses per object highlight the need for large cache capacities that are feasible only when cheap cloud storage types are used. Third, diverse and dynamic access patterns within and across traces make it essential to monitor workloads and automatically reconfigure the cache when needed.

In response to these findings, we introduce **Macaron**, an auto-configuring cache system that monitors the workload, dynamically adjusts cache capacity, and utilizes different cloud storage types to minimize data access costs and latency from remote data lakes (Fig. 3.1a). A key insight behind Macaron is that cache sizes in the cloud are constrained by cost rather than hardware constraints, causing us to rethink cache design and eviction policy. Macaron analyzes the need



(a) Macaron in multi-cloud



(b) Costs for each approach

Figure 3.1: Multi-cloud/region workloads are challenging to run cost-efficiently. Early industry adopters rely on accessing all data remotely or replicating it locally, with some using an existing in-memory caching cloud service (`ECPC`). Macaron significantly reduces costs compared to these methods and achieves costs comparable to offline optimal (`Oracular`). The experiment details are provided in §3.6.2.

to add new objects to the cache and adjust cache capacity accordingly, while exploiting cheap storage types for caching, rather than solely relying on a cache replacement policy [102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112] or efficient sharding of limited storage hardware among applications [113, 114, 115, 116, 117, 118, 119, 120, 121].

Macaron leverages two storage types for caching. Object storage capacity cost is considerably lower than the egress cost incurred by cache misses, so Macaron uses it to minimize the overall cost of accessing remote data. A distributed DRAM component is elastically adjusted to ensure acceptable latency. The capacity of each tier is periodically adjusted based on data access patterns, using a version of the miniature simulation technique [122], modified to obtain byte miss curves and average latency curves. We also explored a variant of Macaron that adopts a cache policy using time-to-live (TTL) for eviction, adjusting TTL instead of capacity. Our results show that both achieve similar cost savings. A serverless implementation allows Macaron to achieve performance suitable for online miniature simulation of very large caches.

Fig. 3.1b shows that Macaron significantly improves upon the cost of existing approaches. Each bar represents the total cost of running the 19 cloud-based object storage workloads analyzed in this project (see §3.2.2) across clouds. Macaron achieves 73% cost reduction compared to accessing all data remotely by avoiding egress costs, and a 81% cost reduction compared to replicating all data locally by reducing both capacity and synchronization-driven egress costs. Macaron also achieves 66% cost reduction compared to elastic cloud provider caching (ECPC) services, which incur expensive DRAM storage costs even when tuned intelligently. An oracular approach with perfect knowledge of future accesses only improves cost savings by an additional 9%, compared to Macaron, without latency reduction.

We have evaluated Macaron with traces from IBM, Uber, and VMware, and our results show

Operation	AWS	Azure	GCP
Egress to Internet (per GB)	9¢	8.7¢	11¢
Egress btw. regions (per GB)	2¢	2¢	2¢
Object storage (per GB-mo.)	2.3¢	2.1¢	2.3¢
DRAM (per GB-mo.)	700-1200¢		
Object GET (per 1k requests)	0.04¢	0.05¢	0.04¢
Object PUT (per 1k requests)	0.5¢	0.65¢	0.5¢

Table 3.1: Cloud storage pricing[1] of three public cloud providers is similar, with egress cost dwarfing other costs.

the importance of adapting the cache size and configuration to workload changes and that substantial cost savings can be achieved by combining object storage and an elastic DRAM cache cluster.

Contributions. (1) We collected cloud storage workload traces from Uber and VMware, and publicly released Uber trace [123]. (2) We analyze real-world cloud storage workloads and derive design objectives for effectively caching these workloads. (3) We describe the Macaron cache that is adaptively auto-configured to minimize costs without compromising latency by leveraging object storage as a cache storage type. (4) We experimentally demonstrate Macaron's ability to achieve 65% reduction on average in remote data access costs compared to existing solutions. (5) Lastly, we release the Macaron prototype [124] and simulator [125] code.

3.1 Motivation and challenges

Cross-region data access within a cloud provider is gaining prevalence for several reasons. First, computation and data could end up separated due to resource unavailability within a region [126, 127], e.g., due to high demand for GPUs. Second, new services are usually not available in all regions simultaneously, requiring applications to span regions in order to adopt new technologies [128, 129]. Finally, data sharing via cross-region data access is essential for international business teams [130] and applications can be distributed across multiple regions to provide lower latency to end-users [131].

Recent research [132, 133] has shown that distributing a data pipeline across multiple clouds saves costs due to price differences between providers, as demand for multi-cloud solutions is rising. A recent survey [134] found that 55% of multi-cloud users already deploy a single workload across multiple clouds, which often requires cross-cloud data transfers. In conversations with a major trading company we found that during workload bursts, they utilize multiple cloud providers to ensure trading performance scales with demand even when resource availability becomes an issue for one cloud provider [135]. Other companies have to split data to abide by data residency rules preventing data movement [136].

While multi-region/cloud strategies have clear advantages, we identify two important challenges: high data egress cost, and increased data access latency.

[1]Prices from N. Virginia region, <10 TB egress to the Internet, inter-region transfers within N. America, and <50 TB storage capacity.

Trace	Op. %		Skew	Total	Data accessed		Remarks
	Put	Get	(Zipf)	data size	Put	Get	
IBM 9	N/A	100	0.22	6 TB	0	34 TB	Short lifetime: last - first access $<$ 10min
IBM 12	1	99	0.97	5 TB	4 TB	603 TB	High data access repetitiveness
IBM 18	2	98	0.64	4 TB	231 GB	14 TB	High request rate, small object sizes
IBM 55	55	45	0.42	13 TB	12 TB	10 TB	Strong diurnal access pattern
IBM 83	40	60	0.72	64 TB	37 TB	94 TB	Low compulsory miss ratio (=0.12)
IBM 96	58	42	0.20	78 TB	46 TB	36 TB	High compulsory miss ratio (=0.87)
Uber	N/A	100	0.52	324 TB	0	941 TB	Stable data access pattern
VMware	N/A	100	0.47	215 GB	0	71 TB	Small total data size, high request rate for testing

Table 3.2: We collected and analyzed new traces from Uber and VMware to understand how to efficiently cache cloud storage workloads. The IBM traces represent diverse access patterns among the busiest cloud object storage traces from IBM's repository.

Challenge 1: Prohibitive data egress cost. While transferring data into public clouds is often free, moving data out incurs substantial charges based on the volume of data being transferred (Table 3.1). For instance, one of the IBM traces we have analyzed accesses 694TB in a week, resulting in cross-cloud data transfer costs of \$64K/week or \$3.3M/year. The same workload would incur \$14K/week or \$0.73M/year if data was transferred across regions of the same cloud.

Despite being controlled by public cloud providers, data egress costs have remained stable over time – GCP, AWS, and Azure have maintained their egress costs unchanged for the past 6 to 10 years. Moreover, egress costs have been consistently identified as a barrier to cross-region/cloud adoption in many surveys [134, 137]. *Reducing the data egress cost is crucial for embracing the multi-region/cloud era.*

Challenge 2: High access latency. In latency-sensitive workloads, like real-time analytics or streaming services, encountering consistently high latency is unacceptable [138, 139]. Even in less latency-sensitive workloads, high data access latency can increase costs by increasing runtimes and causing workloads to use compute resources for longer periods [82].

Cross-region data access causes higher latency compared to single-region access. We measured object retrieval times at a public cloud and found that retrieving a 1KB object from local object storage in one U.S. region showed significantly lower latency, taking 10s of milliseconds, than fetching the same object from another U.S. region or Europe, which took 100s of milliseconds. Real-world workloads we evaluated experienced $2-5\times$ higher average latency with cross-region data access. *High data access latency to remote data should be mitigated for both performance and cost.*

3.2 Macaron design drivers

In this section, we analyze existing approaches for cross-cloud/region data access, and real-world cloud object storage workloads from three large companies. We derive three design objectives that have inspired the design of Macaron.

3.2.1 Limitations of current approaches

The simplest approach to accessing data across clouds or regions is remote access, requiring no additional synchronization efforts. However, in scenarios with repetitive data access patterns, as observed in various storage workloads [1, 112, 140, 141], egress costs are repeatedly incurred for the same objects, alongside latency issues.

One effective approach to reduce latency is to replicate all data and access them from local object storage, which eliminates recurring data egress costs as well. However, it does not solve the cost problem entirely, as the transfer cost to synchronize *dark data* [142, 143, 144] (i.e., data that is written once but never accessed) inflates the egress cost. Recent surveys [145, 146, 147, 148] have indicated that the percentage of dark data can range from 40% to as high as 95% across different organizations. Even worse, maintaining a large capacity of replicated data lake is also expensive.

We view the above two patterns as endpoints of a spectrum, with caching solutions providing a middle ground. While using existing cloud caching services appears straightforward, no service currently offers cost optimization solutions for addressing high egress costs associated with remote data access. Users must manually configure cache settings, such as cache capacity and storage type, a task that can be challenging even for experienced cache experts. Moreover, most existing services prioritize using DRAM or block storage for caching, primarily focusing on single-region data access performance, but our evaluation confirms that such approaches remain costly due to expensive capacity expenses.

Design objective 1: *We need caching strategies that bridge the gap between all-remote data access and full data replication. These strategies should support auto-configuration of the cache to optimize both cost and performance.*

3.2.2 Cloud object storage workload characteristics

To better understand how cloud object storage workloads should be cached, we analyzed IBM cloud object storage traces [101], along with traces we collected from Uber and VMware. These traces are collected from systems operating within a single region. The Uber and VMware traces represent workload accesses that are not expected to change substantially when moved to an architecture that spans regions or clouds. Additionally, we evaluated diverse workload patterns using IBM traces to broaden our evaluation, extrapolating these workloads to the cross-region and cross-cloud scenarios described in §3.1.

IBM traces. These are anonymized object access logs from IBM cloud's object storage over a 7-day period. We identified 15 traces [2] with the most traffic, together making up 95% of all

[2]IBM traces with IDs 4, 9, 11, 12, 18, 27, 34, 45, 55, 58, 66, 75, 80, 83, 96.

data accessed across all 98 IBM traces. While we have studied all 15 traces, for brevity we present detailed results for 6 traces that are representative of all unique workload characteristics that appear in the trace collection (Table 3.2).

Uber trace. We collected object access logs generated from Uber's Presto production deployment, primarily used for processing and analyzing large-scale real-time event data streams, and querying data streamed through Apache Kafka to provide real-time data insights [149]. Our logs span three Presto engines and over 18 days. Over 70% of the accesses are generated by periodic jobs.

VMware trace. We collected AWS S3 requests generated by AWS Athena queries from VMware's test infrastructure, spanning an 8-day period. These analytics queries, comprising a mix of ad-hoc and scheduled jobs, analyze security data and resemble queries in the production system but are smaller in scale as part of testing before deployment in production [150]. This workload exhibits a high data request rate despite its relatively small dataset size.

Our analysis of these workloads helped us derive two design objectives for caching cloud object storage workloads.

Large objects and higher spread of accesses. Data accesses often follow the Zipf distribution [1, 141, 151, 152, 153], and we have confirmed that to be a good fit for our cloud object storage traces as well. Zipf's exponent α [154] represents the skewness in data access frequency per object. Higher α values indicate fewer objects receiving most accesses, so smaller cache capacities suffice for these workloads.

We find that cloud object storage workloads generally have lower α than those of KV-store or block I/O traces. Over 78% of IBM workloads and both Uber and VMware workloads have $\alpha < 0.6$, while more than half of Twitter KV-store traces [141] have $\alpha > 1.1$. Thus, while previously studied workloads achieve acceptable cache miss ratios with small cache sizes relative to the overall dataset, cloud object storage workloads need to cache a significant portion of data to mitigate bytes missed, which directly affects egress costs.

Many objects in cloud object storage workloads are large. For example, the IBM traces' median object is 10-100KB, while for Twitter it is 20-30B, orders of magnitude smaller. Given that data access frequency being skewed towards few accesses per object, this suggests that reducing egress costs through caching requires a large cache capacity.

Design objective 2: *Cloud object storage workloads are skewed towards low accesses per object, so to reduce high data egress cost we need large cache capacities, which are feasible by leveraging cheap storage types.*

Diverse and dynamic data access patterns. Cloud object storage serves as backend storage for a variety of workloads including machine learning [155, 156, 157, 158], ETL [159, 160], SQL queries [161, 162, 163], and file serving [164, 165, 166], resulting in diverse data access patterns. Understanding the unique characteristics of each trace in Table 3.2 is vital for establishing a cost-efficient cache configuration. For instance, while IBM 96 is larger than IBM 83, the difference in data access skewness necessitates larger cache capacities for IBM 83, which are cost-efficient only using cheaper cloud storage. IBM 9, despite having low data access skewness, does not benefit from large cache capacity due to its short-lived objects, necessitating a different approach for cost-efficiency.

Despite thorough observation and understanding of workload characteristics, a cost-efficient cache configuration varies over time. For traces with more dynamicity, like IBM 80, dynamically adjust cache size results in 85% cost reduction compared to a statically configured cache. But even for traces with stable, periodic data access patterns, like Uber, a dynamically configured cache can result in a 15% cost reduction.

Design objective 3: *Cache re-configuration based on workload monitoring is necessary to accommodate diverse and evolving data access patterns.*

3.3 Macaron design

Macaron is an auto-configured cache system designed to minimize the cost of remotely accessing data stored across clouds or regions, while ensuring acceptable latency. Macaron intelligently configures the cache storage type and adaptively adjusts cache size by periodically analyzing data access patterns. We elaborate on the key characteristics guiding Macaron's design (§3.3.1) addressing the design objectives from Section 3.2, then describe Macaron's architecture (§3.3.2).

3.3.1 Macaron design characteristics

Adopting object storage for cache storage type. While object storage is typically used for data lake storage [158, 167, 168], Macaron uses it as a second-level cache storage type. Cloud-based object storage workloads, as detailed in §3.2.2, are often cache-unfriendly [169, 170, 171], necessitating a large cache capacity to mitigate costly egress expenses. By leveraging the remarkably low object storage capacity cost (300× cheaper than DRAM), Macaron can provision extensive cache capacity cost-effectively, and still reduce overall costs through lower egress transfer costs.

With DRAM-based caches, however, neither opting for a large cache size nor settling for a smaller one presents a satisfactory solution to minimize costs; the former incurs prohibitively high capacity costs, while the latter results in a significant total miss penalty (i.e., egress costs). Therefore, as a first-level cache, Macaron uses the smallest DRAM cache cluster size that meets performance requirements.

While flash caching is often used as an inexpensive storage type [172, 173, 174], we leave exploring other options for future work, as object storage remains significantly cheaper and its inherent elasticity aligns better with adaptive cache reconfiguration. Otherwise, Macaron would need to manage a virtual storage cluster for flash caching. By default, Macaron uses standard object storage types like S3 Standard or Azure Blob Storage Hot Tier for caching, with support for other types by adjusting the cost policy.

Adaptive cache reconfiguration. Macaron periodically analyzes data access patterns and adjusts capacity of each cache level, assuming patterns will repeat in the future, similar to prior work [113, 175, 176]. Macaron determines a cost-efficient object storage cache capacity that minimizes overall remote data access costs and a DRAM cluster size that meets acceptable average latency. Our evaluation confirms that frequent optimization (every 15 minutes) leads to more cost-efficient solutions (§3.6.3), enabled by cloud resource elasticity and Macaron's rapid workload analysis. To achieve the latter, we extend miniature simulation techniques [122] and implement them in serverless functions. Achieving highly accurate workload prediction [177, 178]

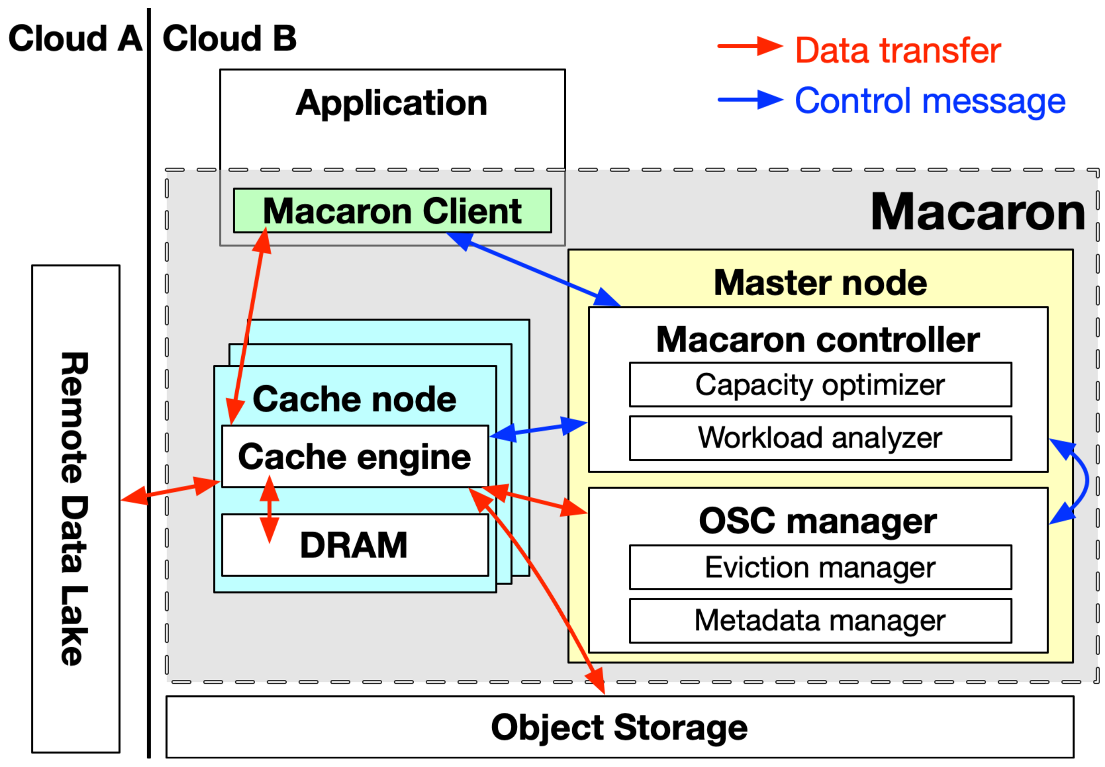


Figure 3.2: Macaron Overview: an Object Storage Cache (OSC) manages data egress costs, Cache nodes leverage DRAM to improve latency, and the Macaron controller is responsible for cache auto-configuration.

falls beyond the scope of our work, yet our evaluation demonstrates Macaron's robustness in handling real-world workloads, even when unobserved workload patterns emerge (§3.6.2).

3.3.2 Macaron architecture

Macaron consists of four components: a Macaron client, a cache engine, an object storage cache manager, and the Macaron controller (Fig. 3.2). Macaron uses two-level caching. As the first-level cache, a cache engine and DRAM cache scale together across cache nodes that make up a cache cluster. An object storage cache (OSC), the second-level cache of Macaron, is controlled by the OSC manager. Macaron uses inclusive caching, where data in the cache cluster is also stored in the OSC.

The **Macaron client** is the primary interface for applications, facilitating a connection to Macaron and the transmission of data requests, such as put, get, and delete operations to a remote data lake. It employs consistent hashing for message routing to the cache cluster that is auto-scaled. The Macaron client maintains up-to-date cache cluster information by communicating with the Macaron controller to determine which node to access.

Upon receiving requests from the Macaron client, the **Cache Engine cluster** interacts with both cache layers and the remote data lake. Macaron uses a write-through and inclusive policy by default, so the cache engines are responsible for cache promotion. When Macaron targets solely minimizing the cost, the cache cluster is not deployed, and the cache engine is co-located

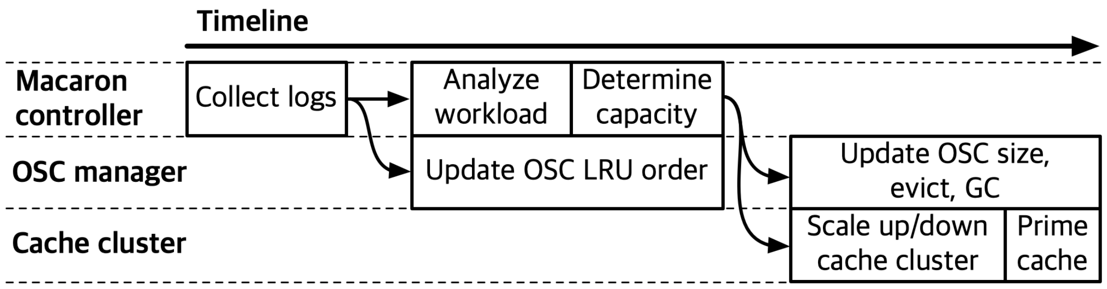


Figure 3.3: End-to-end pipelining of cache capacity optimization and reconfiguration. Macaron executes processes in parallel wherever feasible for fast reconfiguration.

in the same node as the Macaron client.

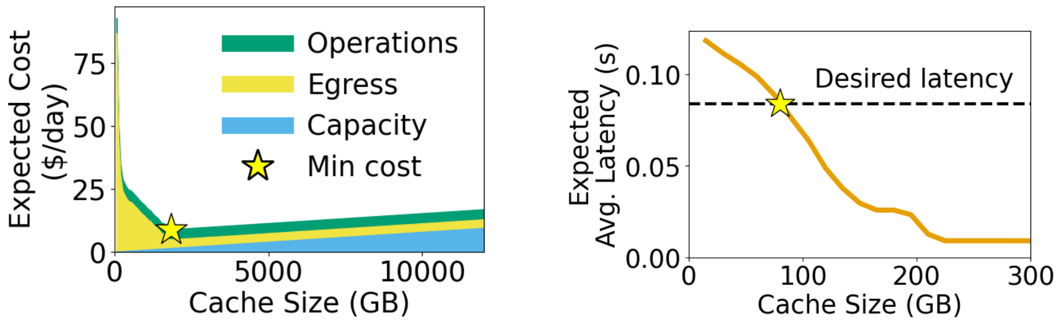
While the DRAM cache is self-managed, Macaron deploys an **OSC manager** to manage metadata and cache eviction from the OSC. To reduce operational costs of OSC, Macaron uses object packing that combines small objects into packing blocks when writing cache items into the OSC. OSC manager's metadata manager provides the mapping of cache objects to the corresponding packing block. Macaron lazily evicts cache items from the OSC using the Eviction Manager. We use the LRU eviction policy for both the OSC and DRAM cache, but alternative policies can be easily incorporated.

Finally, the **Macaron controller** is responsible for adaptive cache management. Its optimizer determines the sizes of both the OSC and the cache cluster to minimize cost while striving to enhance performance based on past data access patterns. Then it scales both cache layers accordingly. The Macaron controller gains insights into data access patterns through the Workload Analyzer that periodically collects and analyzes data access logs.

Supported operations. The Put operation synchronously writes data to the packing block being constructed, the DRAM cache (if present), and the remote data lake, before returning to the client. The packing block is asynchronously flushed to the OSC in the background. This ensures data durability for the remote data lake. The Get operation attempts to retrieve data from the DRAM cache, the OSC, and the remote data lake in sequence, returning it immediately upon success. The Delete operation removes data from Macaron's DRAM cache, OSC, and the remote data lake before returning.

3.3.3 Consistency model

Macaron is designed to guarantee the same consistency model as using the remote data lake alone. To do so, Macaron assumes data is immutable, a paradigm prevalent in data lakes such as Meta and Alibaba data warehouses [179, 180], AWS S3 lakeFS [181], and in cloud database formats like Apache Parquet and Apache Iceberg. With its write-through policy, Macaron ensures consistency equivalent to using the remote data lake alone. Applications requiring mutable data must handle it, potentially by using TTL for each item or functions like S3 Object Lambda to detect and invalidate stale data.



(a) Expected cost curve (b) Expected avg. latency curve

Figure 3.4: Example curves (Trace 55) utilized to optimize OSC and the cache cluster capacities.

3.4 Cache auto-configuration pipeline

We introduce the workflow used by Macaron to adaptively optimize the capacity of each cache level by analyzing data access patterns for both cost and performance.

Workflow. Macaron triggers the reconfiguration process at fixed intervals, set to 15 minutes by default. When reconfiguration is triggered (Fig. 3.3): the Macaron controller first collects data access logs from cache engines, then the Workload Analyzer (§3.4.2) generates key metrics on the recent data access pattern (e.g., miss ratio curve, byte miss curve), and the OSC manager updates the LRU cache item list. Based on these metrics, the Macaron controller determines the most cost-efficient capacity and reconfigures the OSC and cache cluster accordingly (§3.4.1).

3.4.1 Capacity optimizer

OSC capacity. Macaron determines the OSC capacity that minimizes the overall cost of accessing remote data across clouds or regions. The capacity optimizer generates an *expected cost curve* based on past data access patterns, predicting the expected cost for different OSC capacities during the next time window and selects the size that minimizes the expected cost (Fig. 3.4a). The expected cost for a cache capacity (C) is computed as:

$$\begin{aligned} TotalCost(C) &= OSCCapacityCost(C + GarbageSize) \\ &\quad + EgressCost(C) + OpCost(C) \\ EgressCost(C) &= EgressPrice \times ByteMissCurve(C) \\ OpCost(C) &= PutPrice \times \\ &\quad \left(\frac{\#Writes + \#Reads \times MissRatioCurve(C)}{\#Objects\ per\ Packing\ Block} \right) \end{aligned}$$

In summary, the expected total cost consists of the object storage capacity cost, egress cost, and operation cost of object storage. Capacity cost is based on OSC size and garbage size, a side effect of object packing, tracked by the OSC manager, which monitors total data stored in OSC and its determined capacity. Egress cost is proportional to the bytes missed from the OSC. Operational costs for storing cache items to OSC are proportional to the number of Put operations and cache admissions, divided by the number of packed objects, since admitted objects are written to the OSC as a block. The costs unaffected by changing the OSC capacity are omitted in this formula, including the VM cost for Macaron controller, data transfer costs incurred by write operations (due to the write-through policy). As capacity increases, capacity cost increases while operation cost and egress cost decrease, attributed to the reduction in miss ratio and byte miss in OSC.

Cache cluster capacity. Macaron aims to configure the minimal cache cluster capacity needed to achieve better average latency than the replication approach. Macaron utilizes the average latency curve (Fig. 3.4b) to select the minimum cache cluster capacity that meets the desired latency threshold. However, in traces with high cold miss ratios, achieving this objective may not always be feasible. In such cases, the Macaron controller uses a maximum curvature method [182] to identify the knee-point. It connects the latency-cache size curve's two endpoints and locates the farthest point between the curve and this line, beyond which further expansion of the cluster size yields no latency improvement.

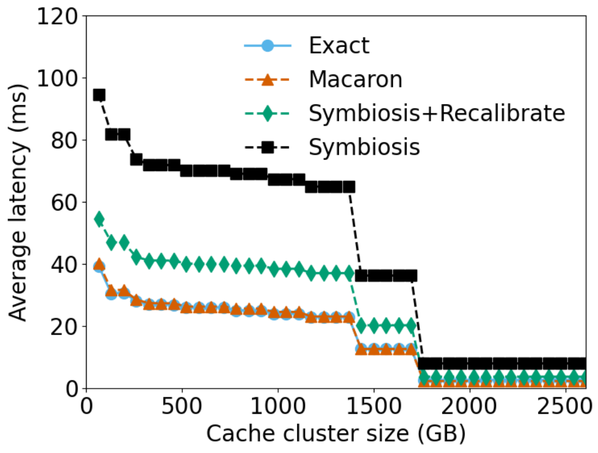
TTL cache for OSC. Given that there is no capacity limit in object storage, implementing a TTL cache on the object storage is another viable option. To assess whether Macaron's techniques can be applied to adaptive TTL-based object storage caching, we implemented Macaron-TTL, a variant of Macaron that uses a TTL cache and automatically determines a TTL that minimizes the total cost of accessing remote data. This variant employs the same Workload Analyzer to generate necessary metrics for computing the expected cost for TTL instead of cache size, considering a similar trade-offs: a TTL that is too short increases cache misses and egress costs, while one that is too long raises storage expenses. Our evaluation, as shown in §3.6.8, confirms that cost savings achieved by Macaron-TTL are nearly identical to those from optimizing OSC capacity.

3.4.2 Workload analyzer

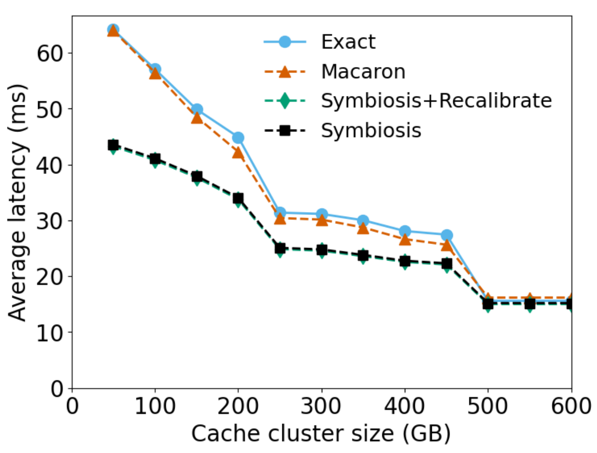
Macaron uses a short optimization window, set to 15 minutes, to leverage the cost benefits of frequent reconfigurations, which requires fast yet accurate workload analysis. The Workload Analyzer adopts and extends the miniature simulation technique [122] to derive three key metrics representing the recent access pattern: miss ratio curve (MRC), byte miss curve (BMC), and average latency curve (ALC). Then, using accumulated metrics from historical access patterns, Macaron generates aggregated metrics for capacity optimization. For Macaron-TTL, the same curves are used but with TTL on the X-axis instead of cache size.

Miniature Simulation. Waldspurger et al. [122] introduced a technique for generating MRCs that emulates caches of any specified size by proportionally scaling down the actual cache size and using spatial sampling to sample data accesses. Among MRC generation studies [152, 183, 184, 185, 186, 187], we chose this method for its efficiency in generating MRCs online and its adaptability in computing additional metrics utilized by Macaron, such as missed bytes or average latency.

We follow the original miniature simulation approach to obtain the MRC, and monitor cache miss bytes from the mini-caches, dividing them by the sampling ratio to approximate miss bytes of the original cache sizes, thereby generating the BMC. This process deviates only slightly from a full simulation, with a mean absolute error of 0.0023 for the MRC and a mean average



(a) ALC of IBM 55



(b) ALC of IBM 12

Figure 3.5: Macaron's ALC achieves high accuracy by computing the latency at runtime and incorporating proper request delaying, closely matching the exact average latency.

percentage error of 0.015 for the BMC, evaluating all 19 traces.

Symbiosis [113] uses miniature simulation to generate a MRC and produces an ALC based on the MRC to auto-tune application and kernel cache sizes. It uses access latency measured at the beginning and hit ratios measured at runtime of each cache layer to calculate average latency. However, Macaron considers two more factors, workload change and false positive hits, improving accuracy further.

First, Symbiosis assumes cache and disk latency do not change, but we observe that the access latency distribution to object storage varies over time since it depends on the object size distribution, which varies over time. Second, when uncached data are consecutively accessed within a very short time (before a remote access completes), Macaron's cache engine delays subsequent accesses until the first request completes to reduce redundant egress costs, causing them to experience remote access latency. However, in simulation, subsequent accesses after the first one are classified as hits, underestimating latency by using the cache cluster latency and generating an ALC with lower latency values.

To resolve these issues, Macaron directly computes average latency for each access during miniature simulation and aggregates them afterward, and we added the request delay in the simulation used by Macaron. Moreover, we ran two-level mini-caches that depict the cache cluster and OSC, using the same sampling and scaling logic. The cache cluster capacity serves as the independent variable for ALC, while OSC's cache capacity works as input to ALC optimization, since it is decided by the Macaron controller.

Fig. 3.5a illustrates the accuracy of Macaron factoring in workload changes compared to Symbiosis that uses the unchanged measured latencies for 7 days. In this case, the workload changes from accessing large objects to small objects, thus Symbiosis yields inaccurately higher latency (black). Symbiosis behaves much better once we force it to recalibrate every 15 minutes, but still worse than Macaron. Fig. 3.5b demonstrates the effect of false positive hits, where Symbiosis reports inaccurately lower average latency.

Metric Aggregation. After analyzing the local data access pattern, the Workload Analyzer stores the results for future reference. It aggregates those saved local metrics according to the optimization goal.

To optimize costs, retaining historical data access patterns is crucial due to the high data egress cost, which aligns with the cost incurred for storing the same object in object storage over extended periods, specifically 116 days for cross-cloud and 26 days for cross-region accesses. Therefore, conservatively including old data access patterns is a key for cost-efficiency. To achieve this goal and adapt to workload changes, we use an exponential decay mechanism by multiplying metrics by a decay factor $\gamma^{elapsed\_time}$, which diminishes the influence of older metrics. This is simple yet yields effective results (§3.6.3). Additionally, we also multiply weights proportional to the number of requests per reconfiguration window. This prevents metrics derived from a small number of requests from misrepresenting the overall data access pattern.

To optimize performance, however, only the latest access pattern is important as Macaron needs to quickly scale-in the cache cluster when the large cache cluster capacity is ineffective. Thus, Macaron uses the latest ALC to configure the cache cluster capacity.

3.4.3 Policy during observation period

Macaron starts to trigger optimization after the cache is warmed up and stable data access patterns are observed, using the first day as the observation period. During the observation period, Macaron can either cache all accessed data or none at all. We find that storing all accessed data led to a significant reduction in the cost of remote data access, averaging at 37% compared to not storing any data. The main reason for this is twofold: (1) object storage cache is cheap, so storing all data for 24 hours does not incur significant overhead, and (2) on the first day, if no data is cached, the egress cost for repetitively accessed data is very high. Comparing to the use of optimal cache capacity during the observation period did not yield significant cost savings either.

3.4.4 Offline optimal algorithm

To assess Macaron's cost-efficiency, we compare it with the optimal solution, `Oracular`, which has complete knowledge of trace requests. While the Belady algorithm [188] is known to be the optimal eviction algorithm, OSC differs in two main ways: (1) its elastic nature, eliminating the need for forced evictions, and (2) its focus on overall cost rather than just miss ratio. Given these differences, `Oracular` determines for each cache item access whether the cost to store data in the OSC until the next access is less than the data transfer cost. If higher, the item might be evicted or not stored.

For our comparison, we assume zero operation cost for `Oracular`, suggesting optimal packing and minimal operation costs. Also, we take the end of the trace length as the end of the workload for `Oracular`, but real-world workloads continue beyond trace lengths. Hence, `Oracular` stands as an idealized benchmark that Macaron aims to approach, even if it may not be reachable.

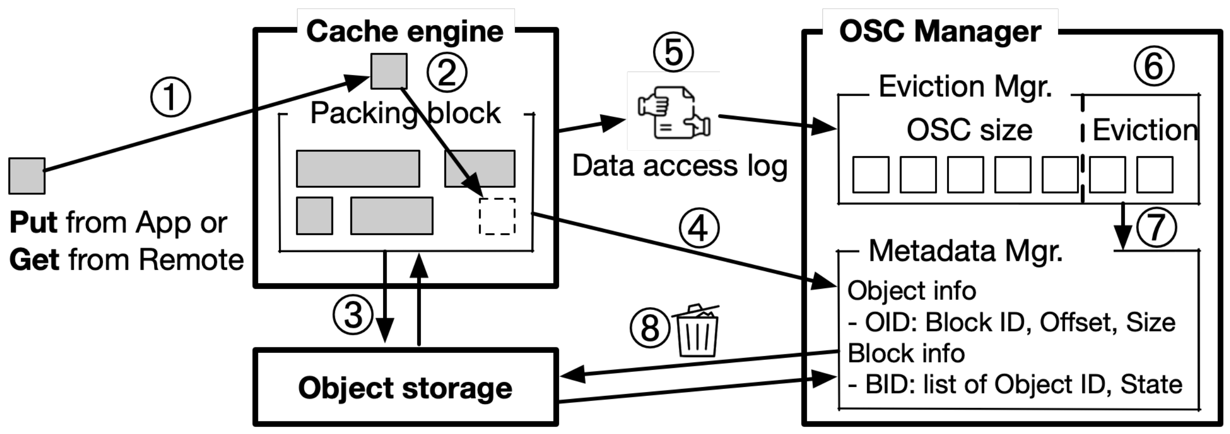


Figure 3.6: Overview of how OSC manager manages object packing, lazy eviction, and garbage collection.

3.5 Implementation

We implemented the Macaron prototype in 8k LoC of C++, and we explain the important implementation details that influence performance and cost.

3.5.1 Object storage cache implementation

Object packing. Object storage write operations are 12.5-13× more expensive than reads. However, Macaron has to perform frequent writes for cache admissions, driven by the low skewness of object storage traces. Macaron mitigates this issue through object packing [80], bundling small objects into larger blocks before writing them to object storage. Macaron's Cache Engine (Fig. 3.6 ①-④) combines objects that need to be written in OSC into blocks, then full blocks are written to OSC, and OSC manager metadata is updated to map blocks to objects. The Cache Engine uses byte-range fetches to retrieve objects from blocks.

By default, Macaron sets a packing threshold of up to 40 objects and a block size of 16MB. The workloads we studied break data in up to 4MB objects for caching, for which object packing can achieve 4× operation cost reduction, while smaller objects see reductions up to 40×. Larger block sizes reduce request costs but increase memory consumption on the cache nodes, which store data at block granularity.

Lazy eviction and garbage collection. Traditional caches evict items when reaching physical capacity. Macaron exploits object storage elasticity, delaying evictions and batch processing them to reduce operational costs. When eviction is triggered (Fig. 3.6 ⑤-⑧), the OSC manager leverages access logs to update its state of OSC objects to `Evicted`. Lazy evictions are followed by garbage collection for blocks with over 50% `Evicted` or `Deleted` objects, where a block is read and a new block is written out to OSC containing only active objects. Note that Macaron does not traverse all blocks for garbage collection. Instead, when a Delete or Evict occurs, it computes the percentage of valid items in each affected block. If the percentage falls below a threshold, the block ID is added to the *GCList* for tracking. Only the blocks in this list are targeted during garbage collection.

Lazy evictions resolve performance issues related to updating cache replacement policy metadata [101, 189, 190], by removing it from the critical path of requests.

3.5.2 Cache cluster implementation

DRAM cache priming. The speed at which Macaron can warm up new cache capacity following an auto-scaling event is crucial for efficiently utilizing new cache nodes.

We observe that many object storage workloads, especially IBM traces, have lower object request rates than the other key-value or block I/O workloads. For example, while average data request rates of Twitter [141] and Enterprise VDI storage traces [191] are 7k and 33k RPS, IBM traces do not exceed 344 RPS. This disparity is likely due to the high latency and monetary costs associated with retrieving data from cloud object storage, prompting users to maximize the use of fetched data. Hence, new cache capacity will get populated slowly, reducing our ability to mitigate latency spikes through existing methods like a Gradual algorithm [192].

To address this, Macaron incorporates cache priming for newly launched cache nodes. During this process, the OSC manager scans the LRU order of cache items and preloads data into new cache nodes until they are full.

3.5.3 Macaron controller implementation

Miniature simulation. Using spatial sampling at a ratio of 5%, we ran at most 200 mini-caches (ghost caches), with the largest covering the total data size of each workload[3]. Macaron deploys each mini-cache as a serverless function, running 200 simulations in parallel for rapid analysis (§3.6.7). To avoid replaying all accumulated traces in each optimization, Macaron stores the states of each mini-cache in Amazon EFS after simulation, loads the states back for subsequent simulation, and updates them during simulation execution. Metrics like miss ratio, bytes missed, and average latency, generated by each mini-cache during each optimization window, are also stored in EFS for use by the Macaron controller's capacity decision. As detailed in §3.4.2, Macaron runs two types of miniature simulation: one to generate MRC and BMC in a single run, and another to produce the ALC.

The pay-as-you-go pricing model offers cost and performance benefits for running miniature simulations on serverless functions instead of the master node. With serverless functions, costs are incurred only during active periods of execution. Serverless functions run 31 seconds (average across traces) for each 15 minutes optimization window. This makes them more cost-efficient than dedicated instances, due to the memory usage required by miniature simulation, even with sampling. Also, running 200 simulations quickly within a short optimization window requires high parallelism, and provisioning a large dedicated instance that is used only for short intervals would incur higher costs. We evaluate the cost and performance overheads in §3.6.7.

Scaling caches. To scale the OSC, the Macaron controller leverages the elasticity of object storage by storing more cache items to the OSC or deleting objects as needed. For scaling the cache cluster, the Macaron controller tracks the number of cache nodes, deploying new ones or

[3]We used uniform intervals between mini-cache sizes, with the smallest mini-cache size to cover at least 50GB cache simulation.

terminating existing ones when the cluster size changes. It then communicates with Macaron clients to update their cache cluster information for consistent hashing-based message routing. For improved load balancing, availability, and scalability, advanced shard managers like Google's Slicer [193] or Meta's Shard Manager [194] could be employed.

3.6 Evaluation

We evaluate Macaron using real-world object storage workloads to assess the following aspects: its cost-efficiency compared to existing approaches (§3.6.2), its adaptability to workload changes (§3.6.3), cost-breakdown of Macaron's optimization techniques (§3.6.4), its ability to utilize the cache cluster cost-efficiently to achieve the desired performance (§3.6.5), its robustness under varying cloud conditions through sensitivity analysis (§3.6.6), simulation accuracy and prototype reconfiguration overhead (§3.6.7), and its TTL-based variant's cost-efficiency (§3.6.8).

3.6.1 Experimental setup

Traces. We evaluate Macaron with 15 IBM traces, 3 Uber traces, and 1 VMware trace. For brevity, we provide detailed analysis of results obtained from 6 IBM, 1 Uber, and 1 VMware trace (Table 3.2). We use the first day of each trace as the observation period, with optimizations triggered every 15 minutes after the first day. Each evaluation reports the remote data access cost and latency for the remaining days.

For the IBM traces, as in the original paper [101], large objects are divided into 4MB blocks, with each smaller object treated as a separate cache item. We use the same policy for the VMware traces, while for Uber we use 1MB blocks, which is their default policy.

Configurations. Unless stated otherwise, experiments assume workloads running on a different cloud provider than the one hosting the remote data lake, with accesses incurring cross-cloud egress charges. Workloads are located in the N. Virginia region, while the remote data lake is in N. California, but we did perform a sensitivy analysis considering different configurations (§3.6.6). We use AWS's pricing model, but note that cloud providers have similar pricing models.

Baselines and costs. We compare Macaron to three baselines mirroring approaches used today (§3.2.1): accessing all data from a `Remote` cloud, having all data `Replicated` locally, and using existing in-memory caching solutions (`ECPC`) like AWS ElastiCache. Since `ECPC` products rely on users to provide scaling policies, we use Macaron's optimizer to efficiently auto-scale the cache. Finally, we evaluate Macaron against the `Oracular` caching solution (§3.4.4).

At a high-level, `Remote` incurs egress and operation costs for all data accesses, while `Replicated` is plagued by synchronization costs. Egress and capacity costs for `Replicated` are computed based on the rate of increase in total data size, using a 90-day data retention period and 70% dark data, but different portions of dark data are explored too (§3.6.6). We exclude operation costs for `Oracular`'s object storage cache (§3.5). All others incur infrastructure costs for OSC manager and/or Macaron controller, with Macaron and `ECPC` incurring additional serverless function expenses.

Macaron Simulator. Replaying all traces once in a real cloud would cost over $1.5 million (Fig. 3.1b). Thus, we developed a simulator that allows us to assess Macaron across various configurations and constraints. The simulator models key components, replicating their functionalities (§3.3) and interactions. The simulator manages message exchanges between components, including data and control requests for reconfiguration and eviction, ensuring messages are generated in the same way as by our prototype implementation.

To simulate message latencies accurately, we measured data access latencies on AWS for various object sizes accessed from a remote data lake, OSC, and the cache cluster, and fit a Gamma distribution to the collected data.

3.6.2 Cost-efficiency Analysis

Observation 1: *For individual traces, Macaron reduces cross-cloud data access costs by 65% and 75% on average compared to* `Remote` *and* `Replicated`*, respectively.*

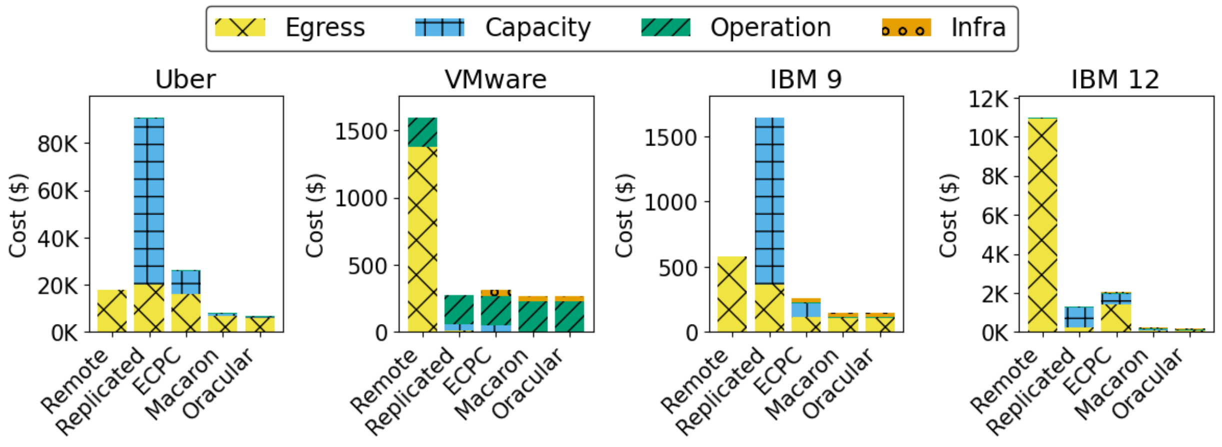
Macaron aims to minimize costs when accessing cross-cloud/region data. We compare remote data access costs of Macaron with those of our baselines. Fig.3.7 shows results for two representative IBM traces for brevity, with a discussion of the overall results provided below.

Overall results. Across 19 traces, Macaron achieves a cost reduction of 2.4% to 99.3% (avg. 65%) compared to `Remote`, and 24.9% to 82.9% (avg. 75%) compared to `Replicated` in cross-cloud scenarios (Fig.3.7b). In cross-region scenarios, for the 16 traces that have lower than 20% compulsory miss ratios, Macaron provides cost reductions of 28.2% to 98.5% (avg. 67.4%) and 18.1% to 91.1% (avg. 78.4%) compared to `Remote` and `Replicated`, respectively (Fig.3.7a). In cross-region scenarios, for the IBM 27, 66, and 96 traces that have high compulsory miss ratios (57%, 79%, 87%), Macaron incurs 24%, 5.8%, and 1.5% higher costs compared to `Remote`, respectively, because the savings achieved are less than the cost of running a single VM to operate Macaron controller and OSC manager. Since cross-cloud egress cost (9¢/GB) is higher than cross-region (2¢/GB), Macaron selects a larger cache capacity for cross-cloud scenarios, allowing for further reduction of egress costs.

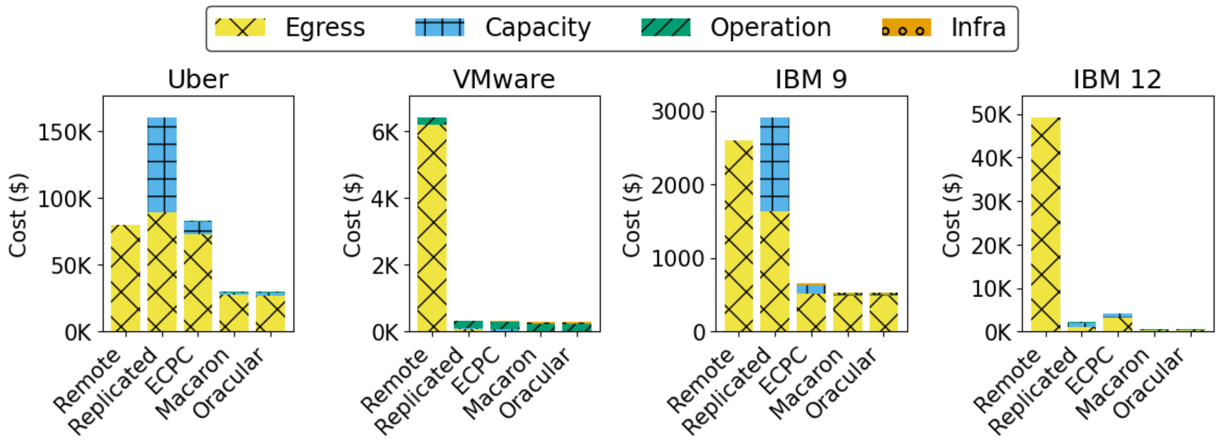
Comparison with `ECPC`. When caching in object storage, the Macaron optimizer exploits its low capacity cost and mitigates egress costs by provisioning high capacities. When caching in DRAM, however, cache capacity costs increase rapidly, leading to a much smaller capacity point for cost optimization. `ECPC`, using DRAM, incurs higher capacity costs even with small capacities, along with increased egress costs compared to Macaron. As a result, across 19 traces, Macaron reduces overall costs by 3.5–89.1% (avg. 46%) compared to `ECPC` for cross-cloud data accesses.

`Remote` vs. `Replicated`. Fig. 3.7 shows that neither `Remote` nor `Replicated` consistently outperforms the other. For cross-cloud scenarios, 6 traces are cheaper to manage with `Remote`, the remaining 13 show cost savings with `Replicated`, and Macaron outperforms both across all traces.

Comparison with `Oracular`. `Oracular` leverages future knowledge for optimal caching decisions, while Macaron relies on past access patterns to predict the future. Still, this results in `Oracular` accessing cross-cloud data at 0.4%-18.3% lower cost (avg. 6.8%) than Macaron across all traces.



(a) Cross-region data access



(b) Cross-cloud data access

Figure 3.7: Detailed analysis on four workloads representing diverse data access patterns.

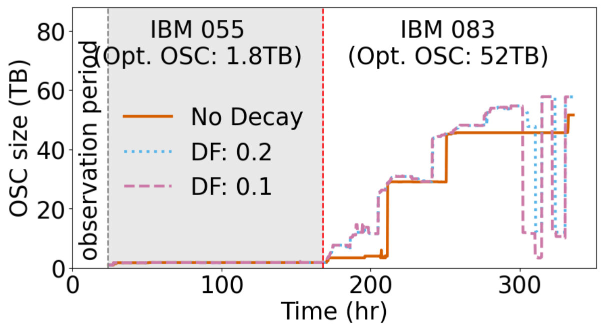
In IBM 12, `Oracular`'s cost is 18% lower than that of Macaron due to misprediction of workload behavior for cross-cloud scenario. For days 1-5, the workload consistently accesses half of the previous day's data, adding an equal amount of new data. On day 6, it accesses data from day 2, causing unexpected cache evictions and remote re-fetching accesses, incurring egress costs.

Despite uncertain workload behaviors, Macaron remains more cost-efficient than the baselines. Monitoring longer-term data access patterns might reduce this uncertainty.

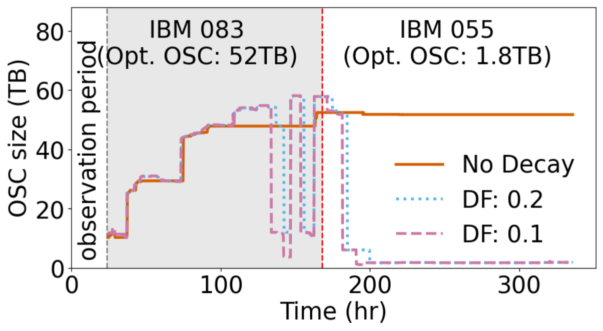
Observation 2: *Macaron's cost reduction stems from aggressively reducing data egress costs by exploiting cheap storage to build large caches, while finding the cache size that minimizes capacity costs.*

We provide a detailed case study for each workload shown in Figure 3.7b. Additional case studies are in Section 3.7.4.

The VMware workload involves running numerous tests periodically on the test dataset, lead-



(a) No decaying leading to high egress cost.



(b) No decaying leading to high capacity cost.

Figure 3.8: Testing Macaron's adaptivity to a workload change (vertical red line), with and without knowledge decaying.

ing to a high frequency of repetitive accesses. Thus, there is a 96% cost reduction compared to `Remote`, and 25% reduction compared to `Replicated`.

The Uber workload has the largest data size among 19 traces, and Macaron achieves 81% cost reduction compared to `Replicated`. Macaron finds a cache capacity of 180TB (56% of total data) yields benefit by avoiding egress transfers.

The IBM 9 workload exhibits a periodic burst for 15 minutes every hour, during which new data retrieval is followed by repeated access. The Workload Analyzer identifies this pattern, provisioning only 1% of the total data size to cache all the data accessed during each burst. This results in 79% cost reduction compared to `Remote`, managing repetitive accesses from OSC, and 82% reduction compared to `Replicated` due to the small cache capacity used.

For the IBM 12 workload, Macaron achieves 98.9% reduction in data egress costs compared to `Remote`, due to strong cache locality. Over 50% of objects are accessed more than 100 times, making caching highly effective. Using `Replicated` is still expensive due to the 101$\times$ higher storage cost compared to caching only hot data identified by Macaron.

3.6.3 Impact of adaptivity mechanisms

Observation 3: *Macaron's adaptive reconfiguration reduces costs by 12% compared to static configurations. When workloads change, Macaron decays its knowledge leading to an additional 5% cost reduction compared to no decaying.*

Macaron optimizes configurations every 15 minutes to match the latest data access patterns. Here, we quantify the benefits of this frequent reconfiguration, and Macaron's mechanism for decaying older learned access patterns.

Reconfiguration window. We evaluate the benefit of Macaron's frequent reconfiguration, by comparing it to a static configuration that is fixed to the optimal capacity obtained from the first day of the trace. For cross-cloud scenarios, Macaron achieves cost reduction 0-85% (avg. 12%) across 19 traces, while for cross-region, it is 0-78% (avg. 8%). When Macaron's reconfiguration window is reduced from 24 hours to 15 minutes, cost is reduced by 0-41% (avg. 4%) for cross-cloud and 0-25% (avg. 3%) for cross-region scenarios.

Exponential decay. By default, Macaron uses a decay factor of 0.2 (i.e., $\gamma^{1day} = 0.2$) to phase out learned patterns. We assess Macaron's cost-efficiency under varying decay factors – 1.0

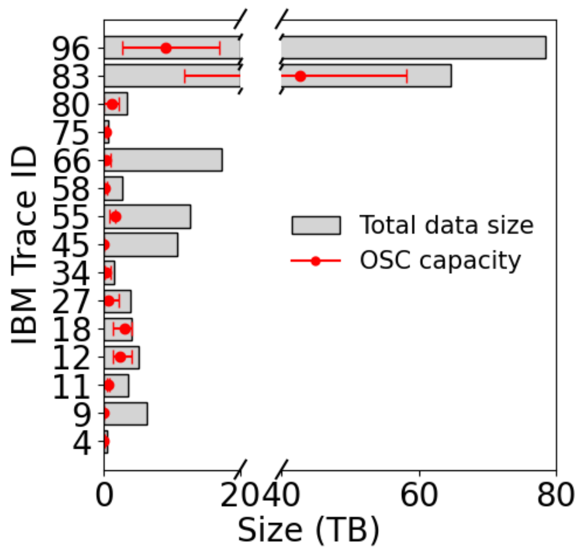


Figure 3.9: Macaron's average OSC capacity (red dot) is 0.8-75.1% of the accessed data size. Error bars show the range over 6 days.

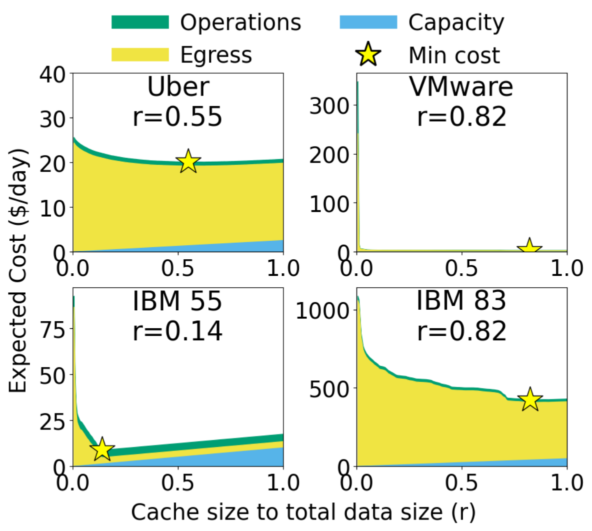


Figure 3.10: Expected cost curves generated by Macaron controller. Sub-optimal OSC size choices can result in higher expenses.

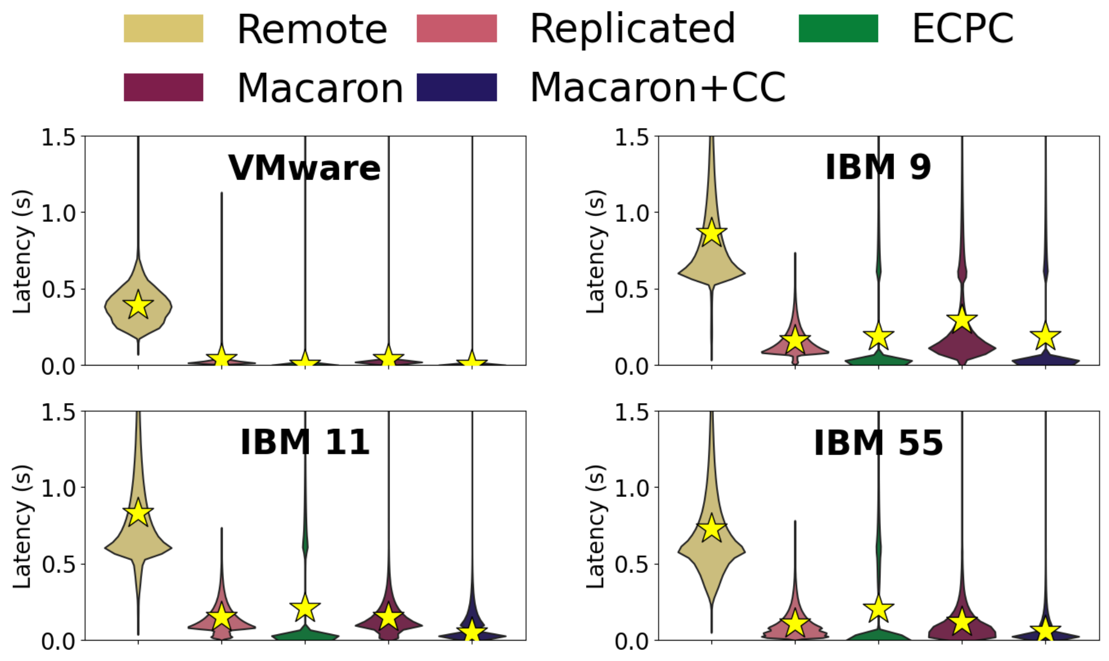


Figure 3.11: Violin plots of latency for each method. By dynamically adjusting cache cluster (Macaron+CC), Macaron reduces latency by 62% and cost by 58% compared to `Replicated`. The star marks the average.

(`NoDecay`), 0.2 (`Default`), 0.1 (`SmallDecay`) – using 15 IBM traces. Specifically, we evaluate its performance (1) within a single trace, and (2) when concatenating two different traces to simulate abrupt changes in data access patterns often observed in real-world workloads [63, 195, 196, 197].

For a single trace, the IBM and VMware workloads span one week, and 18 days for Uber, and exhibit fairly consistent data access patterns. This benefits knowledge accumulated over time. Specifically, our traces show insignificant differences of $\pm 1\%$ with and without knowledge decaying.

To assess Macaron's adaptivity during workload changes we created 30 new concatenated workloads by combining the 6 selected IBM traces and assessed evaluated costs during the second trace's execution to see how Macaron adapts to changes in data access patterns. For 25 concatenated workloads, `Default` and `SmallDecay` reduce costs by 0-30% (avg. 5.2%) and 0-34% (avg.6.1%), respectively, compared to `NoDecay`, which continues to rely on past traces and hampers quick adaptation. For example, combining IBM 55 and IBM 83 in Fig 3.8 results in `NoDecay` facing high egress costs when executing IBM 83 after IBM 55 due to slow scaling out, while changing the order leads to expensive capacity costs because of slow scaling in. However, `Default` and `SmallDecay` can adapt rapidly to such changes.

We observed that for five concatenated workloads[4], `NoDecay` incurred lower overall costs. This was due to the fortuitous alignment of the workloads' unpredictable access patterns with `NoDecay`'s lack of adaptability, preventing it from slowly reducing capacity and retaining unnecessary cache items.

3.6.4 Effects of Macaron optimizations

Macaron's cost savings are primarily attributed to two key optimizations: (1) determining the cost-efficient OSC size and (2) packing small objects when caching them in the OSC. Next, we assess the effectiveness of these optimizations.

Observation 4: *While the cost-efficient cache size varies significantly for each workload, Macaron identifies efficient setups by analyzing each access pattern. Making a less optimal choice can lead to increased costs.*

OSC size optimization. Figure 3.9 depicts the OSC capacity changes over the last six days (after observation period) across 15 IBM traces, with the total data size. The ratio of OSC capacity to data size varies across workloads, ranging 1-98%, highlighting that there is no single ratio ensuring a cost-efficient cache size, and the need for a tool like Macaron.

For IBM 18, Macaron aligns cache capacity with the total data size, suggesting a very large cache can mitigate overall costs by minimizing data egress fees. Unlike conventional caching studies, which favor compact, frequently-accessed data caches, the prohibitively expensive egress costs in cross-cloud, cross-region settings advocate for larger caches.

We found that all traces except one among the 19 evaluated workloads adjusted the cost-efficient OSC capacity at least once, with a standard deviation in the changing OSC size to total data size ratio ranging 0-0.28. The average standard deviation value is 0.1, suggesting that the ratio changed 10% per day, emphasizing the importance of Macaron's adaptivity. This capacity ratio typically increases from day 1 to 7.

Fig. 3.10 demonstrates that erroneous OSC capacity allocations can notably affect cost. Using the same 14% cost-efficient capacity ratio from IBM 55 on IBM 83 causes a $1.5\times$ uptick in

[4]These concatenated traces are 9→18, 18→12, 55→12, 55→18, and 96→12

expected cost relative to Macaron's selection.

Observation 5: *Object packing, especially for workloads with small objects and high request rates, can yield significant savings, with up to a 36% cost reduction.*

Object packing. In our evaluation, IBM 18 and IBM 45 realized cost reductions of 36% and 5%, respectively, due to object packing. Traces with smaller objects and higher request rates, like these two, tend to benefit the most. Similar to Amdahl's law, the higher the contribution of operational costs to the total costs, the greater the potential savings, as object packing impacts only operational costs. Though operational costs average 4% of total costs in cross-cloud scenarios due to high egress expenses (resulting in 3% cost savings), factoring in cross-region egress prices increases operational costs to 8% and savings to 7%.

3.6.5 Macaron with low latency

Observation 6: *Macaron achieves 61% lower latency and 64% cost savings than `Replicated` with its dynamic cache cluster.*

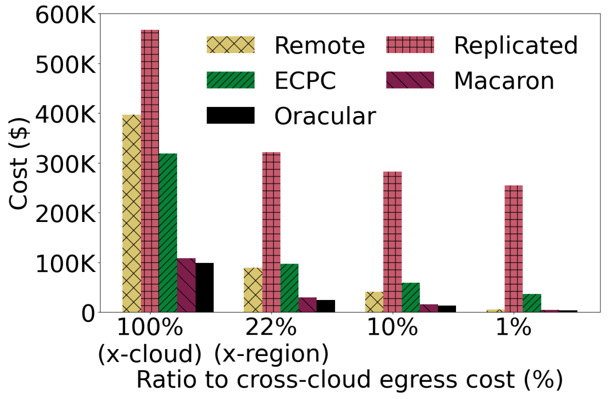
We evaluate whether Macaron can combine performance and cost-efficiency, without compromising on performance. By dynamically adjusting its cache cluster, Macaron cuts remote data access latency by 61% compared to `Replicated`, and still saves 64% costs across 10 traces that showed lower average latency than `Replicated`. This is achieved by serving hot data from cache cluster and smartly scaling the cluster during inactive periods or when smaller capacities suffice. Macaron without cache cluster exhibits, on average, 10% higher latency compared to Replicated. This increase is because Macaron 's latency is lower bounded by the latency of object storage, which is the same as that experienced by `Replicated`. However, using 30% more costs for the cache cluster (still significantly cheaper than `Replicated`), Macaron substantially improves latency.

For 6 remaining IBM traces and Uber traces with high compulsory miss ratios, only full replication achieves low latency, albeit at the previously discussed high cost in §3.6.2. Our rough estimates indicate that if the compulsory miss ratio surpasses 10-20% (depending on workload object size), achieving lower average latency than local object storage becomes challenging, even with all other requests retrieve data from the cache cluster, which matches with our results.

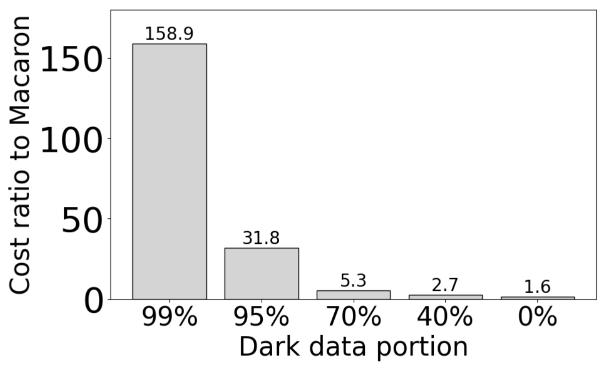
Fig. 3.11 shows the violin graphs that illustrate the latency distributions of four traces, all showing similar characteristics. Interestingly, despite `ECPC` being DRAM-centric solution, it often shows higher latency than Macaron with a cache cluster (Macaron+CC) or even without a cache cluster (IBM 11 and 55 in Fig. 3.11), as not enough DRAM cache server is allocated by prioritizing cost-efficiency, thus resulting in high latency. Specifically, in Fig. 3.11, Macaron+CC demonstrates cost savings of 0.3%, 9%, 37%, and 16% for the VMware, IBM 9, 11, and 55 traces, respectively, while also reducing latency by 3%, 1%, 76%, and 70% compared to `ECPC`.

The plot also shows Macaron's tail latency without a cache cluster resembling `Remote`, while its low latency mirrors `Replicated`. With a cache cluster, Macaron's low latency distribution matches cache cluster latency, substantially lowering average latency.

Our percentile latency analysis further validates these observations. For example, in IBM 9 with a 21% compulsory miss ratio, Macaron's p90 and p99 latencies using DRAM cache are



(a) Cost comparison across pricing models.



(b) Effect of dark data on cost of `Replicated` relative to Macaron.

Figure 3.12: Analysis of the efficiency of Macaron under varying pricing models and with changing dark data portions.

from remote data accesses, but are 27% and 15% lower than `Remote`'s p90 and p99, indicating the impact of serving many requests from OSC. In IBM 55, where the compulsory miss ratio is below 0.1%, Macaron's p90 and p99 latencies from OSC using DRAM cache are even 15% and 6% lower than `Replicated`'s p90 and p99, showcasing the efficiency of serving from the cache cluster.

3.6.6 Sensitivity analysis

We assess Macaron under varying experimental settings, specifically latency, egress cost, and dark data portions.

Different egress costs. We tested Macaron with three alternative egress cost models to ensure its effectiveness across different pricing: 22% (cross-region egress cost), 10% (0.9¢/GB), and 1% (0.09¢/GB) of the standard cross-cloud rate of 9¢/GB. Macaron consistently surpasses the baselines across different pricing models, achieving substantial cost reductions as depicted in Fig. 3.12a, even when egress costs are as low as 1% of the cross-cloud rate.

Different latency. We evaluate Macaron's cost-efficiency with varying latency distributions by switching the inter-region setting from US N. Virginia and US N. California to US N. Virginia and Europe Frankfurt. In this new scenario, higher inter-continent latency makes it harder to achieve local object storage access latency with Macaron. Consequently, one less trace outperforming `Replicated` for both cost and latency compared to the intra-continent scenario, achieving a 71% lower average cost while paying 62% less.

Different dark data percentage. We evaluated Macaron's efficiency against `Replicated` across varying dark data portions, previously set at 70%. Fig. 3.12b shows that with a 0% dark data portion, which means the working set size of the trace is equal to the entire data size, Macaron is 37.5% cheaper than `Replicated`. At 99% dark data, `Replicated` is 158.9× costlier than Macaron.

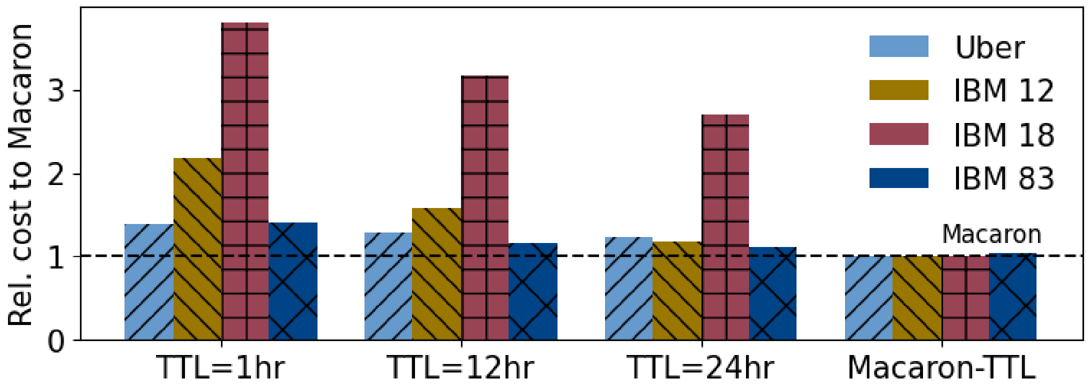


Figure 3.13: Comparison of Macaron against static TTL caches and Macaron-TTL shows that dynamic cache adjustments of Macaron's variants result in cost savings compared to static TTL caches.

3.6.7 Simulation accuracy & Reconfiguration overhead

Observation 7: *Simulator closely mimics Macaron with minimal gaps in cost and latency, up to 0.17% and 7.6%. Reconfiguration time comprises less than 9% of the total runtime, while the cost overhead remains low at 0.6% of the total cost.*

We evaluate Macaron simulator's accuracy by comparing its cost and latency results with those obtained from running our prototype implementation on AWS. Due to budget constraints, we selected three IBM traces: IBM 9, 55, and 58, representing read-only, read/write mixed, and read/write/delete mixed scenarios, respectively. Overall, the cost gap between the simulator and prototype was minimal, ranging from 0.08% to 0.17%. Similarly, the average latency gap was 4-7.6%. Additionally, we validated the number of Get operations hit at each cache level match.

Next, we evaluate reconfiguration overhead. Across the three traces we evaluated, end-to-end reconfiguration took 6 to 418 seconds (avg. $71sec$). When there is no change in cache cluster configuration, it takes only 7 seconds on average, but if there is a change, especially when scaling out, it takes 274 seconds on average. Since reconfiguration occurs only when optimization results in configuration changes, the total reconfiguration time across three traces amounts to 1.6 hours, representing just 9% of the total runtime of 18 hours. During reconfiguration, requests continue to be served without any downtime, and the cost of the resources required to carry it out are factored into Macaron's savings.

Further breakdown reveals that the largest portion of reconfiguration time is spent on miniature simulation and cache cluster reconfiguration. Miniature simulation time is proportional to the request count in the optimization window, taking 0.3–44 seconds (avg. $31sec$ across all optimization windows of 19 traces). Cache cluster reconfiguration, including VM initialization and cache priming, took 132–387 seconds (avg. $256sec$). Finally, the cost overhead of running miniature simulation on AWS Lambda is negligible, accounting for only 0.003–4% (avg. 0.6%) of the total cost of running each trace end-to-end across 19 traces.

3.6.8 Size-based and TTL-based Macaron variants

Observation 8: *Macaron and Macaron-TTL demonstrate similar cost-efficiency, successfully identifying cost-efficient cache sizes and TTLs.*

We evaluated the effects of optimizing TTL (Macaron-TTL) rather than cache size (Macaron) in cross-cloud scenarios. Across 18 traces, Macaron-TTL's cost ranges from -0.8% to 3.3% compared to Macaron, indicating similar performance. In IBM 80, Macaron-TTL was 17% more expensive due to its TTL=24hr selected based on the past data access pattern, which led to all data being forcibly evicted during a two-day no-access period, whereas Macaron saves more in egress costs by avoiding evictions.

We assessed Macaron-TTL 's ability to identify the most cost-efficient TTL for each trace. Through exhaustive search, we tested various static TTL caches throughout each traces[5], and pinpointed those that minimized costs. In the IBM traces, we observed that the optimal TTLs varied widely, ranging from 1 to 168 hours, with an average of 72 hours and a standard deviation of 56 hours. Despite this variability, Macaron-TTL accurately identified the optimal TTLs for 16 traces. For IBM 34, 45, and 58, although Macaron-TTL selected TTLs of 144, 132, and 84 hours – differing from the optimal TTLs of 72, 108, and 24 hours – the cost gaps between the static TTL policies using TTLs chosen by Macaron-TTL and the optimal TTLs were negligible, all under 0.7%. This is because OSC capacity costs are significantly lower than egress costs, so the additional capacity cost has minimal impact.

Fig. 3.13 illustrates a cost comparison between Macaron and Macaron-TTL against static TTL policies. Across 19 evaluated traces, Macaron achieved average cost reductions of 22%, 13%, and 9%, with maximum reductions of up to 74%, 69%, and 63% compared to static TTL policies set at 1, 12, and 24 hours, respectively. This highlights the importance of dynamic cache adjustments for enhancing cost-efficiency.

3.7 Supplementary material

We provide detailed information on the trace collection process, the cloud resources used for prototype evaluation, additional evaluation results validating the Macaron simulator, and the details of the Macaron-TTL algorithm.

3.7.1 Trace collection details

Uber trace. We collected object access logs generated from Uber's Presto workload in their production system. To ensure no impact on production, we collected logs with a spatial sampling with a sampling ratio at 1% from three Presto engines over 18 days. To confirm that 1% sampling retains workload characteristics, we collected a two-hour trace without sampling and performed the same sampling method, where we observed small differences of 7%, 2%, and 8% in request count, accessed data size, and object size, respectively. Therefore, in our evaluation, we scaled the sampled traces by 100× and presented detailed results of one of the three traces where we

[5]For this exhaustive search, TTL intervals of 1 hour, 6 hours, and then every 12 hours up to the full trace length were used (e.g., 1, 6, 12, 24, 36, 48, . . .).

confirmed that they share very similar data access characteristics. Due to the fact that over 70% of the accesses are generated by periodic jobs, the workload exhibits a stable data access pattern during the collection period.

VMware trace. VMware trace involves AWS S3 requests generated by AWS Athena queries from VMware's test infrastructure, spanning an 8-day period. We collected logs by enabling Amazon S3 server access logging service to catch data access requests sent to S3 buckets.

3.7.2 Cloud resource types used for evaluation

For the cache cluster, we used Redis as a distributed in-memory cache solution as it is well-supported, performant, and effectively handles large objects, unlike Memcached.

For both simulator and prototype evaluations, we used the same VM types for consistency. We used an `r5.xlarge` instance for the master node on AWS, which we confirmed had sufficient resources. Since the Macaron controller's workload analysis, a computation-heavy task, ran on AWS Lambda, the CPU power of the `r5.xlarge` instance was adequate. Cache nodes also used the `r5.xlarge` type, which provides 32 GiB of memory. However, our observations showed that the Redis server typically utilized around 26 GiB, aligning with the `cache.r5.xlarge` specification of ElastiCache. As a result, we assumed 26 GiB of memory for the cache nodes in the simulator as well. Additionally, we used AWS Lambda functions with 8 GiB of memory, which provided sufficient resources to run the miniature simulations quickly and cost-effectively.

3.7.3 Cost-efficiency analysis across all traces

Fig. 3.14 and Fig. 3.15 present the cost comparison results of all the traces we evaluated and we explain two more case studies.

In IBM 83 (similar to IBM 55), ranking second in total data size accessed among 15 IBM traces, Macaron achieves 86% cost reduction compared to `Replicated`. Using an average cache capacity of 52 TB (81% of the total data size) and given the low price of object storage, Macaron prioritizes overall cost-efficiency over minimizing capacity costs and incurring higher data egress expenses associated with `Replicated`.

Despite similar total data sizes for IBM 96 and 83, Macaron allocates only 7% of the total data size as cache capacity for IBM 96 due to low data access skewness (Zipfian α=0.2) and a high cold miss ratio. Larger caches don't effectively reduce egress costs for such workloads, yet Macaron remains 1.4% and 81.7% cheaper than baselines for IBM 96.

3.7.4 Details of simulator accuracy evaluation

We carefully designed experiments to validate whether the simulator and prototype yield consistent results, both in terms of cost and performance, across different scenarios: one where Macaron utilizes a cache cluster for performance, and another where it solely relies on OSC to minimize costs. To reduce cost of running experiments without compromising the validity of results, we implemented the following approach.

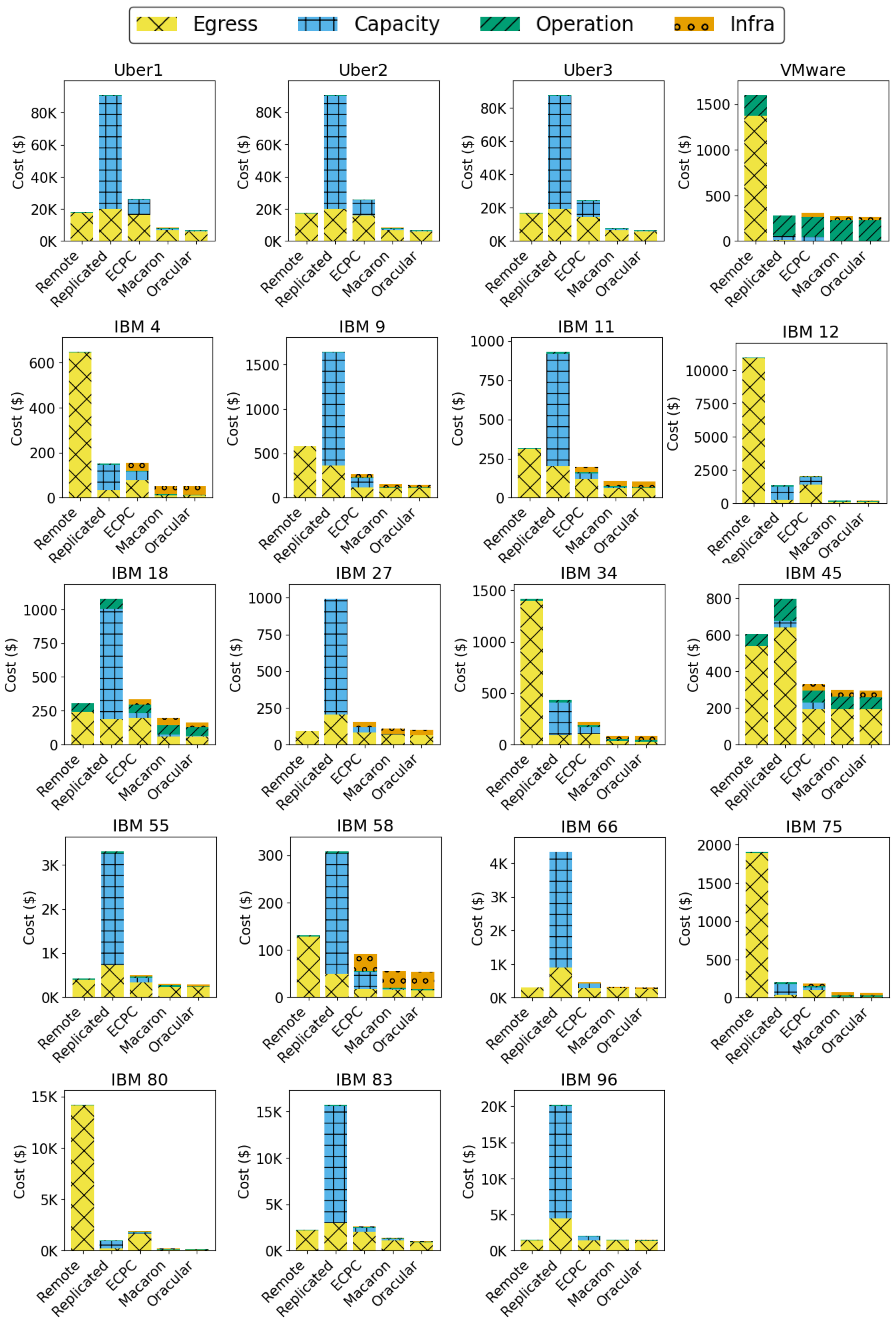


Figure 3.14: Cross-region data access cost analysis for all 19 traces.

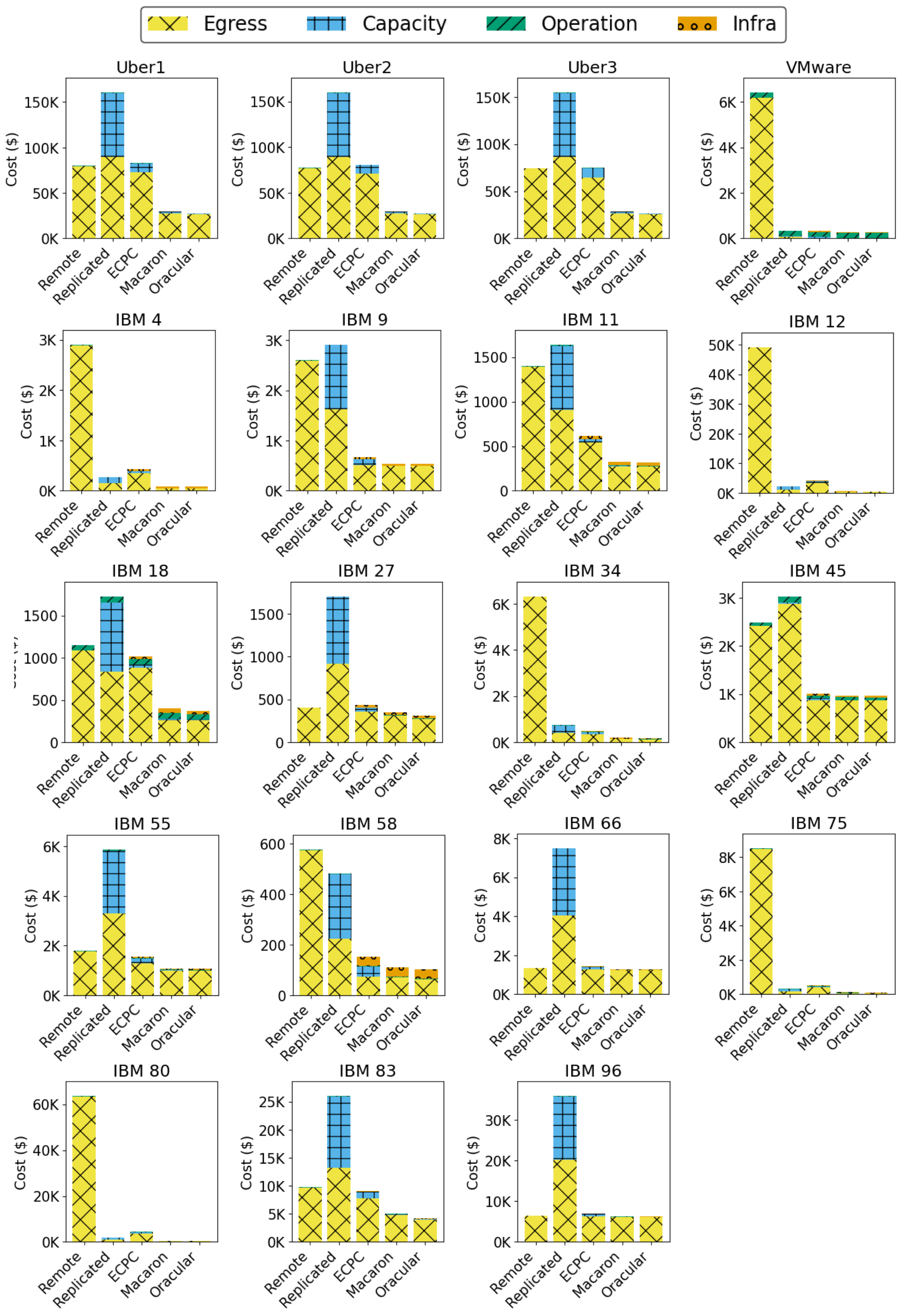


Figure 3.15: Cross-cloud data access cost analysis for all 19 traces.

Trace	Total costs ($)		Get hits at each level		Avg. lat (s)	
	Sim	Pro	Sim	Pro	Sim	Pro
IBM 9	5.86	5.87	45:35:20	46:34:20	0.24	0.26
IBM 55	11.83	11.82	50:5:45	47:9:44	0.30	0.28
IBM 58	2.61	2.60	40:0:60	39:1:60	0.47	0.49

Table 3.3: Detailed results of the accuracy evaluation between the simulator and prototype, with simulator results on the left and prototype results on the right. The comparison includes total costs, the number of Get hits at each level (cache cluster:OSC:remote data lake), and the average latency after running the trace.

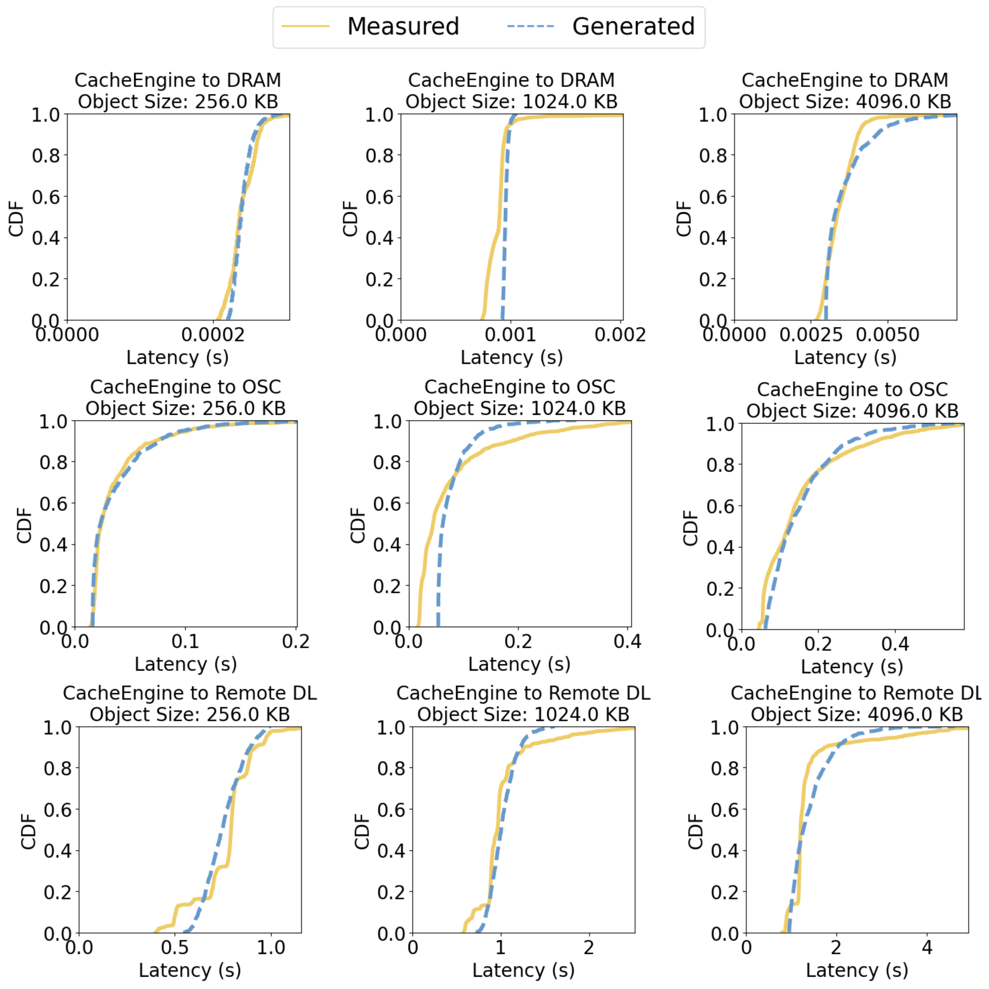


Figure 3.16: Comparison of latency distributions between cache engine and data sources generated by Macaron simulator's latency generator and those observed in measurements.

For Macaron without cache cluster, we ensured both prototype and simulator execute the same reconfiguration for each optimization window and validated the total costs are the same by executing traces end-to-end. We sampled the requests with spatial sampling at 1% and accel-

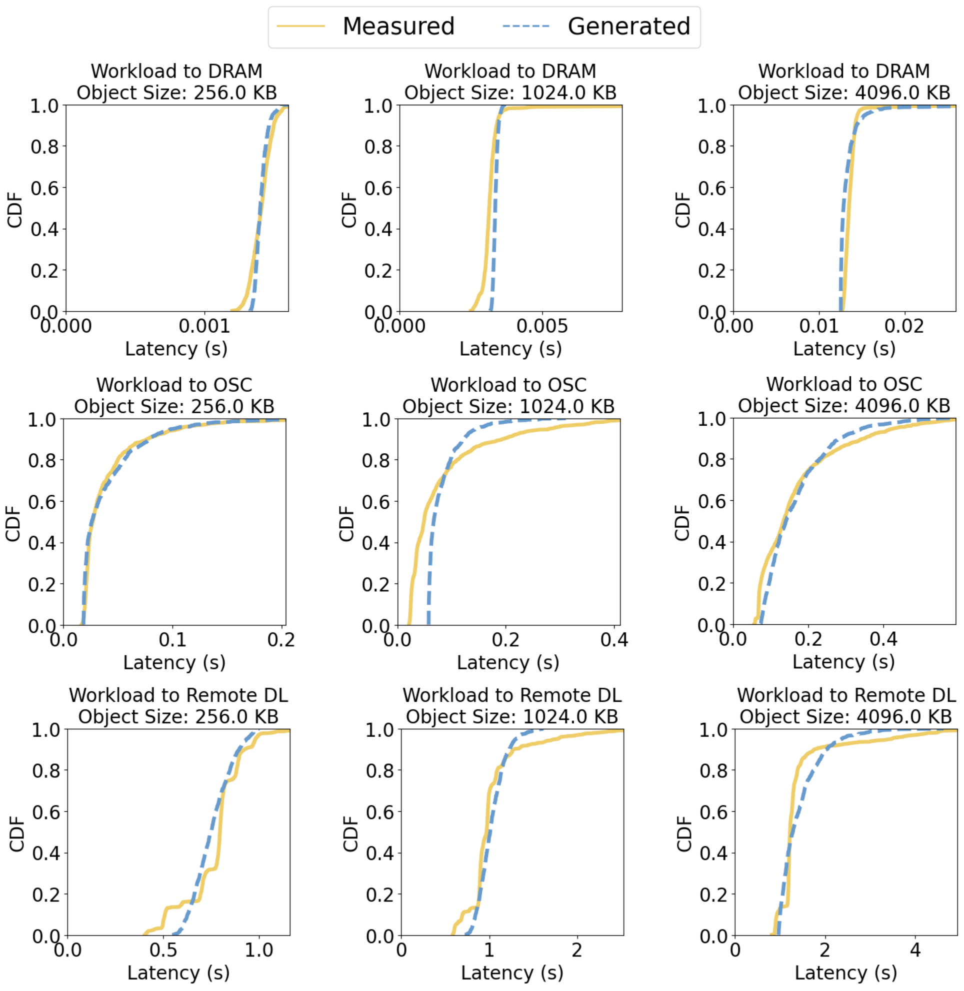


Figure 3.17: Comparison of end-to-end data access latency distributions generated by Macaron simulator's latency generator and those observed in real measurements.

erated execution by 10× to reduce costs and speed up experiments. Reconfiguration occurred every 3 minutes after a 3-hour observation period to make both simulator and prototype to run as many optimizations as possible and show the results are still matching each other. These adjustments do not compromise the validity of cost comparisons.

For Macaron with cache cluster, we focused on comparing average latency results between the prototype and simulator. In this experiment, rather than sampling the trace, we truncated the trace to the first 6 hours and commenced optimization after 30 minutes, with 15-minute optimization windows, maintaining the original request rate to compare latency correctly.

Table 3.3 shows the detailed data of the results we presented in §3.6.7.

3.7.5 Latency generator evaluation

As outlined in §3.6.1, Macaron simulator's latency generator fits a Gamma distribution to the latency measurements taken from the cloud. This distribution is then used to simulate latencies between each component in the system. Here, we verify the accuracy of the latency generator in reproducing a latency distribution similar to the real measurements.

Fig. 3.16 illustrates the cloud-measured latency distribution alongside the distribution generated by the simulator's latency generator for each object size, between the cache engine and each data source (cache cluster, OSC, and remote data lake). Across the average latencies for different object sizes, the mean absolute percentage error was low at 2%.

Fig. 3.17 depicts the end-to-end latency distribution for retrieving data from each source. We further confirmed the simulator's fidelity to the measured latency by showing a mean absolute percentage error of 1.5% between the average latencies.

3.7.6 Detailed algorithm for Macaron-TTL

We extended Macaron 's optimization technique to develop Macaron-TTL. While the core optimization workflow of Macaron, described in §3.4, remains unchanged, the *expected cost curve* explained in §3.4.1 now uses TTL as its parameter instead of cache size:

$$\begin{aligned}
&TotalCost(TTL) = OSCCapacityCost(TTL, GarbageSize) \\
&\qquad\qquad + EgressCost(TTL) + OpCost(TTL) \\
&EgressCost(C) = EgressPrice * ByteMissCurve(TTL) \\
&OpCost(C) = PutPrice \times \left(\frac{\#Writes + \#Reads \times MissRatioCurve(TTL)}{\#Objects\ per\ Packing\ Block} \right)
\end{aligned}$$

To calculate this, we adapted the miniature simulation to use TTL as the X-axis for the miss ratio curves and bytes miss curves. While OSC capacity was straightforward to calculate when using capacity as the parameter, we now need to compute OSC capacity for each TTL during the simulation, producing an OSC Capacity Curve.

For the miniature simulation, we continue using spatial sampling for data access requests. However, we no longer reduce the cache size for simulating mini-caches since cache size does not affect cache eviction under TTL cache. After the simulation, the measured miss ratios from each mini-cache are used as is. Bytes missed are divided by the sampling ratio, similar to Macaron, and OSC capacity is also divided by the sampling ratio to reconstruct the original workload's MRC, BMC, and OSC Capacity Curve before sampling.

3.8 Related work

Cache auto-configuration. Prior works have focused on enhancing performance by reallocating a fixed-capacity shared memory between application runtime needs and caching. Robinhood [198] redistributes cache resources across backend services to reduce latency variability and tail latency. LAMA [115] adjusts Memcached's memory partitioning to improve miss ratios and response times. Memshare [199] reallocates memory among applications to maximize hit rates using idle CPU and memory bandwidth. Sundarrajan et al. [200] enhance CDN cache provisioning using footprint descriptors. D3N [201] adjusts cache sizes to improve big-data job performance and reduce network traffic. While they focus on dynamic reallocation of fixed memory, Macaron assumes total cache capacity is elastic at a cost, and utilizes object storage to reduce that cost. While some studies [121, 192, 202, 203] explore dynamic cache provisioning, they mainly focus on using DRAM or Flash as cache storage medium, and do not account for expensive data transfer costs as part of the miss penalty, which plays a significant role for Macaron.

Optimizing cloud resource costs. With the growing trend of migrating workloads to public clouds, prior work [20, 25, 26, 37] has explored methods for cost-efficiently provisioning cloud storage for various applications. InfiniCache [78] has focuses on cost-effectively utilizing serverless functions for caching data, and there are studies [79, 80] optimizing the use of object storage for file systems and backup solutions, and our packing strategy is based on them. Others [71, 204] have leveraged spot instances] to reduce VM costs by utilizing the affordability of ephemeral VMs, and some of their ideas could be applied to Macaron when dynamically scaling DRAM cache servers. Skyplane [82] aims to optimize egress network resources in public clouds. However, they primarily focus on methods for cost-efficient one-time transfers of bulk data. Macaron's emphasizes optimizing data access costs for long-running workloads with repetitive data access patterns.

Multi-cloud data management. Previous works [83, 84, 85, 86, 205, 206, 207, 208] explored optimizing data placement across clouds or regions for fault tolerance, latency, and cost-efficiency, allowing free data migration or replication across regions. In contrast, Macaron addresses scenarios where data placement is already determined and data cannot be freely migrated due to cost, proximity to users or sources of data generation. In such cases, we need to run applications across regions or clouds as use cases described in §3.1. In these contexts, an auto-configuring caching solution is more suitable than data placement optimization. We believe both approaches are orthogonal and complementary.

Cache replacement policy. Prior work [102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112] focuses on cache replacement policies to enhance miss ratio or latency. However, in the unique context of multi-cloud, multi-region environments, where high egress costs serve as a significant miss penalty and cache capacity cost is very cheap and elastic, even if replacement policy is not optimal, Macaron can extend cache capacity accordingly with minimal costs, making determining the cost-efficient cache capacity more crucial than refining cache replacement policies. Our evaluation, comparing Macaron to `Oracular`, supports this notion. While `Oracular` represents an optimal solution for both cache replacement and capacity determination, our results demonstrate that solely achieving the right cache capacity without exploring replacement policies can yield results close to those of `Oracular`.

Existing approaches. Major cloud providers like AWS, Azure, and GCP already support cross-region data replication [90, 91, 92], and companies such as Snowflake [93] and Juicedata [94] have recently introduced cross-cloud data replication capabilities [209]. There are existing caching services [95, 96, 97] provided by cloud providers that are built on distributed in-memory stores [210, 211]. The third-party services like Alluxio [98], MinIO [99], and Avere [100] support cloud-native cache solutions using memory or flash devices. However, their primary goal is achieving high performance for accessing local data and are not optimized for cross-cloud or cross-region data access costs. Our `ECPC` baseline mimics these approaches but is enhanced by intelligent auto-scaling, yet Macaron remains more cost-efficient.

3.9 Summary

Macaron addresses the high costs of cross-region and cross-cloud data access by introducing an auto-configuring cache that dynamically adjusts its size and storage tiering to minimize both latency and dollar cost. Unlike traditional caches limited by hardware, Macaron treats cache capacity as a tunable cloud resource and leverages cheap object storage to scale effectively. Using a lightweight miniature simulation, it continuously estimates the cost–latency trade-off of different configurations and adapts to changing workload behavior. Across real-world traces from IBM, Uber, and VMware, Macaron reduces access costs by up to 66% compared to cloud-managed caches and achieves near-optimal performance without requiring future knowledge.

Chapter 4

Macaron+: Cost-aware cross-region cache prefetching

As explained in Macaron [21], cross-region data access costs in public clouds can be reduced by auto-configuring cache parameters and leveraging low-cost object storage as a caching layer. While Macaron also reduces data access latency for repeated accesses, it remains ineffective for first-time accesses, which are inherently uncacheable—particularly in workloads with high cold miss ratios. For instance, as discussed in Section 3.6.5, 9 out of 19 traces failed to achieve average data access latency lower than that of local object storage, even with the assistance of a DRAM layer, due to their high compulsory miss ratios.

Some organizations reduce latency by replicating all data locally [90, 91, 92, 93, 94], but this approach is $1.2\times$ to $11\times$ more expensive (average $4.6\times$) than Macaron, as shown in Section 3.6.2. Cache prefetching offers another potential approach to reducing latency beyond what Macaron achieves, but existing prefetching algorithms [212, 213, 214, 215, 216, 217, 218, 219, 220] are incompatible with caches that support highly elastic capacity like Macaron, or they fail to consider the unique characteristics of cloud object storage workloads (§4.3).

A key reason existing prefetching algorithms fail under Macaron's highly elastic cache is that they are designed to prefetch recently evicted blocks based on past accesses, whereas Macaron requires prefetching blocks of objects that have not yet been accessed at all. Moreover, many prior techniques rely on spatial locality in a global address space, learning correlations between consecutively accessed block addresses to predict future accesses. However, cloud object storage systems do not provide a global address space and applying such address-based methods is either infeasible or yields poor results in practice.

To overcome these limitations, we propose **Macaron+**, a prefetching technique tailored to cloud object storage and Macaron's cache model. Macaron+ does not rely on global address space or temporal ordering. Instead, it infers access pattern similarity across objects and uses that similarity to predict which blocks of a newly accessed object should be prefetched, based on the behavior of similar objects. Our key insight is to shift focus from temporal correlation to pattern-based similarity across objects, enabling more effective prefetching for structured object formats such as Parquet or ORC, where similar workloads often exhibit predictable access patterns.

We analyze data analytics SQL workloads from Uber's Presto cluster and identify three key insights regarding object-level access patterns. First, most objects in the trace have a subset of other objects that exhibit similar access patterns. Second, many objects display a degree of consistency in the changes in the access pattern, which means that early access behavior can be used to predict later access behavior. Third, we observed a correlation between object access patterns and file path metadata, in which objects with similar path prefixes are more likely to exhibit similar access patterns.

Our current study focuses on analytics workloads as a representative case, where large-scale SQL queries access data stored in object storage backends. These workloads typically operate on structured object formats such as Parquet or ORC, which exhibit consistent and traceable access patterns that are well-suited for evaluating object-level prefetching strategies. Expanding Macaron+ to support a wider range of workloads, including unstructured application logs, machine learning pipelines, or media content delivery, remains a promising direction for future research.

Based on these findings, we designed a prefetch mechanism that, upon access to each object, uses the object's observed access pattern so far to predict its future access pattern via a lightweight MLP model. Based on this prediction, we employ an approximate nearest neighbor (ANN) search to identify other objects expected to have similar future access patterns. Finally, we reference the block offsets previously accessed from these similar objects to prefetch the corresponding blocks for the current object.

We evaluated Macaron+ using Uber's object storage access traces and found that our access pattern similarity-based prefetching can effectively trade off cost and latency, outperforming existing prefetching approaches in terms of cost-efficiency. While our evaluation shows that a simple sequential prefetching can offer comparable performance, our results highlight that there remains a promising design space to improve the cost effectiveness of the prefetching strategy of Macaron+.

Contributions. (1) We analyze a real-world cloud storage workload and derive key design rationales for prefetching data blocks. (2) We propose a cost-efficient prefetching technique that leverages access pattern similarity between objects to guide block prefetching. (3) We explore multiple variants of the Macaron+ prefetcher, characterizing the cost–latency trade-off space. (4) We demonstrate that Macaron+ achieves a 13–61% cost reduction for the same latency target compared to existing approaches, and identify opportunities for further improvement.

4.1 Motivation and challenges

The demand for multi-region databases continues to grow, driven by globally distributed applications and the need for high availability and low-latency access across regions. In response, many cloud-based database systems now support multi-region deployments [221, 222, 223, 224, 225, 226, 227]. In public cloud environments, there are two predominant approaches for accessing database files stored in object storage across multiple regions.

The first approach, illustrated in Figure 4.1a, replicates the storage layer across regions by continuously synchronizing data from a primary region to secondary regions. This strategy, employed by systems such as Amazon Aurora Global Database [221], Azure Cosmos DB [224], and CockroachDB [225], enables low-latency access in remote regions. However, it introduces additional costs due to the need to replicate and persist data that may never be accessed, as well as the overhead of maintaining synchronized storage across regions.

The second approach, shown in Figure 4.1b, avoids full replication by relying on a centralized metadata store (e.g., AWS Glue Data Catalog, Hive Metastore) to locate required data files, which are then fetched remotely across regions. This model, adopted by services such as Amazon Athena [222] and Google BigQuery external tables [227], reduces replication and storage costs. However, it suffers from increased latency for remote data accesses and high inter-region egress costs, particularly for hot objects that are accessed frequently.

To strike a balance between these two extremes, our prior work, Macaron cache, proposes a hybrid design that caches hot data near compute regions and dynamically reconfigures the cache based on workload characteristics (Figure 4.1c). While this approach can reduce remote access costs, it is insufficient for workloads with high compulsory miss ratios. As shown in §3.6.5, caching alone cannot adequately mitigate access latency in such cases, motivating the need for complementary prefetching techniques.

While cache prefetching has the potential to reduce average data access latency, its effectiveness is fundamentally limited by the lack of precise knowledge about future access patterns. Inaccurate prefetching decisions can not only fail to improve performance but also incur additional costs, particularly in cross-region settings where remote data transfers are expensive.

Challenge 1: Cache pollution. Cache pollution arises when prefetched data displaces useful data from the cache due to limited capacity. Techniques such as bounding the volume of prefetched data [214, 228] or leveraging domain-specific heuristics [229, 230] can alleviate this issue to some extent. However, they are insufficient in resolving the fundamental contention for limited cache space, and often rely on static or manually tuned configurations that yield suboptimal outcomes. In geo-distributed cloud environments, evicting and re-fetching the same data repeatedly can significantly amplify remote access costs.

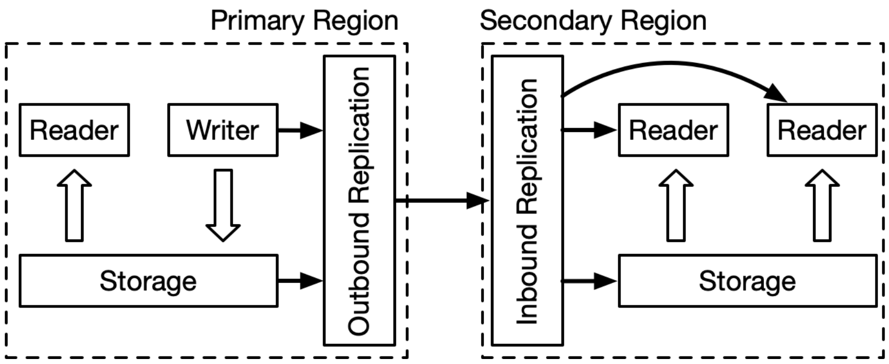
Challenge 2: Wasted prefetches. Prefetching mechanisms may speculate incorrectly, loading data that is never subsequently accessed. In cross-region deployments, such mispredictions translate into unnecessary inter-region data transfers, inflating costs without contributing to latency reduction. As remote access charges in public clouds are tied to actual egress volume, even a modest rate of misprediction can lead to disproportionately high expenses.

Takeaway: *These challenges highlight the need for a cost-aware and selective prefetching strategy that minimizes unnecessary data movement while maximizing prefetch coverage – the fraction of cache misses successfully eliminated by prefetching.*

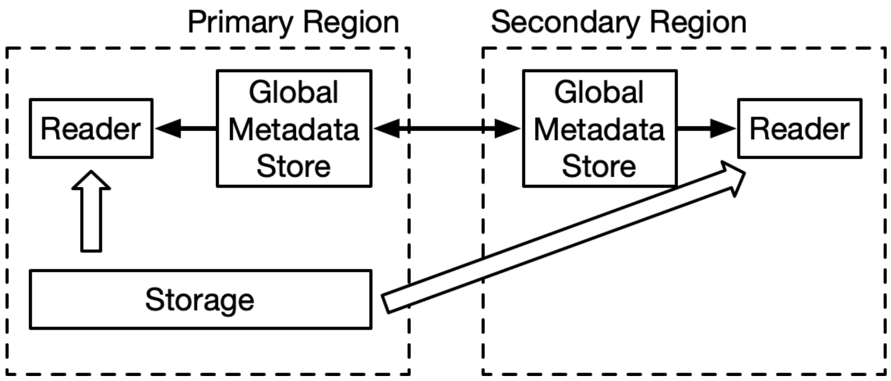
4.2 Prior work

Prefetching is crucial to improve performance of caching, and numerous prior studies have proposed diverse approaches. In this section, we examine these existing methods and explain why they fail to address the unique requirements of multi-region object storage workloads in the public cloud.

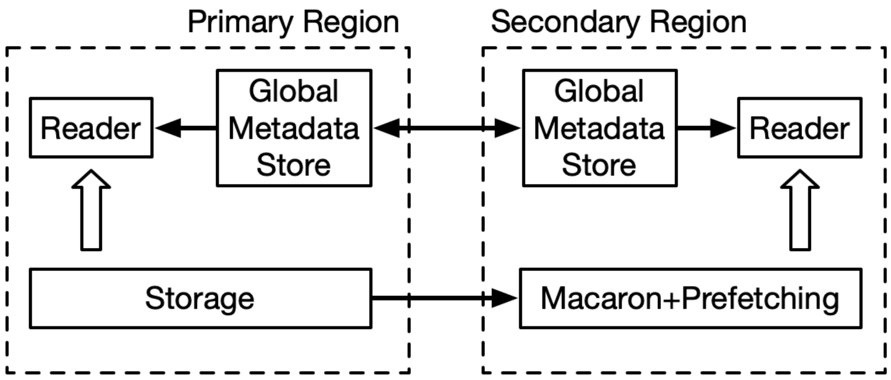
History-based prefetching. Several works, including C-Miner [212], Mithril [213], and Tombolo [214], focus on analyzing past access patterns to identify block correlations. These methods build graphs or utilize data mining techniques to determine related blocks and predict future accesses based on historical behavior. However, they primarily exploit previously accessed



(a) Replicated approach (e.g., AWS Aurora)



(b) Remote approach (e.g., AWS Athena)



(c) Macaron+: Macaron cache with prefetching

Figure 4.1: Two existing approaches and our proposal for supporting cross-region database data access. Our proposal reduces remote data access costs by caching hot data locally and enhances performance through prefetching.

blocks, limiting their effectiveness in scenarios like ours, where cache size can elastically expand (as in Macaron), making accurate prefetching of previously unseen blocks critical.

Sequential prefetching. Sequential-based prefetching, such as AMP [215], SARC [216], and TaP [217], assume strictly sequential access patterns and prefetch consecutive blocks. Although these approaches attempt to enhance naive prefetching by auto-configuring prefetch degree, prefetch cache size, or timing of prefetch operations, they still exhibit poor accuracy for complex or non-sequential workloads. Consequently, they lead to unnecessary data transfers, quickly degrading cost-efficiency relative to latency improvements.

ML-driven prefetching. Recent machine learning-driven prefetching techniques, including SGDP [218], SeLeP [219], DeepPrefetcher [220], and learning-based prefetching for SSDs [231], predict future accesses based on learned patterns or logical block address (LBA) deltas. These methods perform effectively in environments with unified address spaces, such as traditional file systems, where it is straightforward to discover learnable patterns from single block address space. However, in object storage workloads with isolated object address spaces, such as ours, applying similar ML approaches proves ineffective due to the absence of a unified global address space, as demonstrated in Section 4.5. Furthermore, most of the ML-driven techniques identify relationships only among previously accessed blocks, thus failing to address the challenge of prefetching previously unseen blocks.

Summary: *Existing prefetching approaches rely on inter-block correlations among previously accessed blocks and assume a global address space, making them ineffective in our setting, where prefetching must target unseen blocks and each object has its own isolated address space. While sequential prefetching can handle unseen blocks, it cannot capture complex patterns, highlighting the need for a new approach.*

4.3 Block-level access patterns in cloud storage

Based on analysis of the block-level access pattern of the cloud object storage workload from Uber's Presto cluster, we identify three key characteristics that motivate Macaron+ design: (1) many objects exhibit similar block-level access patterns (§4.3.1), (2) early access patterns are often predictive of future access behavior (§4.3.2), and (3) file path prefixes correlate with access patterns (§4.3.3).

Why does Macaron+ cache and prefetch at the block level? Access traces [123] from Uber's Presto cluster show that queries to Parquet files stored in object storage typically retrieve only small portions of each file. This is achieved through S3 Get operations with byte-range requests, which leverage Parquet's predicate pushdown capabilities. As illustrated in Figure 4.2, during an 18-day period, more than 80% of the files had less than 25% of their data accessed. Under such skewed access patterns, fetching entire files leads to significant read amplification and increased data transfer costs. To address this, Macaron+ caches data at the block level, dividing objects into 1 MB blocks, as in the original Macaron design, rather than storing full files. *Its prefetching algorithm further reduces unnecessary I/O by selectively prefetching blocks that are most likely to be accessed in the near future.*

4.3.1 Access pattern similarity between objects

We observed that, in our target workload, many objects have a set of other objects with highly similar access patterns. To evaluate the significance of this observation, we performed the following analysis. For each object in the trace, we constructed a block-level access set containing all blocks accessed at least once during the entire trace period. We then computed the pairwise Jaccard similarity [232] between these sets, identified the top-N most similar objects for each

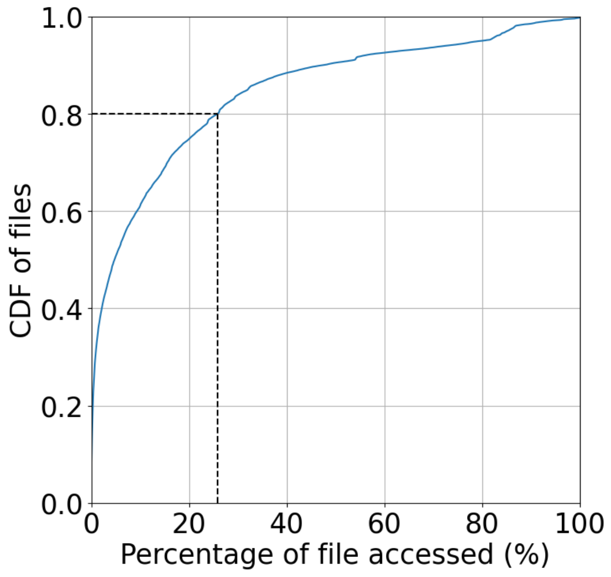


Figure 4.2: Through the Parquet file's predicate pushdown feature, most files accessed in Uber's Presto cluster workload involve only a small portion of their data, rendering file-level caching or prefetching inefficient due to high read amplification.

object, and measured their similarity scores to quantify how closely related these objects are in terms of access patterns. We excluded objects with only a single accessed block, as their similarity comparisons are not meaningful.

Here, given two objects X and Y, the Jaccard similarity is defined as:

$$\text{Jaccard}(X, Y) = \frac{|B_X \cap B_Y|}{|B_X \cup B_Y|}$$

where B_X and B_Y are the sets of blocks accessed in objects X and Y, respectively. A Jaccard similarity of 1 indicates identical access patterns, while values closer to 0 indicate disjoint block access.

Figure 4.3 shows the distribution of Jaccard similarity scores between each object and its top-N most similar objects (N = 1, 5, 10, 50), ranked in descending order of similarity. We find that 87% of objects have at least one other object with an identical access pattern (i.e., Jaccard similarity = 1.0). Moreover, 67%, 54%, and 14% of objects have such exact matches among their top-5, top-10, and top-50 similar objects, respectively.

These results indicate that object access patterns are far from random and instead exhibit strong structural similarity. By identifying objects with similar block-level access behavior, we can use the access history of one object to predict the future access patterns of others in the same similarity cluster. This observation forms the basis for our prefetching approach, which leverages shared behavior across similar objects to improve accuracy and efficiency.

Opportunity 1: *Over half of objects have at least 10 peers with identical block-level access patterns. This observation suggests that objects can be grouped based on their access patterns, and the access histories of objects within the same group can be used to prefetch blocks for one another, enabling prefetching even for previously unseen blocks.*

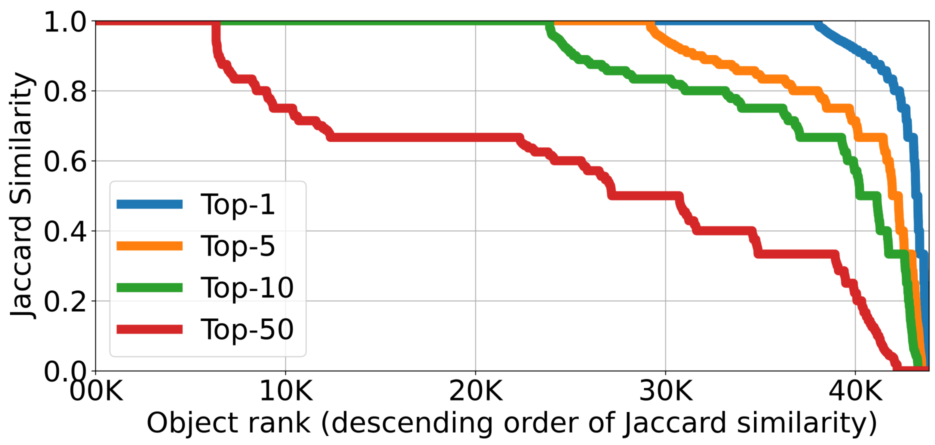


Figure 4.3: Similarity score distribution for the top-N most similar objects (N = 1, 5, 10, 50) shows that more than half of the objects have at least 10 other objects with a Jaccard similarity of 1. This indicates that similar access patterns are common, and many objects share similar block-level access set within the trace.

4.3.2 Future access pattern predictability

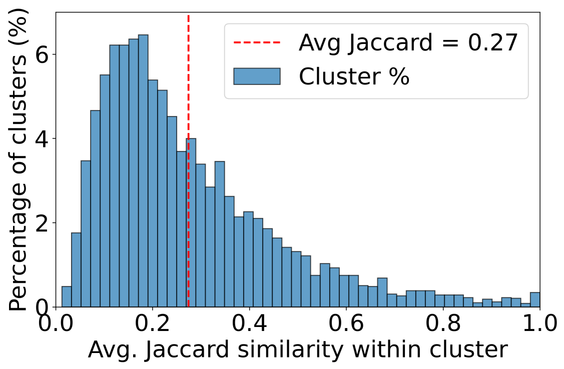
In an ideal oracle prefetcher that has knowledge of all future accesses, the prefetcher can prefetch exactly the blocks that will be used in the near future. In reality, however, online prefetchers see only the history up to the current time, and need to predict future blocks based on the current access information.

To quantify how well early history predicts later behavior, we proceed as follows. Let B_i^{early} and B_i^{late} denote the sets of blocks accessed in an early stage and at the end of the trace for object i, respectively. We consider two definitions of the early-accessed blocks set B_i^{early}:

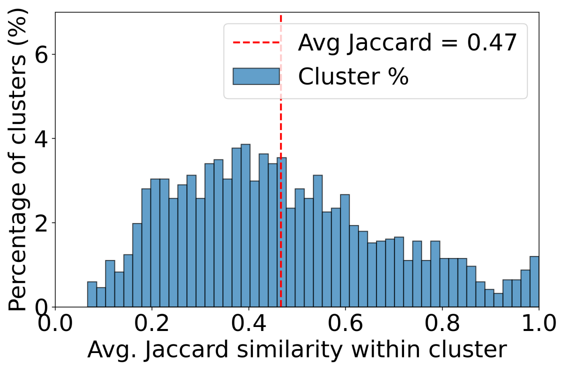
1. All blocks accessed up to and including the second distinct byte-range request.
2. All blocks accessed until the Nth distinct block request (with $N = 5, 10$).

For each early-access definition, we form clusters by grouping any two objects whose early-access sets satisfy $Jaccard\big(B_i^{\text{early}}, B_j^{\text{early}}\big) > 0.9$, i.e., each object in a cluster has at least one peer sharing over 90% of its accessed blocks. Within each resulting cluster, we then compute the average pairwise Jaccard similarity over the corresponding late-access sets B_i^{late}.

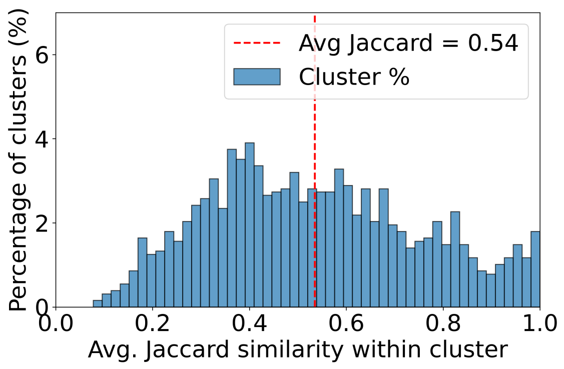
Figure 4.4 plots the distribution of intra-cluster late-access Jaccard similarity for various definitions of B_i^{early}. When B_i^{early} is defined as all blocks accessed up to the second distinct byte-range request (Figure 4.4a), clusters formed with a 90% early-Jaccard threshold exhibit only 27% average overlap in their B_i^{late} sets. This poor predictability is due to the small size of B_i^{early}, on average only 4 blocks, making early clustering unreliable. By imposing a minimum-size constraint and filtering out objects with B_i^{early} size smaller than 5 early blocks (Figure 4.4b), the average intra-cluster late-access similarity rises to 47%, demonstrating that requiring a larger early history substantially improves prediction accuracy. Further increasing the threshold to exclude objects with fewer than 10 early blocks (Figure 4.4c) yields even higher late-access similarity, indicating that richer early histories lead to more reliable clustering and better forecasts of future block accesses.



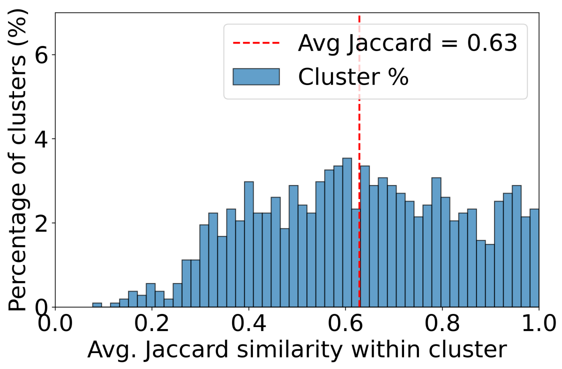
(a) Early-2nd Request



(b) Early-5 Blocks



(c) Early-10 Blocks



(d) Early-10 Blocks+Directory-based

Figure 4.4: Comparison of intra-cluster late-access Jaccard similarity for four early-history clustering criteria: (a) blocks up to the 2nd distinct byte-range request, (b) requiring ≥ 5 early-accessed blocks, (c) requiring ≥ 10 early-accessed blocks, and (d) clustering only objects sharing the same directory path, all using a 90% Jaccard similarity cutoff on B_i^{early}.

Opportunity 2: *Partial early-access history (e.g., the first 5 or 10 blocks) can provide a signal to predict later block accesses, enabling online clustering and prefetching without full knowledge of future requests.*

4.3.3 Correlation between file path and block-level access pattern

There have been many studies [233, 234, 235] exploring metadata-driven file classification and system optimization. In cloud object storage, it is common practice to embed date or time information in object key prefixes (e.g., '/year=2025/month=07/day=21/...') to manage incoming data and support analytics [159, 236, 237, 238]. Such partitioning not only simplifies incremental data ingestion but also enables predicate pushdown on key names, significantly improving object access efficiency for time-based queries.

In our analysis of Uber's Presto workload on Parquet files, we observed that objects under the same parent directory but with different embedded dates (e.g., /a/b/date=2025-07-21/file1 and /a/b/date=2025-07-22/file2) are likely to have similar block-level access patterns. This suggests that directory-path metadata, including time-partitioned prefixes, can effectively group objects with analogous access behavior. To validate this, we repeated our clustering on B_i^{early} with an additional constraint that only objects under the same parent directory (but different dates) to be grouped (see Figure 4.4d). Under this directory-path constraint, the average intra-cluster Jaccard similarity of the corresponding B_i^{late} sets increased by 9% compared to clustering without the constraint. In our current implementation, we rely on previously identified patterns (e.g., /date=MM-DD-YYYY/ or /created_date_MM_DD_YYYY/) to extract parent directory paths. However, we believe this approach can be generalized by leveraging language models to automatically identify and interpret diverse date-related string formats in directory paths.

Opportunity 3: *Directory-path metadata, such as time-partitioned key prefixes, can be used as a hint for grouping objects with similar access behavior, enabling more accurate prefetching.*

4.4 Design of Macaron+ prefetcher

Macaron+ extends the Macaron cache with a cost-efficient prefetcher that reduces data access latency by proactively fetching previously unseen blocks. The prefetcher predicts which objects are likely to exhibit similar block-level access patterns and leverages the access histories of these peer objects to infer the blocks most likely to be needed next (**Opportunity 1**). We begin by describing the overall workflow of the prefetcher (§4.4.1), then explain how and why access patterns are represented as embedding vectors (§4.4.2). We then detail our access pattern prediction algorithm (§4.4.3; **Opportunity 2**), describe how similar objects are identified and how blocks are selected for prefetching (§4.4.4; **Opportunity 3**), and finally explain how the prefetch degree is dynamically adjusted on a per-object basis (§4.4.5).

4.4.1 Workflow

Figure 4.5 illustrates Macaron+'s prefetching workflow. When an object is accessed with a specified byte-range, the following steps are executed: ① The *prefetch process* retrieves the object's *access vector* from the Access vector database, updates it by marking the newly accessed blocks, and saves it back for subsequent accesses. The *access vector* is a binary vector with a predefined maximum length, determined by the maximum supported file size and block size (we used 32 GB as a maximum file size with 1 MB block size results in a vector of length 32,000). Each element is set to 1 if the corresponding block has been accessed, and 0 otherwise. ② The updated access vector is converted into an *embedding vector* by the *Embedding vector predictor*, which captures the predicted future block-level access pattern. ③ Using the embedding vector, an approximate nearest neighbor (ANN) search identifies objects whose predicted access patterns (represented by their own embedding vectors) closely resemble the current object's future access pattern. ④ Once similar objects are identified, the prefetch process analyzes their historical access patterns to determine which blocks are most frequently accessed. These blocks are proactively fetched from the remote data lake and stored in the prefetch cache. ⑤ Upon actual access, the prefetched blocks move from the prefetch cache to the demand cache, which is managed by the original Macaron cache logic.

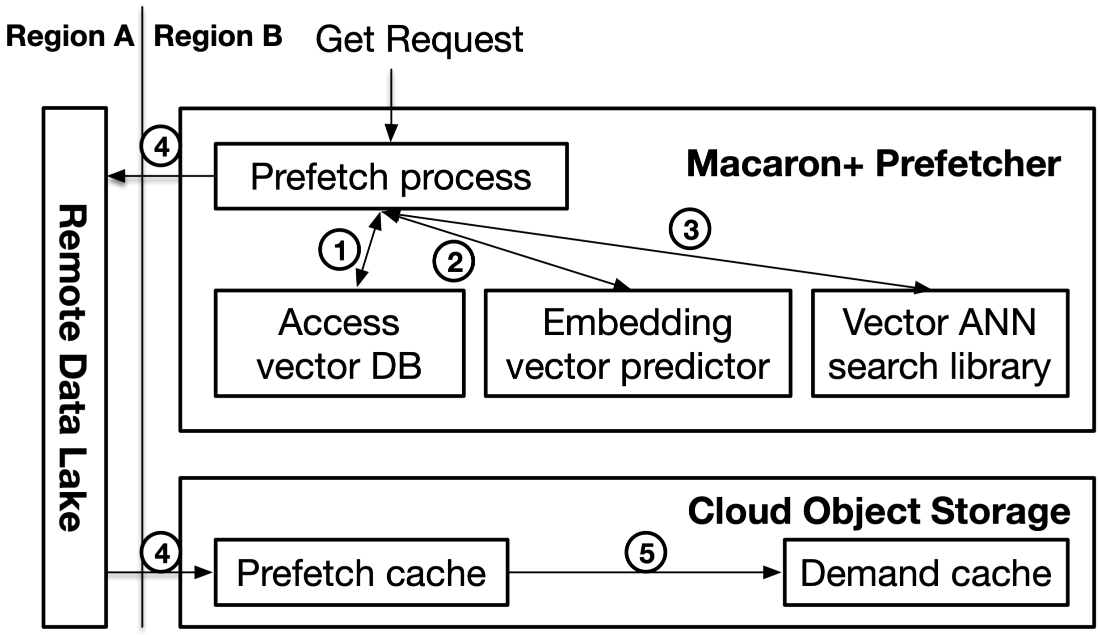


Figure 4.5: Macaron+ overview: Upon receiving a Get request, Macaron+ updates the corresponding access vector, predicts the future access pattern by generating an embedding vector, identifies objects with similar predicted patterns, and prefetches blocks accessed by those similar objects.

4.4.2 Embedding of access vector

To efficiently identify similar objects based on their access patterns in an online manner, Macaron+ compresses the original high-dimensional access vector (of length 32,000 blocks) into a reduced-dimensional embedding vector (of length 1,070). While directly using the full access vector is feasible, it poses several practical limitations. High-dimensional vectors suffer from increased susceptibility to noise and irrelevant bits, reduced robustness in similarity measurements due to the curse of dimensionality [239], and significant computational and storage overhead, especially for approximate nearest neighbor (ANN) search.

Prior studies [218, 220, 231] have proposed embedding techniques to denoise access patterns, improve generalization, and enable learning-based methods. However, most such techniques are designed for address access sequences in systems with a global logical block address (LBA) space, where temporal and spatial correlations can be exploited. In contrast, our setting involves isolated, per-object access vectors, where block indices are local to each object. This makes sequence-based or LBA-delta-based embeddings less effective. Instead, Macaron+ adopts a simpler yet effective approach: embedding each object's binary access vector, where each dimension indicates whether the corresponding block was accessed.

Among dimensionality reduction methods, Principal Component Analysis (PCA) [240] is commonly used. However, PCA is not effective for our online environment due to the computational cost of periodic recomputation as new data arrives. We instead adopt Sparse Random Projection [241], which is highly efficient for streaming data and preserves pairwise distances with high probability.

Formally, given a binary access vector $x \in \mathbb{R}^d$ with $d = 32{,}000$, we compute the embedding

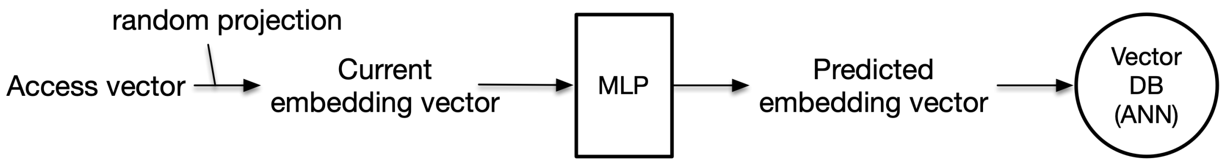


Figure 4.6: Predicting similar access patterns: A pre-trained MLP model predicts an embedding vector representing the future access pattern based on the current access pattern, enabling identification of objects with similar expected behavior.

$y \in \mathbb{R}^k$ with $k = 1{,}070$ using a sparse random matrix $R \in \mathbb{R}^{k \times d}$:

$$y = \frac{1}{\sqrt{k}} Rx, \quad R_{ij} = \begin{cases} +1 & \text{with probability } \frac{1}{2\sqrt{d}}, \\ 0 & \text{with probability } 1 - \frac{1}{\sqrt{d}}, \\ -1 & \text{with probability } \frac{1}{2\sqrt{d}} \end{cases}$$

This construction ensures that R is sparse, reducing both computation and memory usage. According to the Johnson–Lindenstrauss lemma, this projection approximately preserves the pairwise distances between two original vectors x_i and x_j:

$$(1 - \epsilon)\|x_i - x_j\|^2 \leq \|y_i - y_j\|^2 \leq (1 + \epsilon)\|x_i - x_j\|^2$$

with probability at least $1 - \delta$, provided:

$$k \geq \frac{4 \ln n}{\frac{\epsilon^2}{2} - \frac{\epsilon^3}{3}}$$

where n is the number of objects. We empirically set $k = 1{,}070$, which allows up to 30% distortion in pairwise distances, though the actual distortion is often significantly lower in practice.

4.4.3 Prediction of future block-level access pattern

Macaron+ identifies objects with similar access patterns. A naive approach is to use the embedding vector computed from the current access vector and compare it against those of other objects in the embedding pool. However, we observed that this approach often results in low prefetching accuracy, especially when the target object has only been partially accessed and thus contains only a small number of observed blocks. In such early stages, the access vector lacks sufficient information, leading to unreliable similarity comparisons and incorrect prefetch decisions.

To address this, we leverage the insight that final access patterns exhibit a meaningful correlation with earlier patterns (§4.3.2). Based on this observation, Macaron+ trains a lightweight predictor that estimates an object's future access pattern embedding from its current access vector. This prediction improves the reliability of similarity comparisons and consequently the quality of prefetch decisions.

As shown in Figure 4.6, we use a simple multilayer perceptron (MLP) model to predict future access embedding. The input is the current access embedding vector and the output is the

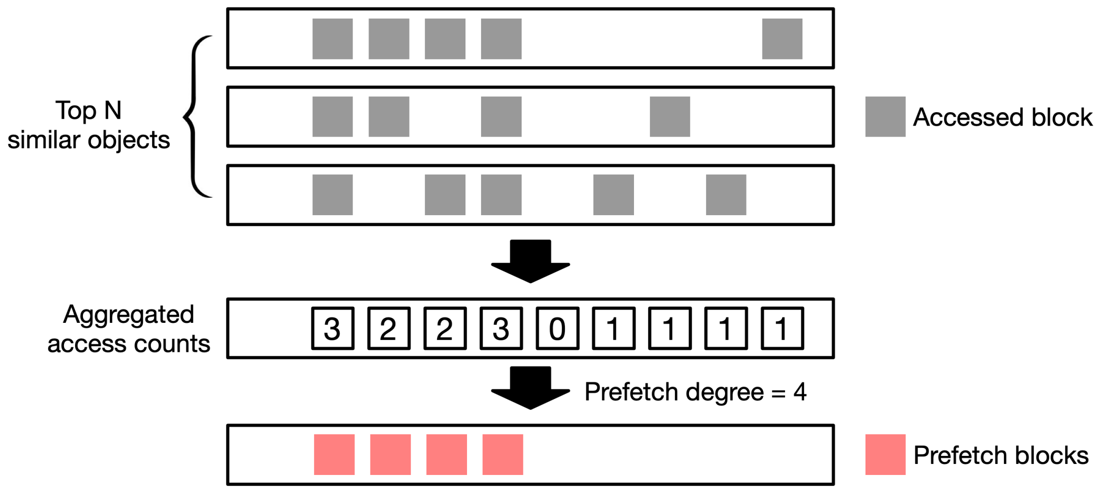


Figure 4.7: Block selection strategy in Macaron+: After identifying objects with similar access patterns, Macaron+ aggregates their access histories, ranks blocks by access frequency, and incrementally selects blocks to prefetch in descending order of popularity until the prefetch degree is satisfied.

predicted final embedding vector. The training dataset consists of per-object embedding vectors recorded during accesses: each training pair (x, y) consists of an intermediate embedding vector x and the final embedding vector y (i.e., generated from the object's last access). To avoid overfitting, we use the data from the last day as validation data and terminate training when the validation error plateaus.

The MLP model consists of three hidden layers with dimensions 2048, 1024, and 2048, respectively. Each layer uses ReLU activations, batch normalization, and dropout (rate = 0.3) to avoid overfitting. The model is trained using a custom loss function that combines Mean Squared Error (MSE) and cosine similarity loss:

$$\mathcal{L}(x, y) = \alpha \cdot \text{MSE}(x, y) + (1 - \alpha) \cdot (1 - \cos(x, y))$$

where $\cos(x, y)$ is the cosine similarity between the predicted and target vectors, and $\alpha \in [0, 1]$ balances the magnitude accuracy and the directional similarity. We set $\alpha = 0.5$ to give equal weight to both components.

4.4.4 Similarity search and block selection strategy

For each access to the object, the *embedding vector predictor* generates an embedding vector representing the future access pattern predicted by the object. While similar objects could be identified by computing pairwise distances against all previously accessed objects, this approach is computationally expensive and impractical for online use. Instead, we employ Voyager [242], an efficient approximate nearest neighbor (ANN) search library developed by Spotify that implements the HNSW algorithm [243], to retrieve a small set of similar objects. Our evaluation shows that using approximate neighbors identified by Voyager yields cost-efficiency in latency reduction nearly identical to that of an approach using exact nearest neighbors. As discussed in Section 4.3.3, Macaron+ can optionally incorporate directory-path metadata, such as time-partitioned key prefixes, as a hint to constrain similar object search, and we evaluate the effectiveness of this hint-based approach in Section 4.5.3.

Once similar objects are identified, Macaron+ must determine which blocks to prefetch based on their access histories. We considered two strategies. The first approach incrementally selects the most recently accessed blocks from the top-ranked similar objects, one by one, until the desired prefetch degree is reached. The second approach, illustrated in Figure 4.7, aggregates the access vectors of the top-N most similar objects and selects blocks in descending order of their aggregate access frequency. We empirically found that the latter method yields a higher prefetch accuracy and thus we adopt it as the default strategy in Macaron+.

4.4.5 Dynamic prefetch degree

Several prior works on sequential prefetching [215, 216, 244, 245, 246] have proposed dynamically adjusting the degree of prefetch based on whether the observed access pattern appears sequential or random. Inspired by these techniques and because cloud object storage lacks a global address space, Macaron+ dynamically adjusts the prefetch degree per object based on the effectiveness of past prefetches. It tracks each object's prefetch hit ratio, defined as the fraction of prefetched blocks that were subsequently accessed, and adjusts the degree accordingly. If the ratio is too low (below 20%), the degree is reduced to one-fourth. If the ratio is high (above 50%), it is doubled to exploit predictable access patterns. This lightweight feedback-driven mechanism allows Macaron+ to apply aggressive prefetching only when it is likely to be effective.

4.5 Evaluation

We evaluated Macaron+ to assess the following aspects: its ability to reduce latency cost-effectively compared to baselines (§4.5.2), comparisons with Macaron+ variants under different constraints (§4.5.3), effectiveness against an online approach without prediction (§4.5.4), and sensitivity to various configuration parameters (§4.5.5).

4.5.1 Experimental setup

Traces. We evaluate Macaron+ using a production trace from Uber's Presto cluster, which captures object accesses generated by interactive and batch analytics queries on Parquet files. We used the first 10 days of the trace as a warm-up and to train the prediction model used by Macaron+ 's prefetcher, and evaluated the rest of the 8 days of the trace, which in total 18 days of the trace. All other experimental settings follow those described in Section 3.6.1.

Configurations. By default, we deploy the Macaron object storage cache (OSC) only, omitting the DRAM cache layer to focus on evaluating the effectiveness of the prefetcher. However, users seeking even lower latency can combine our prefetcher with Macaron's DRAM cache layer, which can be automatically configured. The workload is assumed to run on AWS N. Virginia region, while the remote data lake is in N. California, incurring cross-region egress charges. We adopt the AWS pricing model for all cost measurements.

The prefetch degree, which is the number of 1 MB blocks fetched per prefetching action, is evaluated with seven configurations: 2, 4, 8, 16, 32, 64, and 128. While the demand cache capacity is automatically configured by Macaron, we fix the prefetch cache capacity at 2 TB. Our sensitivity analysis confirmed that this size is sufficient for storing useful prefetched items and that the associated capacity cost was negligible compared to the egress and operation costs. Automatically tuning the prefetch cache capacity is a promising direction for future work. Finally, we select the top 5 similar objects for block selection, as described in Section 4.4.4.

Baselines and costs. We compare Macaron+ with existing prefetching techniques:

- **LDC** [247] (History-based prefetching): Implements the Local Delta Correlation (LDC) prefetcher, which tracks the deltas (differences in block addresses) between consecutive accesses within a stream. It predicts future accesses by matching recent delta sequences with patterns stored in a correlation table.
- **Sequential** [216, 248, 249, 250] and **Linux** [244] (Sequential prefetching): Includes a basic sequential prefetcher that fetches a fixed number of blocks (equal to the prefetch degree) following the most recently accessed block within an object. Also includes the Linux VFS read-ahead strategy, which dynamically adjusts the prefetch degree based on observed access contiguity.
- **LSTM** [231] (ML-driven prefetching): Uses the same delta-based representation as the Local Delta Correlation prefetcher but replaces the table lookup with an LSTM model trained to predict the next delta. This allows learning more complex sequential patterns in block accesses.

We additionally implemented and evaluated **Mithril** [213] and **Tombolo** [214], which use graph- or table-based representations to infer relationships between previously accessed blocks for prefetching. However, we exclude them from the evaluation results as they fail to prefetch unseen blocks from the remote data lake, resulting in no latency reduction.

For comparison with broader configurations without prefetching, we include the following baselines as defined in Section 3.6.1:

- **Remote**: All data accesses are served directly from the remote data lake, incurring full egress costs and high data access latency.
- **Replicated**: All data is proactively replicated to the local region, avoiding egress charges for data accesses, but incurring high synchronization and storage costs.
- **Macaron**: The original system with the object storage cache, without prefetching.
- **Oracular**: An offline prefetching policy that has access to the complete trace of future accesses. When an object is first referenced during trace replay, the prefetcher immediately issues prefetches for every block that will be accessed for that object later in the trace and places those blocks into the prefetch cache. Because cloud object storage uses a per-object address space and the prefetcher makes no cross-object inferences, prefetching is triggered only on the first access to each object. Blocks accessed at that initial access are cold misses, while subsequent accesses to the prefetched blocks are cache hits. Since every block prefetched by this baseline is known a priori to be accessed, we treat this baseline as a best-case upper bound and do not limit the prefetch cache capacity, avoiding evictions of

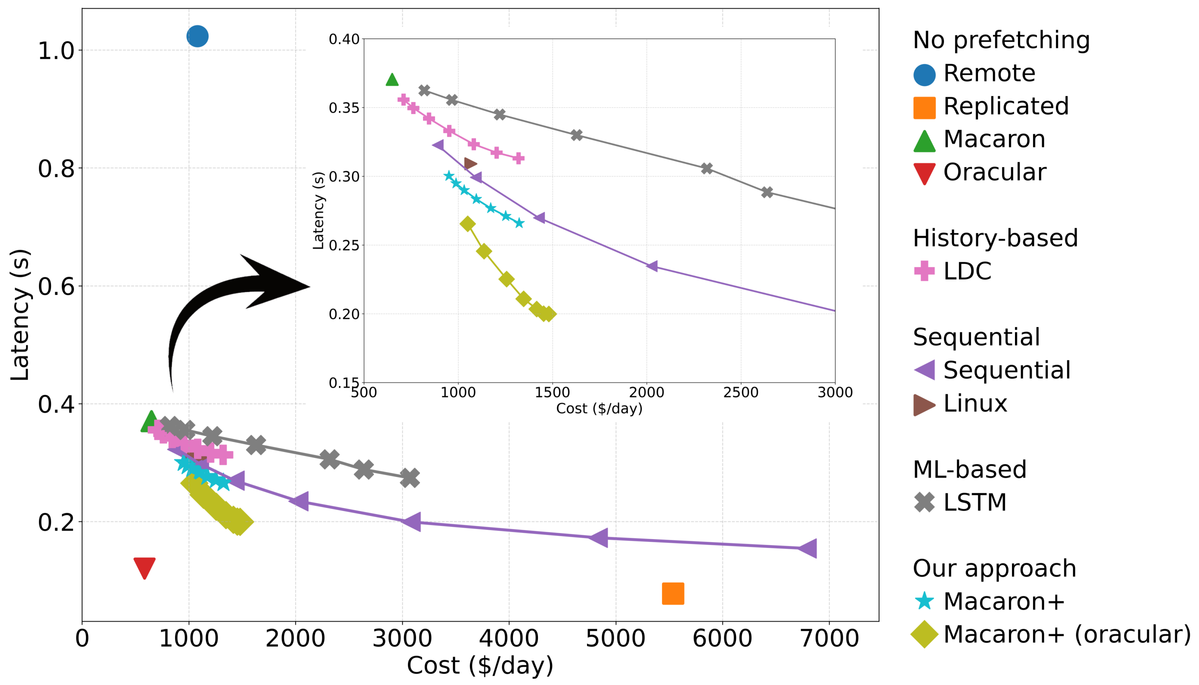


Figure 4.8: Cost–latency trade-off of various prefetching approaches. Macaron+ achieves the highest cost-efficiency among online prefetching approaches and demonstrates potential for further improvement, approaching the performance of Macaron+ oracular. Sequential prefetching, while simple and initially competitive, becomes increasingly cost-inefficient at higher prefetch degrees due to naive unnecessary data transfers.

blocks that will be needed later.

Finally, we include two versions of Macaron+:

- **Macaron+**: Our proposed method. Trained on the first 10 days of access history, it uses an MLP-based prediction model to estimate future access patterns and performs similarity-based prefetching on the remaining 8 days.
- **Macaron+ (oracular)**: Given the complete trace, we compute for every object the final set of accessed block offsets, derive pairwise similarity scores between all object pairs from those final offsets (i.e., using oracle knowledge), and use the resulting precomputed similarities to drive Macaron+ 's prefetcher. Although this policy is less performant than the true **Oracular** baseline described above, it provides a near-optimal upper limit for Macaron+ 's similarity-based prefetching under the idealized assumption of perfect knowledge about which objects will be truly similar over the trace.

The cost model used for each baseline follows the definitions in Section 3.6.1, with additional components to account for prefetching: specifically, we include data egress costs incurred by prefetched blocks and capacity costs associated with maintaining the prefetch cache.

4.5.2 Trade-off between cost and latency reduction

Observation 1: Macaron+ reduces average data access latency by 28% at the cost of 103% increase in total data access expenses, compared to the Macaron cache without prefetching.

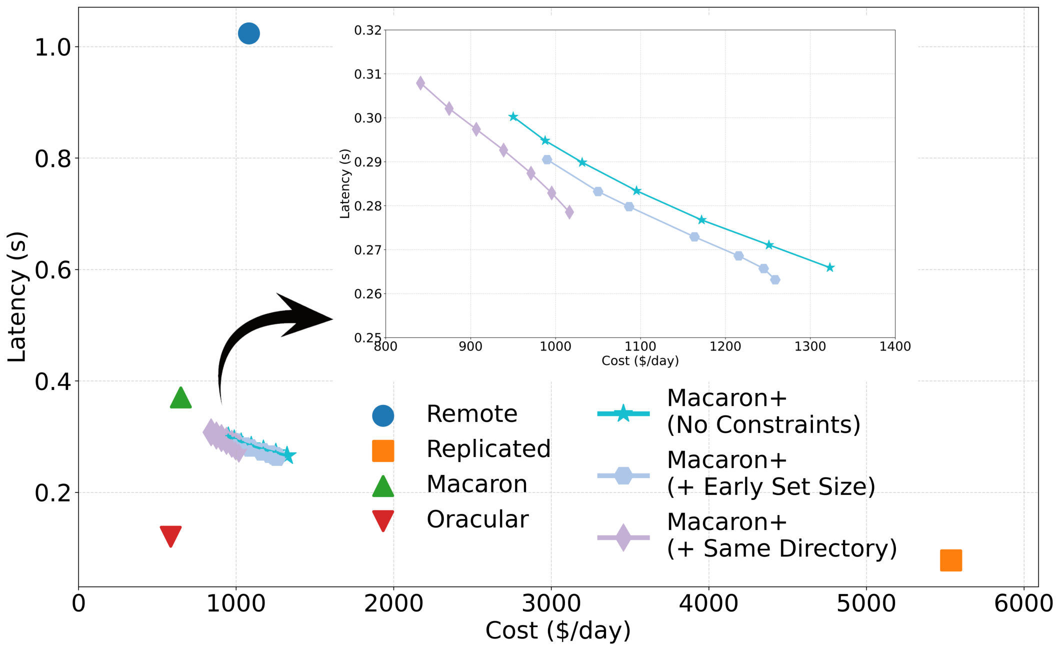
Macaron+ aims to reduce average data access latency with minimal increases in remote data access costs. To evaluate the cost-efficiency of prefetching, Figure 4.8 presents the latency-cost trade-off across different prefetching strategies. The results show the average data access latency and daily remote data access cost measured over the final 8 days of the Uber trace, following a 10-day observation period.

Overall results. Among the online prefetching approaches, Macaron+ achieves the highest cost-efficiency, reducing the average latency of data access by 19–28% at a cost increase of 46–103% compared to Macaron cache without prefetching, depending on the prefetch degree. For the same average latency, Macaron+ reduces the costs by 13–61% (avg. 41%) compared to other baselines, and at the same cost, reduces latencies by 5–22% (average 12%). In particular, compared to sequential prefetching, the most cost-efficient baseline, Macaron+ achieves the same latency with 14% lower cost.

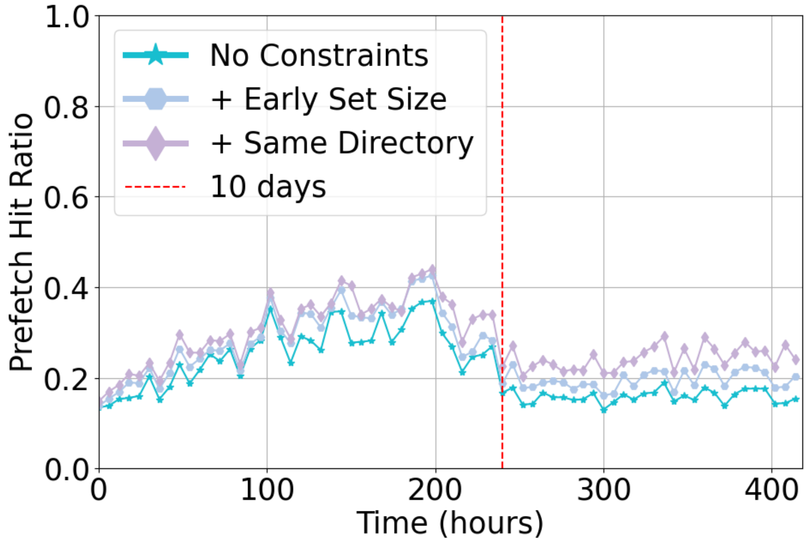
Comparison with Macaron+ (oracular). The key advantage of Macaron+ (oracular) is that it can accurately identify objects with similar future access patterns by observing their complete access histories over the trace. This enables Macaron+ (oracular) to infer the precomputed object similarity during trace replay and to prefetch the precise blocks that similar objects will access, resulting in near-optimal performance for similarity-based prefetching. In contrast, Macaron+ is based on a trained MLP model to predict future access patterns using only partial access histories. Due to this limitation, Macaron+ (oracular) achieves 8–21% lower latency than Macaron+ at the same cost. However, this performance gap highlights the opportunity to improve Macaron+ by enhancing its prediction accuracy.

Comparison with sequential prefetching. As shown in Figure 4.8, sequential prefetching effectively reduces latency by aggressively fetching consecutive blocks, achieving the lowest average access latency - at the cost of significantly higher spending, even exceeding that of the replicated approach. While its cost-efficiency degrades rapidly with larger prefetch degrees, it performs better than other baselines at lower prefetch degrees and is even comparable to Macaron+ in those regimes. The Linux prefetching is on the sequential prefetching curve, as it builds on sequential prefetching with a dynamic adjustment of the prefetch degree based on detected access sequentiality.

Block address delta based approaches. Both GHB and LSTM fail to reduce latency cost-efficiently. These methods rely on block address delta sequences within a global block address space, but cloud object storage assigns each object an independent address space, making inter-object deltas undefined and delta patterns sparse. As a result, GHB lacks sufficient information to trigger prefetching and LSTM struggles to learn effectively. We also tested assigning global offsets to simulate a unified space, but this further degraded performance as this artificial alignment was semantically invalid.



(a) Cost-latency trade-off with added constraints.



(b) Prefetch hit ratio over time.

Figure 4.9: Impact of incrementally adding constraints on Macaron+'s prefetching performance. (a) Limiting prefetching to objects with at least five accessed blocks and restricting similarity search to the same directory improves prediction accuracy and cost-latency trade-off. (b) Prefetch hit ratio increases from 16% to 24% over time by applying these constraints.

4.5.3 Ablation study and Macaron+ variants

Observation 2: Macaron+ improves prefetch accuracy from 16% to 24% by incorporating more early access information and file path metadata, at the cost of reduced prefetch aggressiveness and limited latency improvement.

As described in Section 4.3, objects with at least five accessed blocks tend to yield more accurate predictions, as the richer early access information better reflects their eventual access patterns. Moreover, restricting similarity search to objects sharing the same directory path is expected to further improve prediction precision. We evaluate the impact of these two constraints in this section.

Figure 4.9a presents how the cost–latency trade-off of Macaron+ changes when each constraint is added incrementally. When limiting prefetching to objects with at least five accessed blocks, the prediction becomes more accurate due to the increased input information, resulting in a better cost-efficient trade-off. Interestingly, although fewer objects participate in prefetching, those that have more accessed blocks, leading to a higher number of prefetched blocks for the same prefetch degree. In contrast, when similarity is computed only among objects within the same directory path, the number of candidate objects decreases significantly, reducing the number of blocks prefetched. However, the improved precision in identifying truly similar objects leads to better cost–latency trade-off overall.

Figure 4.9b shows the prefetch hit ratio over time as the trace is replayed, using a model trained on the first 10 days of data. During this initial period, since the model is evaluated on data it was trained on, prediction accuracy remains high, resulting in a high prefetch hit ratio. As noted in Section 4.4.3, the first 9 days are used for training and the 10th day for validation, which explains the slight drop in hit ratio on day 10. After day 10, when the model is applied to unseen data, we observe that introducing constraints progressively improves prediction quality, increasing the average prefetch hit ratio from 16% to 24% over the remaining 8 days, a 50% relative improvement.

4.5.4 Effectiveness of access pattern prediction

Observation 3: Without future access prediction, similarity-based prefetching is less cost-efficient than sequential prefetching.

In this section, we evaluate why relying solely on an object's current access pattern is insufficient for similarity-based prefetching, and we demonstrate the necessity of predicting future access behavior. As described in Section 4.4.3, Macaron+ first predicts a future access embedding for each object and then uses that embedding to identify similar objects. A simpler alternative is to skip prediction and instead use an object's current access pattern to find similar objects.

Figure 4.10 shows that this simpler strategy performs poorly: When Macaron+ uses only current access patterns (no prediction), its cost-efficiency for latency reduction is worse than that of the basic sequential prefetching strategy. The root cause is that, during the early stages of object accesses, available block-level information is sparse and yields inaccurate similarity estimates. Consequently, objects that are not truly similar in their eventual access behavior are often misidentified as similar, producing ineffective prefetches and incurring unnecessary cost.

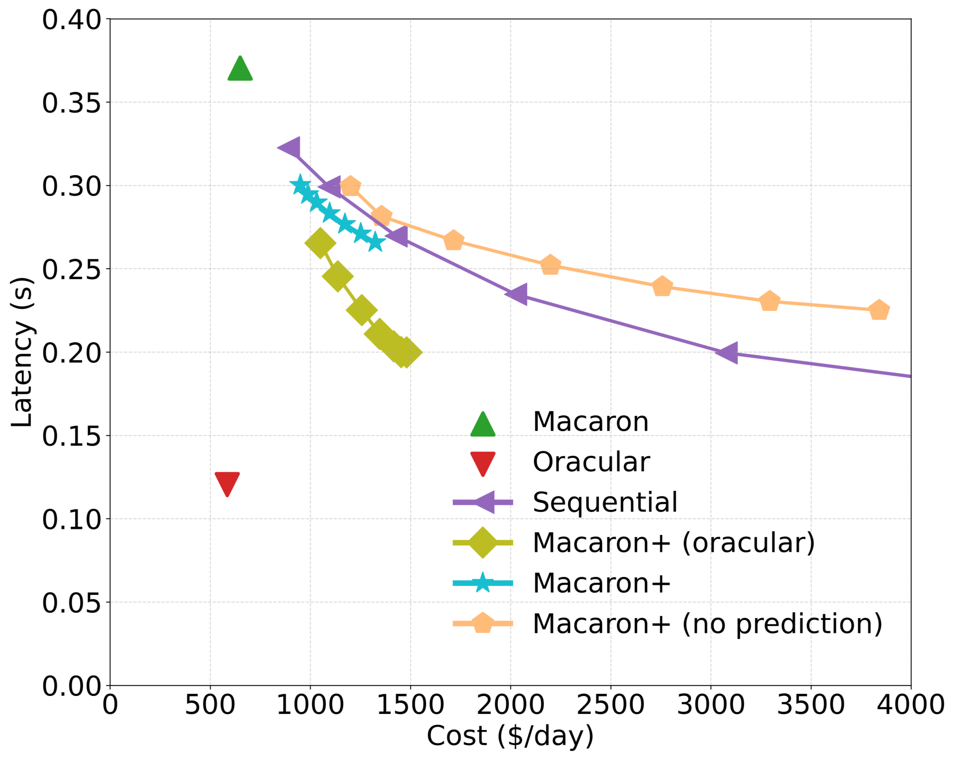


Figure 4.10: Without predicting future access patterns, Macaron+ performs worse than simple sequential prefetching due to inaccurate similarity estimation from limited early access data.

In contrast, Macaron+ (oracular) derives object similarities from the complete trace of access events; these oracular similarities correctly identify objects that will exhibit similar access behavior in the future, substantially improving both prefetching accuracy and overall cost-efficiency.

To quantify the prediction quality, Figure 4.11 plots the cosine similarity between the predicted final embedding of each object and its true final embedding. We use cosine similarity because Macaron+'s similarity search (ANN) operates on embeddings using cosine similarity. Our MLP-based access pattern predictor achieves high cosine similarity on training data: for example, among the data used for training (i.e, the accesses from the first nine days used for training the MLP model), about 80% of the predictions achieve cosine similarity above 0.8. However, the distribution of cosine similarities on the evaluation data (i.e, accesses from the last eight days used for evaluation) is worse, indicating that the predicted final embeddings generalize less well to unseen data and that future embedding prediction is therefore less accurate; despite this, Macaron+ with prediction still yields more cost-efficient prefetching than other baselines and Macaron+ without prediction.

This gap in prediction accuracy explains why Macaron+ performs less well relative to Macaron+ (oracular). Importantly, the results also point to a clear opportunity: improving the prediction module, for example through more expressive models, larger training sets, or advanced training procedures, could narrow the gap to the oracular upper bound. In other words, advances in modeling and training have the potential to bring similarity-based prefetching much closer to the cost-efficiency exhibited by Macaron+ (oracular). Section 5.2.3 discusses potential directions

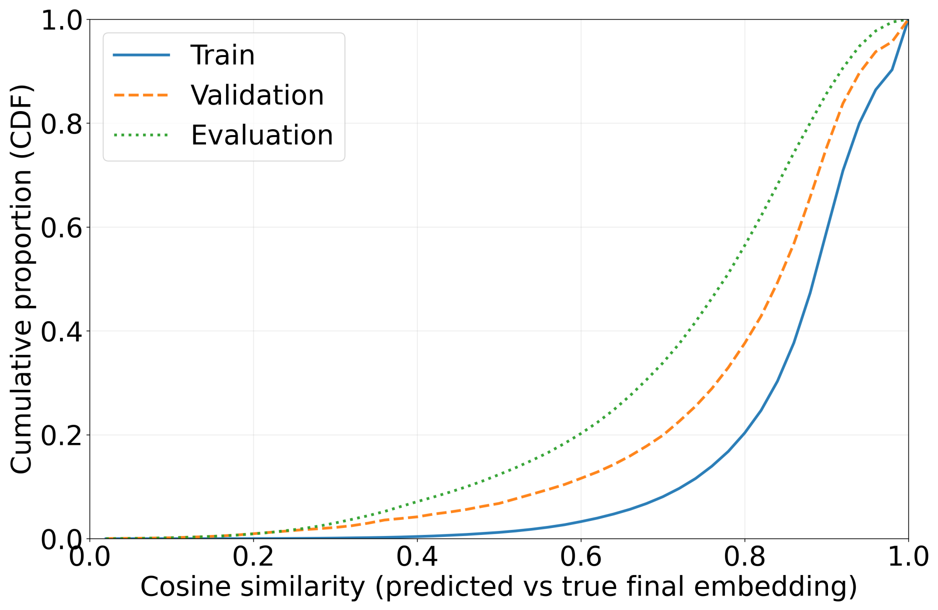


Figure 4.11: Cosine-similarity distributions (predicted vs. true final embeddings) for the train, validation, and evaluation splits. Despite the lower prediction accuracy observed on the evaluation split, which explains the cost-efficiency gap to Macaron+ (oracular), Macaron+ with prediction remains more cost-efficient than other baselines.

for understanding and reducing the performance gap between Macaron+ and Macaron+ (oracular), including investigating why prediction quality degrades on the evaluation data relative to the validation data in Figure 4.11.

4.5.5 Sensitivity analysis

Observation 4: Using too many similar objects degrades cost-efficiency due to reduced similarity precision, while selecting too few limits latency reduction due to insufficient prefetch candidates.

We analyze how the number of similar objects used for block selection affects Macaron+'s prefetch accuracy and latency. As described in Section 4.4.4, Macaron+ selects blocks from the top-N most similar objects per access. Figure 4.12 shows the cost-latency trade-off when using top 3, 5, and 10 similar objects.

Using more similar objects (e.g., top 10) increases the number of candidate blocks, enabling lower latency (as low as 182 ms on average), but includes less similar objects, reducing prefetch accuracy and cost-efficiency. Conversely, using fewer objects (e.g., top 3) improves similarity precision and achieves 15%-38% (avg. 30%) higher cost-efficiency than top 10 at the same latency. However, due to the limited block pool, it can only reduce latency to 200 ms at best, which is 10% higher than with top 10.

This trade-off suggests that choosing the number of similar objects involves balancing prefetch precision and aggressiveness, depending on system goals, and Macaron+ uses $N = 5$ as a default

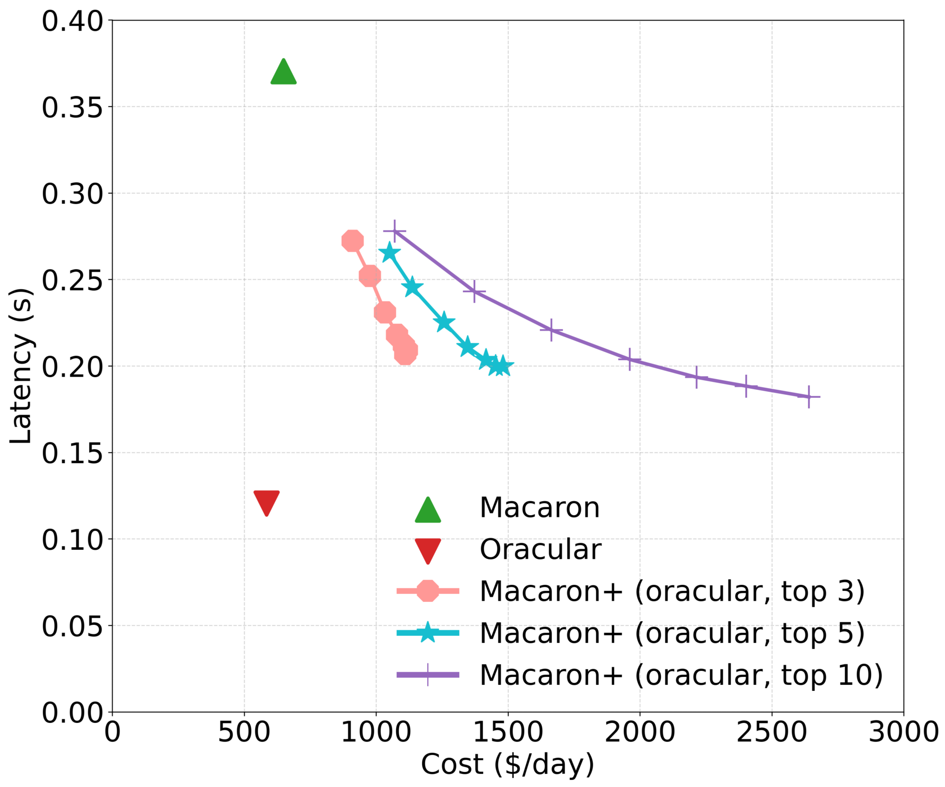


Figure 4.12: Cost-latency trade-off when using top 3, 5, or 10 similar objects. Fewer objects improve cost-efficiency; more enable lower latency.

policy.

4.6 Summary

Macaron+ reduces latency in the access of cross-region data by prefetching blocks based on future predicted access patterns and object-level similarity, rather than relying on the global address space or eviction history. Designed to complement Macaron's elastic caching, Macaron+ addresses workloads with high compulsory miss ratios by prefetching data before any access occurs. It achieves a 13–61% cost reduction compared to other prefetching baselines for the same latency target. However, when provided with complete knowledge of which objects will be eventually similar at the end of the trace, a version of Macaron+ that uses this oracular information achieves better cost-efficiency. This gap highlights a promising direction for future improvement, where more accurate access pattern prediction, such as leveraging advanced AI models, could bring Macaron+ closer to the performance of the oracle case.

Chapter 5

Conclusion and future directions

We conclude this dissertation by summarizing our research contributions that support the thesis statement and by outlining potential future directions for further exploration in this area.

5.1 Conclusion

This dissertation explores how to reduce storage-related costs in public cloud environments while maintaining the performance required by modern applications. Based on the thesis that **leveraging the elasticity and diversity of public cloud resources, combined with real-time workload monitoring, can effectively reduce both storage deployment costs and cross-region/cloud data access costs**, this thesis presents three systems: Mimir, Macaron, and Macaron+ that embody key cloud-aware design principles.

First, Mimir demonstrates that provisioning cost-efficient virtual storage clusters requires fine-grained awareness of workload characteristics and the heterogeneous performance-cost tradeoffs of cloud storage options. By profiling workloads and benchmarking storage types, Mimir selects optimal storage configurations, reducing deployment costs by up to 81% compared to prior techniques.

Second, Macaron introduces an auto-configuring cache that mitigates high egress costs and latency in cross-region/cloud access scenarios. By dynamically adjusting cache capacity and monitoring using miniature simulations, Macaron reduces access costs by up to 66% compared to managed cloud caching services and approaches the performance of an ideal offline oracle cache with only 9% higher cost.

Third, Macaron+ builds upon Macaron by tackling workloads with high compulsory miss ratios that are inherently uncacheable at the first access. It introduces a cost-aware prefetching mechanism based on access pattern similarity and prediction, avoiding the limitations of traditional address- or eviction-based prefetchers. Evaluations on real-world traces show that Macaron+ can reduce latency by up to 61% lower cost than existing prefetching approaches for the same latency target.

Together, these systems validate our thesis by showing that cost-efficiency in native cloud storage systems can be significantly improved by exploiting heterogeneity, elasticity, and workload awareness.

5.2 Future directions

We now discuss potential future directions that build on the systems and findings presented in this dissertation.

5.2.1 Mimir

Incorporating data reliability and latency to SLOs. While Mimir currently focuses on identifying cost-optimized storage cluster configurations that meet user-defined throughput requirements, an important future direction is to extend its optimization model to incorporate additional service level objectives (SLOs), such as data reliability, availability, and latency. For example, although local SSDs offer a cost-effective way to achieve high throughput, they are ephemeral and tied to the lifecycle of the hosting VM, making them less suitable for workloads requiring persistent data. In contrast, network-attached volumes like AWS EBS are more durable since they are decoupled from VM failures, but they typically require over-provisioning to meet the same throughput targets, leading to higher costs. A promising direction is to quantify and incorporate the reliability and availability trade-offs of different storage types, for example, modeling the reliability of three-way replicated local SSDs to match that of a single EBS volume, and reflect them in Mimir's configuration search. Additionally, current formulations do not explicitly incorporate latency constraints. Extending Mimir to filter or penalize configurations that cannot meet user-specified latency SLOs would enable broader applicability to latency-sensitive workloads and promote more practical deployment decisions in real-world settings.

Throughput prediction without benchmarking. Mimir currently relies on empirical benchmarking to estimate the achievable throughput of each virtual machine and storage combination for a given workload. While accurate, this process is both time and cost intensive, especially when evaluating a large number of configurations. In contrast, prior work [16, 25, 46] in compute-focused VM selection has shown promise in predicting performance using system specifications. A promising future direction is to extend Mimir with a performance prediction model that takes VM and storage specifications along with workload characteristics as input and outputs the expected throughput. We have explored early prototypes using gradient boosting trees [52] for this purpose, but found that prediction errors on unseen configurations limited practical adoption. Improving the accuracy and generalizability of such models remains an open and compelling direction for future work.

5.2.2 Macaron

Expanding to multi-tier storage hierarchies. Macaron currently targets a two-tier cache model comprising DRAM and general-purpose object storage such as AWS S3 Standard or Azure Hot Blob. However, modern cloud providers offer a broader spectrum of storage options with varying cost-performance trade-offs, including high-performance tiers such as AWS S3 Express One Zone and Azure Premium Block Blob, and low-cost infrequent access tiers like AWS S3 Infrequent Access and Azure Cool Blob. Incorporating these additional levels into Macaron's cache design can enable more flexible caching strategies that better align with workload latency and cost requirements. For example, performance-sensitive data could be placed in fast tiers to reduce latency, while rarely accessed blocks could be stored in cheaper archival tiers to reduce cost. Designing cache tiering policies and reconfiguration mechanisms that make effective use of these multi-tier storage hierarchies presents a promising future direction.

Extending to active-active consistency models. Macaron is currently designed for read-heavy workloads, with the assumption that the remote data lake remains unchanged during cache usage. However, as demand increases for active-active architectures that allow updates and deletions from multiple regions, extending Macaron to support such scenarios becomes an important direction. One potential approach is to use services like AWS S3 Object Lambda, which can invoke callback functions when remote objects are modified or deleted. This would allow the cache to update or invalidate affected entries in response. However, invoking such functions on every access can lead to increased costs proportional to access frequency, and the additional management overhead may impact performance. Investigating these trade-offs and developing cost-efficient and scalable update propagation mechanisms remains an open challenge for future research.

5.2.3 Macaron+

Cross-object correlation beyond access patterns. Macaron+ currently identifies prefetch candidates based solely on intra-object access patterns, using block-level similarity to decide which parts of the current object should be prefetched. However, as shown in Section 4.5.3, adding constraints such as minimum early access set size or path similarity improves accuracy and cost-efficiency but reduces the number of prefetched blocks, limiting the upper bound of latency reduction. A promising future direction is to incorporate cross-object correlation, where the system learns that certain objects are often accessed together in recurring query patterns or due to semantic relationships such as shared table partitions. By prefetching relevant blocks not only from the current object but also from other correlated objects, the system can improve both the accuracy and coverage of prefetching. This approach has the potential to further reduce latency without sacrificing cost-efficiency. Future work can explore how to identify and leverage these relationships using features like co-access frequency, query lineage, or metadata hierarchy.

Dynamic prefetch cache size and prefetch degree. While Macaron dynamically reconfigures its demand cache size based on workload characteristics and cost-efficiency goals, Macaron+ currently uses a fixed prefetch cache size and prefetch degree set by the user at deployment time. However, our evaluation shows a clear trade-off between prefetch degree and cost-efficiency: higher degrees reduce latency by prefetching more blocks, but often include less relevant data, which increases cost. A promising future direction is to design a system that takes a user-defined cost–latency balance as input and automatically determines appropriate values for both the prefetch cache size and prefetch degree. We conducted preliminary experiments using the miniature simulation approach, where a miniature cache is used to estimate performance in different configurations. This method showed high accuracy for simple prefetching strategies such as sequential prefetching. Extending this idea to support more complex similarity-based algorithms like Macaron+ and evaluating whether miniature simulations can still guide dynamic tuning effectively is a compelling area for future exploration.

Developing similarity-based prefetching. Unlike traditional time-series prefetchers, Macaron+ uses a similarity-based strategy that relies on object-level access patterns. As shown in Section 4.5.2, when supplied with oracular knowledge of future accesses, the system can identify truly similar objects and achieve substantially higher cost-efficiency. This suggests that improvements in future access prediction will directly increase Macaron+'s ability to find appropriate neighbors and thus improve the cost-effectiveness of similarity-based prefetching.

Two promising complementary directions follow. First, incorporate richer file-level signals into similarity and prediction models. In addition to path information, which our evaluation already shows to be a strong indicator, metadata such as creation time, file size, file format, and ownership can provide informative priors about access behavior. Developing this idea requires an initial observational step to identify which candidate features actually help distinguish similar objects in practice, followed by a careful design phase to integrate the most useful features into the similarity metric and prediction pipeline.

Second, explore more expressive model architectures and robust training methodologies to predict future access embeddings. Whereas our current MLP provides a light-weight and fast solution, attention-based architectures (e.g., Transformers) or sequence models may capture richer temporal and contextual patterns, and thus yield more accurate embeddings. Because larger models introduce runtime and deployment trade-offs for online inference, model compression techniques, such as quantization, pruning, knowledge distillation, and 1-bit/low-precision approaches, are important complementary tools to retain inference efficiency while improving representational power.

Together, richer feature sets and more powerful yet practically deployable models are likely to raise prefetch hit ratios and coverage. Evaluating these directions while accounting for training/inference costs and online deployment constraints is a key next step toward closing the gap with the oracular upper bound.

Exploring generality across workloads and file formats. Exploring the generality of Macaron+ across workloads and file formats is an important open question. The Uber Presto trace used in this study is a real production trace and is representative of many analytical workloads, but its access characteristics are shaped by query engine optimizations and Parquet file structure. In particular, production engines, including Uber's, often apply aggressive predicate pushdown and selective row-group reads, causing queries to touch only small, highly selective portions of files. Under these conditions, simple sequential prefetchers may appear effective, as we have shown in our evaluation, and similarity-based methods may show different relative benefits than they would on workloads with different access patterns. Demonstrating broader generality, therefore, requires careful, controlled evaluation on workloads and query-engine configurations that reflect production behavior.

Two complementary directions are especially important to investigate. First, additional distributed analytics traces should be evaluated to determine whether the access patterns we observed in the Uber trace occur elsewhere. Doing so can confirm the data-access pattern observations from the Uber trace and help debug weaknesses in the access-pattern prediction module, which in turn can guide improvements to prefetching. The primary challenge is obtaining representative workloads: public benchmarks often do not reproduce production-level optimizations (e.g., aggressive predicate pushdown) by default, so careful selection and configuration of traces and query engines are required to analyze and evaluate traces precisely.

Second, it is important to evaluate Macaron+ on a wider variety of workload classes beyond analytic queries, for example, data accesses from large-scale machine learning workloads or collaboration scenarios in which teams access data remotely across regions. Two concrete questions arise here: does similarity-based prefetching improve performance for workloads with different access characteristics, and how well does the Macaron+ prediction model generalize across workload types? One intriguing direction is to build a shared foundational model that can provide good predictions across many workloads with light task-specific fine-tuning. Such an approach could reduce the cost of per-workload training, although trade-offs between generality and accuracy must be carefully evaluated.

Bibliography

[1] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association. (document), 2, 2.4, 2.2, 2.4.2, 2.4.6, 3.2.1, 3.2.2

[2] Flexera. 2024 state of the cloud report. `https://info.flexera.com/CM-REPORT-State-of-the-Cloud-2024-Thanks`, 2024. 1

[3] IDC. Worldwide software and public cloud services spending guide. `https://www.idc.com/getdoc.jsp?containerId=prUS52460024`, 2024. 1

[4] Gartner. Gartner forecasts worldwide public cloud end-user spending to surpass $675 billion in 2024. `https://www.gartner.com/en/newsroom/press-releases/2024-05-20-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-surpass-675-billion-in-2024`, 2024. 1

[5] Gartner. Keeping cloud costs in check: It leader perspectives. `https://www.gartner.com/peer-community/oneminuteinsights/omi-keeping-cloud-costs-check-it-leader-perspectives-rfz`, 2024. Accessed: 2025-07-25. 1

[6] Gartner. Gartner forecasts worldwide public cloud revenue to grow 17.5 percent in 2019. `https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g`, 2019. Accessed: 2025-07-25. 1

[7] Gartner. Gartner forecasts worldwide public cloud revenue to grow 6.3% in 2020. `https://www.gartner.com/en/newsroom/press-releases/2020-07-23-gartner-forecasts-worldwide-public-cloud-revenue-to-grow-6point3-percent-in-2020`, 2020. Accessed: 2025-07-25. 1

[8] Gartner. Gartner forecasts worldwide public cloud end-user spending to grow 23% in 2021. `https://www.gartner.com/en/newsroom/press-releases/2021-04-21-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-23-percent-in-2021`, 2021. Accessed: 2025-07-25. 1

[9] Gartner. Gartner forecasts worldwide public cloud end-user spending to reach nearly $500 billion in 2022. `https://www.gartner.com/en/newsroom/press-`

releases/2022-04-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-500-billion-in-2022, 2022. Accessed: 2025-07-25. 1

[10] Gartner. Gartner forecasts worldwide public cloud end-user spending to reach nearly $600 billion in 2023. https://www.gartner.com/en/newsroom/press-releases/2022-10-31-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-600-billion-in-2023, 2022. Accessed: 2025-07-25. 1

[11] Gartner. Gartner forecasts worldwide public cloud end-user spending to surpass $675 billion in 2024. https://www.gartner.com/en/newsroom/press-releases/2024-05-20-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-surpass-675-billion-in-2024, 2024. Accessed: 2025-07-25. 1

[12] Gartner. Gartner forecasts worldwide public cloud end-user spending to total $723 billion in 2025. https://www.gartner.com/en/newsroom/press-releases/2024-11-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-total-723-billion-dollars-in-2025, 2024. Accessed: 2025-07-25. 1

[13] Supreeth Shastri and David Irwin. Hotspot: automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 493–505, New York, NY, USA, 2017. Association for Computing Machinery. 1, 2

[14] Andrew Or, Haoyu Zhang, and Michael Freedman. Resource elasticity in distributed deep learning. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 400–411, 2020. 1, 2

[15] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 121–134, New York, NY, USA, 2018. Association for Computing Machinery. 1, 2

[16] Muhammad Bilal, Marco Canini, and Rodrigo Rodrigues. Finding the right cloud configuration for analytics clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 208–222, New York, NY, USA, 2020. Association for Computing Machinery. 1, 2, 2.2, 5.2.1

[17] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, March 2017. USENIX Association. 1, 2, 2, 2.2

[18] Virtana. The state of hybrid cloud storage: 2023 survey report. Technical report, Virtana, 2023. Accessed: 2025-07-24. 1

[19] Wasabi Technologies. 2025 anz executive summary: State of the cloud storage report. Technical report, Wasabi Technologies, 2025. Accessed: 2025-07-24. 1

[20] Hojin Park, Gregory R. Ganger, and George Amvrosiadis. Mimir: Finding cost-efficient storage configurations in the public cloud. In *Proceedings of the 16th ACM International Conference on Systems and Storage*, SYSTOR '23, page 22–34, New York, NY, USA, 2023. Association for Computing Machinery. 1.3, 3, 3.8

[21] Hojin Park, Ziyue Qiu, Gregory R. Ganger, and George Amvrosiadis. Reducing cross-cloud/region costs with the auto-configuring macaron cache. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 347–368, New York, NY, USA, 2024. Association for Computing Machinery. 1.3, 4

[22] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst.*, 19(4):483–518, November 2001. 2

[23] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. USENIX Association. 2

[24] John D. Strunk, Eno Thereska, and Christos Faloutsos. Using utility to provision storage systems. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association. 2

[25] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 759–773, Boston, MA, July 2018. USENIX Association. 2, 2.2, 3, 3.8, 5.2.1

[26] Ashraf Mahgoub, Alexander Michaelson Medoff, Rakesh Kumar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 189–203. USENIX Association, July 2020. 2, 2.2, 2.3.1, 2.4.3, 2.4.6, 3, 3.8

[27] Amazon Web Services. Amazon Elastic Block Store., June 2022. 2.1.1

[28] Microsoft. Azure Disk Storage, June 2022. 2.1.1

[29] Google. Google Compute Engine Persistent Disks., June 2022. 2.1.1

[30] Jens Axboe. Flexible I/O Tester, 2022. 2.1.1

[31] Amazon Web Services. Amazon EBS volume types., 2022. 2.1.1

[32] Flavio P. Junqueira, Ivan Kelly, and Benjamin Reed. Durability with BookKeeper. *SIGOPS Oper. Syst. Rev.*, 47(1):9–15, January 2013. 2.1.2

[33] Haoyu Wang, Haiying Shen, Qi Liu, Kevin Zheng, and Jie Xu. A reinforcement learning based system for minimizing cloud storage service cost. In *Proceedings of the 49th International Conference on Parallel Processing*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery. 2.2

[34] Oliver Niehorster, Alexander Krieger, Jens Simon, and Andre Brinkmann. Autonomic re-

source management with support vector machines. In *2011 IEEE/ACM 12th International Conference on Grid Computing*, pages 157–164, 2011. 2.2

[35] Viktor Leis and Maximilian Kuschewski. Towards cost-optimal query processing in the cloud. *Proc. VLDB Endow.*, 14(9):1606–1612, May 2021. 2.2

[36] Peipei Zhou, Jiayi Sheng, Cody Hao Yu, Peng Wei, Jie Wang, Di Wu, and Jason Cong. Mocha: Multinode cost optimization in heterogeneous clouds with accelerators. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 273–279, New York, NY, USA, 2021. Association for Computing Machinery. 2.2

[37] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association. 2.2, 3, 3.8

[38] Barzan Mozafari, Carlo Curino, and Samuel Madden. Dbseer: Resource and performance prediction for building a next generation database cloud. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013. 2.2

[39] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, page 127–144, New York, NY, USA, 2014. Association for Computing Machinery. 2.2

[40] Vasanth Balasundaram, Geoffrey Fox, Ken Kennedy, and Ulrich Kremer. A static performance estimator to guide data partitioning decisions. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, page 213–223, New York, NY, USA, 1991. Association for Computing Machinery. 2.2

[41] Kristi Morton, Magdalena Balazinska, and Dan Grossman. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, page 507–518, New York, NY, USA, 2010. Association for Computing Machinery. 2.2

[42] Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for sql queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 803–814, New York, NY, USA, 2004. Association for Computing Machinery. 2.2

[43] Salvatore Dipietro, Giuliano Casale, and Giuseppe Serazzi. A queueing network model for performance prediction of apache cassandra. In *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools on 10th EAI International Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS'16, page 186–193, Brussels, BEL, 2017. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 2.2

[44] Subrata Mitra, Shanka Subhra Mondal, Nikhil Sheoran, Neeraj Dhake, Ravinder Nehra, and Ramanuja Simha. Deepplace: Learning to place applications in multi-tenant clusters.

In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, page 61–68, New York, NY, USA, 2019. Association for Computing Machinery. 2.2

[45] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 270–288, New York, NY, USA, 2019. Association for Computing Machinery. 2.2

[46] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. Selecting the best vm across multiple public clouds: a data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 452–465, New York, NY, USA, 2017. Association for Computing Machinery. 2.2, 5.2.1

[47] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 363–378, USA, 2016. USENIX Association. 2.2

[48] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, Boston, MA, February 2019. USENIX Association. 2.3.1

[49] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. 2.3.1

[50] Markus Klems, Adam Silberstein, Jianjun Chen, Masood Mortazavi, Sahaya Andrews Albert, P.P.S. Narayan, Adwait Tumbde, and Brian Cooper. The yahoo! cloud datastore load balancer. In *Proceedings of the Fourth International Workshop on Cloud Data Management*, CloudDB '12, page 33–40, New York, NY, USA, 2012. Association for Computing Machinery. 2.3.1

[51] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Comput. Surv.*, 51(4), July 2018. 2.3.1

[52] Jerome Friedman. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics*, 29:1189–1232, 10 2001. 2.3.3, 5.2.1

[53] George E. Andrews. *The Theory of Partitions*. Cambridge University Press, 1976. 2.3.5

[54] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021. 2.4.1

[55] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. Differentiated Key-Value storage management for balanced I/O performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 673–687. USENIX Association, July 2021. 2.4.2

[56] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang,

Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: a raft-based htap database. *Proc. VLDB Endow.*, 13(12):3072–3084, August 2020. 2.4.2

[57] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. Atlas: Baidu's key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2015. 2.4.2

[58] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.*, 40(1):53–64, June 2012. 2.4.2

[59] Jonathan R. M. Hosking and Jamie Wallis. Parameter and quantile estimation for the generalized pareto distribution. *Technometrics*, 29:339–349, 1987. 2.4.2

[60] Manish Shukla and Sanjay Jharkharia. Applicability of arima models in wholesale vegetable market: An investigation. *Int. J. Inf. Syst. Supply Chain Manag.*, 6(3):105–119, July 2013. 2.4.6

[61] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *2012 IEEE Network Operations and Management Symposium*, pages 1287–1294, 2012. 2.4.6

[62] Peng Wang, Haixun Wang, and Wei Wang. Finding semantics in time series. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 385–396, New York, NY, USA, 2011. Association for Computing Machinery. 2.4.6

[63] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In *2007 IEEE 10th International Symposium on Workload Characterization*, pages 171–180, 2007. 2.4.6, 3.6.3

[64] Oracle. 98% of Enterprises Using Public Cloud Have Adopted a Multicloud Infrastructure Provider Strategy. `https://www.oracle.com/emea/news/announcement/98-percent-enterprises-adopted-multicloud-strategy-2023-02-09/`, 2023. 3

[65] AAG. The Latest Cloud Computing Statistics (updated March 2024) | AAG IT Support. `https://aag-it.com/the-latest-cloud-computing-statistics/`, 2024. Section: Business. 3

[66] Cody Slingerland. 101 Shocking Cloud Computing Statistics. `https://www.cloudzero.com/blog/cloud-computing-statistics/`, December 2023. 3

[67] Crayon. Understanding Hybrid Cloud vs. Multicloud: Key Differences Explained - Crayon. `https://www.crayon.com/us/resources/blogs/multicloud-a-guide-to-multicloud-strategy-management-and-multicloud-architecture/`, 2024. 3

[68] Liquid Web. The benefits and challenges of multi-cloud management. `https:`

//www.liquidweb.com/blog/multi-cloud-benefits-challenges/, February 2024. 3

[69] Justice Opara-Martins, Reza Sahandi, and Feng Tian. Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. *Journal of Cloud Computing*, 5(1):4, December 2016. 3

[70] Gabriel Costa Silva, Louis M. Rose, and Radu Calinescu. A Systematic Review of Cloud Lock-In Solutions. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, pages 363–368, Bristol, United Kingdom, December 2013. IEEE. 3

[71] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 126–138, New York, NY, USA, 2019. Association for Computing Machinery. 3, 3.8

[72] Lauren van der Vaart. How Salesforce® Optimizes Multi-Cloud Costs With CloudHealth® by VMware®. https://blogs.vmware.com/cloud/2021/07/19/salesforce-optimizes-multi-cloud-costs/, July 2021. 3

[73] Sandipan Sarkar. Living in a data sovereign world. https://www.ibm.com/blog/living-in-a-data-sovereign-world/, October 2023. 3

[74] Oracle. Oracle Cloud®for Sovereignty: Customized, Flexible Solutions. https://www.oracle.com/cloud/sovereign-cloud/, 2024. 3

[75] Microsoft. Data residency - Microsoft® Cloud for Sovereignty. https://learn.microsoft.com/en-us/industry/sovereignty/data-residency, December 2023. 3

[76] AWS. How to Accelerate Performance and Availability of Multi-region Applications with Amazon S3® Multi-Region Access Points | AWS News Blog. https://aws.amazon.com/blogs/aws/s3-multi-region-access-points-accelerate-performance-availability/, September 2021. Section: Amazon Simple Storage Service (S3). 3

[77] CockroachDB. CockroachDB® : The multi-region database. https://dantheengineer.com/cockroachdb-multi-region/, February 2023. 3

[78] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association. 3, 3.8

[79] Iwona Kotlarska, Andrzej Jackowski, Krzysztof Lichota, Michal Welnicki, Cezary Dubnicki, and Konrad Iwanicki. InftyDedup: Scalable and Cost-Effective cloud tiering with deduplication. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 33–48, Santa Clara, CA, February 2023. USENIX Association. 3, 3.8

[80] Saurabh Kadekodi, Bin Fan, Adit Madan, Garth A. Gibson, and Gregory R. Ganger. A case for packing and indexing in cloud file systems. In *10th USENIX Workshop on Hot*

Topics in Cloud Computing (HotCloud 18), Boston, MA, July 2018. USENIX Association. 3, 3.5.1, 3.8

[81] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. Exploiting Spot and Burstable Instances for Improving the Cost-efficacy of In-Memory Caches on the Public Cloud. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 620–634, New York, NY, USA, April 2017. Association for Computing Machinery. 3

[82] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. Skyplane: Optimizing transfer cost and throughput using Cloud-Aware overlays. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1375–1389, Boston, MA, April 2023. USENIX Association. 3, 3.1, 3.8

[83] Maomeng Su, Lei Zhang, Yongwei Wu, Kang Chen, and Keqin Li. Systematic Data Placement Optimization in Multi-Cloud Storage for Complex Requirements. *IEEE Transactions on Computers*, 65(6):1964–1977, June 2016. 3, 3.8

[84] Pengwei Wang, Caihui Zhao, Yi Wei, Dong Wang, and Zhaohui Zhang. An Adaptive Data Placement Architecture in Multicloud Environments. *Scientific Programming*, 2020:e1704258, June 2020. 3, 3.8

[85] Quanlu Zhang, Shenglong Li, Zhenhua Li, Yuanjian Xing, Zhi Yang, and Yafei Dai. CHARM: A Cost-Efficient Multi-Cloud Data Hosting Scheme with High Availability. *IEEE Transactions on Cloud Computing*, 3(3):372–386, July 2015. 3, 3.8

[86] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. SPANStore: cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 292–308, New York, NY, USA, November 2013. Association for Computing Machinery. 3, 3.8

[87] AWS. Accessing tables across Regions - AWS Lake Formation. `https://docs.aws.amazon.com/lake-formation/latest/dg/data-access-across-region.html`, 2024. 3

[88] AWS. Querying across regions - Amazon Athena®. `https://docs.aws.amazon.com/athena/latest/ug/querying-across-regions.html`, 2024. 3

[89] Google Cloud. Load data with cross-cloud operations | BigQuery®. `https://cloud.google.com/bigquery/docs/load-data-using-cross-cloud-transfer`, 2024. 3

[90] Azure. Cross-region replication in Azure®. `https://learn.microsoft.com/en-us/azure/reliability/cross-region-replication-azure`, April 2023. 3, 3.8, 4

[91] Bhagat Nimesh. Ensure business continuity across regions with replicas. `https://cloud.google.com/blog/products/databases/introducing-cross-region-replica-for-cloud-sql`, 2023. 3, 3.8, 4

[92] AWS. Replicating objects - Amazon Simple Storage Service. `https://docs.aws.`

amazon.com/AmazonS3/latest/userguide/replication.html, 2023. 3, 3.8, 4

[93] Snowflake. The Data Cloud | Snowflake®. https://www.snowflake.com/en/, 2023. 3, 3.8, 4

[94] JuiceFS. JuiceFS® - Open Source Distributed POSIX File System for Cloud. https://juicefs.com/en/, 2023. 3, 3.8, 4

[95] AWS. Redis®and Memcached-Compatible Cache – Amazon ElastiCache – Amazon Web Services. https://aws.amazon.com/elasticache/, 2024. 3, 3.8

[96] MS Azure. Azure Cache for Redis | Microsoft Azure. https://azure.microsoft.com/en-us/products/cache, 2024. 3, 3.8

[97] Google Cloud. Memorystore: in-memory Redis compatible data store. https://cloud.google.com/memorystore, 2024. 3, 3.8

[98] Alluxio. Alluxio®- Data Orchestration for the Cloud. https://www.alluxio.io/, 2024. 3, 3.8

[99] MinIO Inc. MinIO® | MinIO Enterprise Object Store Cache feature. https://min.io, 2024. 3, 3.8

[100] Avere. Avere vFXT - Scalable File System for Hybrid HPC | Microsoft Azure. https://azure.microsoft.com/en-us/products/storage/avere-vfxt, 2024. 3, 3.8

[101] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs. FIFO. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020. 3, 3.2.2, 3.5.1, 3.6.1

[102] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. USENIX Association. 3, 3.8

[103] Pei Cao and Sandy Irani. Cost-Aware WWW proxy caching algorithms. In *USENIX Symposium on Internet Technologies and Systems (USITS 97)*, Monterey, CA, December 1997. USENIX Association. 3, 3.8

[104] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001. Conference Name: IEEE Transactions on Computers. 3, 3.8

[105] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association. 3, 3.8

[106] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, June 2002. 3, 3.8

[107] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer

management replacement algorithm. In *Very Large Data Bases Conference*, 1994. 3, 3.8

[108] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association. 3, 3.8

[109] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery. 3, 3.8

[110] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, May 1999. 3, 3.8

[111] Chen Zhong, Xingsheng Zhao, and Song Jiang. LIRS2: an improved LIRS replacement algorithm. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR '21, New York, NY, USA, 2021. Association for Computing Machinery. 3, 3.8

[112] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 130–149, New York, NY, USA, 2023. Association for Computing Machinery. 3, 3.2.1, 3.8

[113] Yifan Dai, Jing Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Symbiosis: The art of application and kernel cache cooperation. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 51–69, Santa Clara, CA, February 2024. USENIX Association. 3, 3.3.1, 3.4.2

[114] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117, 2018. 3

[115] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, Santa Clara, CA, July 2015. USENIX Association. 3, 3.8

[116] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. Centaur: Host-side ssd caching for storage performance control. In *2015 IEEE International Conference on Autonomic Computing*, pages 51–60, 2015. 3

[117] Jaewon Kwak, Eunji Hwang, Tae-Kyung Yoo, Beomseok Nam, and Young-Ri Choi. In-memory caching orchestration for hadoop. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 94–97, 2016. 3

[118] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. vCacheShare: Automated server flash cache space management in a virtualization environment. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 133–144, Philadelphia,

PA, June 2014. USENIX Association. 3

[119] G. Suh, Larry Rudolph, and Sahana Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28:7–26, 04 2004. 3

[120] Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. predis: Penalty and locality aware memory allocation in redis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 193–205, New York, NY, USA, 2019. Association for Computing Machinery. 3

[121] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery. 3, 3.8

[122] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 487–498, Santa Clara, CA, July 2017. USENIX Association. 3, 3.3.1, 3.4.2

[123] Jonathan Baker Ziyue Qiu. Uber object access trace. `https://github.com/uber-research/presto-HDFS-read-data/`, 2024. 3, 4.3

[124] Hojin Park. Macaron prototype code. `https://github.com/hojinp/macaron/`, 2024. 3

[125] Hojin Park. Macaron simulator code. `https://github.com/hojinp/macaron_simulator/`, 2024. 3

[126] janober. Ask HN: Azure has run out of compute – anyone else affected? | Hacker News. `https://news.ycombinator.com/item?id=33743567`, 2024. 3.1

[127] Diana Goovaerts. The one where we hate on egress fees even more. `https://www.silverliningsinfo.com/ai/one-where-we-hate-egress-fees-even-more`, January 2024. 3.1

[128] Google Cloud. GPU regions and zones availability | Compute Engine Documentation. `https://cloud.google.com/compute/docs/gpus/gpu-regions-zones`, 2024. 3.1

[129] MS Azure. Azure Products by Region | Microsoft Azure. `https://azure.microsoft.com/en-us/explore/global-infrastructure/products-by-region/`, 2024. 3.1

[130] Snowflake Staff. Pfizer® uses Snowgrid® for cross-region collaboration and business continuity. `https://www.snowflake.com/blog/pfizer-uses-snowgrid-for-cross-region-collaboration/`, April 2023. 3.1

[131] Denis Magda. Multi-Region Applications With Kong Mesh and YugabyteDB®. `https://medium.com/@magda7817/multi-region-applications-with-kong-mesh-and-yugabytedb-3f472b4dc88f`, September 2023. 3.1

[132] K. Indira and M. K. Kavithadevi. Efficient Machine Learning Model for Movie Recommender Systems Using Multi-Cloud Environment. *Mobile Networks and Applications*,

24(6):1872–1882, December 2019. 3.1

[133] Pasquale Salza, Erik Hemberg, Filomena Ferrucci, and Una-May O'Reilly. Towards evolutionary machine learning comparison, competition, and collaboration with a multi-cloud platform. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 1263–1270, New York, NY, USA, July 2017. Association for Computing Machinery. 3.1

[134] Charlie Custer. The state of multi-cloud 2024, 2024. Accessed: 2024-07-20. 3.1, 3.1

[135] Laurent Gil. GPU Shortage Mitigation: How to Harness the Cloud Automation Advantage. `https://cast.ai/blog/gpu-shortage-mitigation-how-to-harness-the-cloud-automation-advantage/`, June 2023. 3.1

[136] Senior Writer Michael Chen. Do You Know What Multicloud Is? `https://www.oracle.com/cloud/multicloud/what-is-multicloud/`, 2024. 3.1

[137] Andrew Smith, Natalya Yezhkova, and Ashish Nadkarni. Future-proofing storage: Modernizing Infrastructure for Data Growth Across Hybrid, Edge, and Cloud Ecosystems. *IDC*, March 2021. 3.1

[138] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, December 1968. Association for Computing Machinery. 3.1

[139] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. The Internet at the Speed of Light. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 1–7, New York, NY, USA, October 2014. Association for Computing Machinery. 3.1

[140] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 557–573, New York, NY, USA, 2023. Association for Computing Machinery. 3.2.1

[141] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020. 3.2.1, 3.2.2, 3.5.2

[142] Ce Zhang, Jaeho Shin, Christopher Ré, Michael Cafarella, and Feng Niu. Extracting Databases from Dark Data with DeepDive. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 847–859, New York, NY, USA, June 2016. Association for Computing Machinery. 3.2.1

[143] P. Bryan Heidorn. Shedding Light on the Dark Data in the Long Tail of Science. *Library Trends*, 57(2):280–299, 2008. 3.2.1

[144] David J. Hand. Dark Data: Why What You Don't Know Matters. In *Dark Data*. Princeton University Press, February 2020. 3.2.1

[145] Jonathan Johnson. What Is Dark Data? The Basics & The Challenges. `https://www.bmc.com/blogs/dark-data/`, 2023. 3.2.1

[146] Aparavi Adrian Knapp. Businesses must eliminate the unnecessary energy costs of data processing, August 2022. `https://venturebeat.com/data-infrastructure/businesses-must-eliminate-the-unnecessary-energy-costs-of-data-processing/`. 3.2.1

[147] Maria Korolov. Unlocking the hidden value of dark data. `https://www.cio.com/article/404526/unlocking-the-hidden-value-of-dark-data.html`, 2023. 3.2.1

[148] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1917–1923, Melbourne Victoria Australia, May 2015. ACM. 3.2.1

[149] Yang Yang. Presto® on Apache Kafka® At Uber Scale. `https://www.uber.com/en-AU/blog/presto-on-apache-kafka-at-uber-scale/`, April 2022. 3.2.2

[150] Sabina Hristova. Join Us in Transforming Cybersecurity. `https://blogs.vmware.com/bulgaria/2021/06/01/join-us-in-transforming-cybersecurity/`, June 2021. 3.2.2

[151] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320)*, volume 1, pages 126–134 vol.1, March 1999. ISSN: 0743-166X. 3.2.2

[152] Damiano Carra and Giovanni Neglia. Efficient miss ratio curve computation for heterogeneous content popularity. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 741–751. USENIX Association, July 2020. 3.2.2, 3.4.2

[153] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. HotRing: A Hotspot-Aware In-Memory Key-Value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 239–252, Santa Clara, CA, February 2020. USENIX Association. 3.2.2

[154] George Kingsley Zipf. Relative Frequency as a Determinant of Phonetic Change. *Harvard Studies in Classical Philology*, 40:1–95, 1929. 3.2.2

[155] Or Ozeri, Effi Ofer, and Ronen Kat. Object Storage for Deep Learning Frameworks. In *Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning*, DIDL '18, pages 21–24, New York, NY, USA, December 2018. Association for Computing Machinery. 3.2.2

[156] AWS. Using Amazon S3 with Amazon ML - Amazon Machine Learning. `https://docs.aws.amazon.com/machine-learning/latest/dg/using-amazon-s3-with-amazon-ml.html`, 2024. 3.2.2

[157] Z. Daher and Hassan Hajjdiab. Cloud storage comparative analysis amazon simple storage vs. microsoft azure blob storage. *International Journal of Machine Learning and Computing*, 8:85–89, 02 2018. 3.2.2

[158] Matei A. Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *Conference on Innovative Data Systems Research*, 2021. 3.2.2, 3.3.1

[159] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. Delta lake: high-performance acid table storage over cloud object stores. *Proc. VLDB Endow.*, 13(12):3411–3424, August 2020. 3.2.2, 4.3.3

[160] Eftim Zdravevski, Petre Lameski, Ace Dimitrievski, Marek Grzegorowski, and Cas Apanowicz. Cluster-size optimization within a cloud-based etl framework for big data. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 3754–3763, 2019. 3.2.2

[161] Mengchu Cai, Martin Grund, Anurag Gupta, Fabian Nagel, Ippokratis Pandis, Yannis Papakonstantinou, and Michalis Petropoulos. Integrated querying of sql database data and s3 data in amazon redshift. *IEEE Data Eng. Bull.*, 41:82–90, 2018. 3.2.2

[162] AWS. Interactive SQL - Amazon Athena - AWS. `https://aws.amazon.com/athena/`, 2024. 3.2.2

[163] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 131–141, New York, NY, USA, 2020. Association for Computing Machinery. 3.2.2

[164] Mahmoud Ismail, Salman Niazi, Gautier Berthou, Mikael Ronström, Seif Haridi, and Jim Dowling. Hopsfs-s3: Extending object stores with posix-like semantics and more (industry track). In *Proceedings of the 21st International Middleware Conference Industrial Track*, Middleware '20, page 23–30, New York, NY, USA, 2020. Association for Computing Machinery. 3.2.2

[165] Kunal Lillaney, Vasily Tarasov, David Pease, and Randal Burns. The case for dual-access file systems over object storage. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association. 3.2.2

[166] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: A shared cloud-backed file system. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 169–180, Philadelphia, PA, June 2014. USENIX Association. 3.2.2

[167] Haoqiong Bian and Anastasia Ailamaki. Pixels: An Efficient Column Store for Cloud Data Lakes. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 3078–3090, May 2022. ISSN: 2375-026X. 3.3.1

[168] Anna Levin, Shelly Garion, Elliot K. Kolodner, Dean H. Lorenz, Katherine Barabash, Mike Kugler, and Niall McShane. AIOps for a Cloud Object Storage Service. In *2019 IEEE International Congress on Big Data (BigDataCongress)*, pages 165–169, July 2019. ISSN: 2642-7273. 3.3.1

[169] Hyojun Kim, Sangeetha Seshadri, Clement L. Dickey, and Lawrence Chiu. Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches. *ACM Transactions on Storage*, 10(4):15:1–15:21, October 2014. 3.3.1

[170] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. Automated synthesis of adversarial workloads for network functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 372–385, New York, NY, USA, August 2018. Association for Computing Machinery. 3.3.1

[171] Leul Belayneh, Haojie Ye, Kuan-Yu Chen, David Blaauw, Trevor Mudge, Ronald Dreslinski, and Nishil Talati. Locality-Aware Optimizations for Improving Remote Memory Latency in Multi-GPU Systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, pages 304–316, New York, NY, USA, January 2023. Association for Computing Machinery. 3.3.1

[172] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 243–262, New York, NY, USA, October 2021. Association for Computing Machinery. 3.3.1

[173] Yongseok Oh, Eunjae Lee, Choulseung Hyun, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Enabling Cost-Effective Flash based Caching with an Array of Commodity SSDs. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 63–74, New York, NY, USA, November 2015. Association for Computing Machinery. 3.3.1

[174] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for google datacenter flash caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1021–1036, Carlsbad, CA, July 2022. USENIX Association. 3.3.1

[175] Rong Gu, Simian Li, Haipeng Dai, Hancheng Wang, Yili Luo, Bin Fan, Ran Ben Basat, Ke Wang, Zhenyu Song, Shouwei Chen, Beinan Wang, Yihua Huang, and Guihai Chen. Adaptive online cache capacity optimization via lightweight working set size estimation at scale. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 467–484, Boston, MA, July 2023. USENIX Association. 3.3.1

[176] Ashraf Mahgoub, Paul Wood, Alexander Medoff, Subrata Mitra, Folker Meyer, Somali Chaterji, and Saurabh Bagchi. SOPHIA: Online reconfiguration of clustered NoSQL databases for Time-Varying workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 223–240, Renton, WA, July 2019. USENIX Association. 3.3.1

[177] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger.

Baleen: ML admission & prefetching for flash caches. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 347–371, Santa Clara, CA, February 2024. USENIX Association. 3.3.1

[178] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 153–167, New York, NY, USA, 2017. Association for Computing Machinery. 3.3.1

[179] Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, et al. MAST: Global scheduling of ML training across Geo-Distributed datacenters at hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 563–580, 2024. 3.3.3

[180] Yuzhen Huang, Yingjie Shi, Zheng Zhong, Yihui Feng, James Cheng, Jiwei Li, Haochuan Fan, Chao Li, Tao Guan, and Jingren Zhou. Yugong: Geo-distributed data and job placement at scale. *Proceedings of the VLDB Endowment*, 12(12):2155–2169, 2019. 3.3.3

[181] Oz Katz. lakefs and amazon s3 express one zone: Highly performant data version control for ml/ai. `https://aws.amazon.com/blogs/storage/lakefs-and-amazon-s3-express-one-zone-highly-performant-data-version-control-for-ml-ai/`, 2024. 3.3.3

[182] Ville Satopaa, Jeannie Albrecht, David Irwin, and Barath Raghavan. Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior. In *2011 31st International Conference on Distributed Computing Systems Workshops*, pages 166–171, June 2011. ISSN: 2332-5666. 3.4.1

[183] Frank Olken. Efficient methods for calculating the success function of fixed space replacement policies. *Perform. Evaluation*, 3(2):153–154, 1983. 3.4.2

[184] Qingpeng Niu, James Dinan, Qingda Lu, and P. Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, page 1284–1294, USA, 2012. IEEE Computer Society. 3.4.2

[185] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, February 2015. USENIX Association. 3.4.2

[186] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 351–364, Denver, CO, June 2016. USENIX Association. 3.4.2

[187] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 335–349, 2014. 3.4.2

[188] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. 3.4.4

[189] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association. 3.5.1

[190] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 115–134, Santa Clara, CA, February 2023. USENIX Association. 3.5.1

[191] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, pages 1–11, New York, NY, USA, May 2017. Association for Computing Machinery. 3.5.2

[192] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. Saving cash by using less cache. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12)*, Boston, MA, June 2012. USENIX Association. 3.5.2, 3.8

[193] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer:Auto-Sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016. 3.5.3

[194] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, et al. Shard manager: A generic shard management framework for geo-distributed applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 553–569, 2021. 3.5.3

[195] Mikhail Genkin and Frank Dehne. Autonomic workload change classification and prediction for big data workloads. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 2835–2844, 2019. 3.6.3

[196] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with In-Network coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 387–406. USENIX Association, November 2020. 3.6.3

[197] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, Shanghai China, October 2017. ACM. 3.6.3

[198] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implemen-*

tation (OSDI 18), pages 195–212, Carlsbad, CA, October 2018. USENIX Association. 3.8

[199] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, Santa Clara, CA, July 2017. USENIX Association. 3.8

[200] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K. Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '17, page 55–67, New York, NY, USA, 2017. Association for Computing Machinery. 3.8

[201] Emine Ugur Kaynar, Mania Abdi, Mohammad Hossein Hajkazemi, Ata Turk, Raja R. Sambasivan, David Cohen, Larry Rudolph, Peter Desnoyers, and Orran Krieger. D3n: A multi-layer cache for the rest of us. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 327–338, 2019. 3.8

[202] Farhana Kabir and David Chiu. Reconciling cost and performance objectives for elastic web caches. In *2012 International Conference on Cloud and Service Computing*, pages 88–95, 2012. 3.8

[203] Jinho Hwang and Timothy Wood. Adaptive Performance-Aware distributed memory caching. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 33–43, San Jose, CA, June 2013. USENIX Association. 3.8

[204] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, George Kesidis, and Qianlin Liang. Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 620–634, New York, NY, USA, 2017. Association for Computing Machinery. 3.8

[205] Muhammed Uluyol, Anthony Huang, Ayush Goel, Mosharaf Chowdhury, and Harsha V Madhyastha. Near-Optimal latency versus cost tradeoffs in Geo-Distributed storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 157–180, 2020. 3.8

[206] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Habinder Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *NSDI*, 2010. 3.8

[207] Tiemo Bang, Chris Douglas, Natacha Crooks, and Joseph M. Hellerstein. Skypie: A fast & accurate oracle for object placement. *Proc. ACM Manag. Data*, 2(1), mar 2024. 3.8

[208] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-Effective redundancy for lower latency variance on cloud storage services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 543–557, Oakland, CA, May 2015. USENIX Association. 3.8

[209] "Snowflake". Introduction to replication and failover across multiple accounts |

Snowflake Documentation. https://docs.snowflake.com/en/user-guide/account-replication-intro, 2024. 3.8

[210] Memcache. memcached - a distributed memory object caching system. https://memcached.org/, 2023. 3.8

[211] Redis. Redis. https://redis.io/, 2023. 3.8

[212] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. C-miner: mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST'04, page 13, USA, 2004. USENIX Association. 4, 4.2

[213] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 66–79, New York, NY, USA, 2017. Association for Computing Machinery. 4, 4.2, 4.5.1

[214] Suli Yang, Kiran Srinivasan, Kishore Udayashankar, Swetha Krishnan, Jingxin Feng, Yupu Zhang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Tombolo: Performance enhancements for cloud storage gateways. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, 2016. 4, 4.1, 4.2, 4.5.1

[215] Binny S. Gill and Luis Angel D. Bathen. AMP: Adaptive multi-stream prefetching in a shared cache. In *5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. USENIX Association. 4, 4.2, 4.4.5

[216] Binny S. Gill and Dharmendra S. Modha. SARC: Sequential prefetching in adaptive replacement cache. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association. 4, 4.2, 4.4.5, 4.5.1

[217] Mingju Li, Elizabeth Varki, Swapnil Bhatia, and Arif Merchant. Tap: table-based prefetching for storage caches. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, USA, 2008. USENIX Association. 4, 4.2

[218] Yiyuan Yang, Rongshang Li, Qiquan Shi, Xijun Li, Gang Hu, Xing Li, and Mingxuan Yuan. Sgdp: A stream-graph neural network based data prefetcher, 2023. 4, 4.2, 4.4.2

[219] Farzaneh Zirak, Farhana Choudhury, and Renata Borovica-Gajic. Selep: Learning based semantic prefetching for exploratory database workloads. *Proc. VLDB Endow.*, 17(8):2064–2076, April 2024. 4, 4.2

[220] Gaddisa Olani Ganfure, Chun-Feng Wu, Yuan-Hao Chang, and Wei-Kuan Shih. Deepprefetcher: A deep learning framework for data prefetching in flash storage devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3311–3322, 2020. 4, 4.2, 4.4.2

[221] Amazon Web Services. Using amazon aurora global database. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-global-database.html, 2024. 4.1

[222] Amazon Web Services. Query across regions. https://docs.aws.amazon.com/athena/latest/ug/querying-across-regions.html, 2024. 4.1

[223] Azure. Designing globally available services using azure sql database. https://learn.microsoft.com/en-us/azure/azure-sql/database/designing-cloud-solutions-for-disaster-recovery, 2024. 4.1

[224] Azure. Distribute your data globally with azure cosmos db. https://learn.microsoft.com/en-us/azure/cosmos-db/distribute-data-globally, 2025. 4.1

[225] Cockroach Labs. Multi-region capabilities overview. https://www.cockroachlabs.com/docs/stable/multiregion-overview, 2025. Accessed: 2025-07-20. 4.1

[226] Yugabyte. Multi-region deployments in yugabytedb. https://docs.yugabyte.com/preview/explore/multi-region-deployments/, 2025. Accessed: 2025-07-20. 4.1

[227] Google Cloud. Introduction to external tables — bigquery. https://cloud.google.com/bigquery/docs/external-tables, 2025. Accessed: 2025-07-20. 4.1

[228] P. Reungsang, Sun Kyu Park, Seh-Woong Jeong, Hyung-Lae Roh, and Gyungho Lee. Reducing cache pollution of prefetching in a small data cache. In *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD 2001*, pages 530–533, 2001. 4.1

[229] Vivek Seshadri, Samihan Yedkar, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks. *ACM Trans. Archit. Code Optim.*, 11(4), January 2015. 4.1

[230] Farhan Tauheed, Thomas Heinis, Felix Schürmann, Henry Markram, and Anastasia Ailamaki. Scout: prefetching for latent structure following queries. *Proc. VLDB Endow.*, 5(11):1531–1542, July 2012. 4.1

[231] Chandranil Chakraborttii and Heiner Litz. Learning i/o access patterns to improve prefetching in ssds. In *Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track: European Conference, ECML PKDD 2020, Ghent, Belgium, September 14–18, 2020, Proceedings, Part IV*, page 427–443, Berlin, Heidelberg, 2020. Springer-Verlag. 4.2, 4.4.2, 4.5.1

[232] Pang-Ning Tan, Michael Steinbach, Abhishek Karpatne, and Vipin Kumar. *Introduction to Data Mining*. Pearson, 2nd edition, 2018. 4.3.1

[233] Michael Mesnier, Eno Thereska, Gregory R. Ganger, and Daniel Ellard. File classification in self-* storage systems. In *Proceedings of the First International Conference on Autonomic Computing*, ICAC '04, page 44–51, USA, 2004. IEEE Computer Society. 4.3.3

[234] Peng Xia, Dan Feng, Hong Jiang, Lei Tian, and Fang Wang. Farmer: a novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale file system performance. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, page 185–196, New York, NY, USA,

2008. Association for Computing Machinery. 4.3.3

[235] Fons Wijnhoven, Chintan Amrit, and Pim Dietz. Value-based file retention: File attributes as file value and information waste indicators. *J. Data and Information Quality*, 4(4), May 2014. 4.3.3

[236] DuckDB Documentation. Hive-style partitioning. `https://duckdb.org/docs/stable/data/partitioning/hive_partitioning.html`, 2024. Accessed: 2025-07-21. 4.3.3

[237] FullStory Developer. Google cloud storage destination. `https://developer.fullstory.com/destinations/google-cloud-storage/`, 2024. Accessed: 2025-07-21. 4.3.3

[238] Microsoft Azure Databricks. Auto loader schema inference and evolution. `https://learn.microsoft.com/en-us/azure/databricks/ingestion/cloud-object-storage/auto-loader/schema`, 2024. Accessed: 2025-07-21. 4.3.3

[239] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 604–613, New York, NY, USA, 1998. Association for Computing Machinery. 4.4.2

[240] I.T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer-Verlag, New York, 2nd edition, 2002. 4.4.2

[241] Ping Li, Trevor J. Hastie, and Kenneth W. Church. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 287–296, New York, NY, USA, 2006. Association for Computing Machinery. 4.4.2

[242] Spotify. Voyager: An efficient approximate nearest neighbor library. `https://github.com/spotify/voyager`, 2020. 4.4.4

[243] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018. 4.4.4

[244] Torvalds, Linus and the Linux Kernel Community. Linux Kernel Source Code: mm/readahead.c. `https://github.com/torvalds/linux/blob/master/mm/readahead.c`, 2025. Accessed: 2025-07-24. 4.4.5, 4.5.1

[245] T. Cortes and J. Labarta. Linear aggressive prefetching: a way to increase the performance of cooperative caches. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, pages 46–54, 1999. 4.4.5

[246] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *1993 International Conference on Parallel Processing - ICPP'93*, volume 1, pages 56–63, 1993. 4.4.5

[247] K.J. Nesbit and J.E. Smith. Data cache prefetching using a global history buffer. In *10th*

International Symposium on High Performance Computer Architecture (HPCA'04), pages 96–96, 2004. 4.5.1

[248] J. Z. Teng and R. A. Gumaer. Managing ibm database 2 buffers to maximize performance. *IBM Syst. J.*, 23(2):211–218, June 1984. 4.5.1

[249] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, August 1984. 4.5.1

[250] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, page 198–212, New York, NY, USA, 1991. Association for Computing Machinery. 4.5.1