UMEÅ UNIVERSITY

Breaking Up with Redis: A Comparative Study of Its Open-Source Alternatives

Evaluating Valkey, KeyDB and Garnet

Carl-Johan Fauvelle Munck af Rosenschöld

Carl-Johan Fauvelle Munck af Rosenschöld
Spring 2025
Degree Project in Computing Science and Engineering, 30 credits
Supervisor: Feras M. Awaysheh
Extern Supervisor: Jakub Krzywda
Examiner: Henrik Björklund
Master of Science Programme in Computing Science and Engineering, 300 credits

Abstract

Redis has long been the standard for in-memory key–value datastores, but recent licensing changes have prompted many to explore open-source alternatives. This thesis evaluates Valkey, KeyDB, and Garnet as potential successors to Redis, focusing on three core aspects: performance, migration effort, and long-term viability.

Each system was deployed in a Kubernetes-based environment and benchmarked using a modified version of `memtier_benchmark`. Realistic, Redis-like workloads were used, including Zipfian key-access patterns and varying concurrency levels. In addition to performance metrics like throughput, tail latency, and resource usage, the evaluation also considered migration complexity, such as compatibility with Redis tooling and API, and long-term viability, including development activity, governance, and open-source licensing.

Garnet delivered the highest performance across all metrics, outperforming Redis by a wide margin in throughput and latency while using significantly less memory. However, Garnet is only partially compatible with Redis and does not fully integrate with existing infrastructure, which increases migration complexity. Valkey performed similarly to Redis and scaled better under load, particularly in read-heavy workloads. As a drop-in replacement, Valkey supports the full Redis API and existing workflows, making migration straightforward. KeyDB showed limited performance improvements and no visible development activity during the evaluation period, raising concerns about sustainability.

The results show that Valkey is the most viable long-term alternative to Redis for general use, offering compatibility and operational continuity. Garnet is a strong option for performance-critical applications where full Redis compatibility is not required.

This study provides a structured, data-driven comparison to support engineers and decision-makers navigating the current Redis ecosystem.

Acknowledgements

I would like to thank Elastisys for giving me the opportunity to carry out this thesis in collaboration with them. I am especially grateful to my supervisors, who have stood by me throughout the entire process, Jakub Krzywda at Elastisys, and Feras M. Awaysheh at Umeå University. Their feedback, insightful discussions, and continuous support were invaluable from start to finish.

Contents

1 Introduction

In-memory datastores are a fundamental component of modern distributed infrastructure, providing extremely fast data access compared to traditional disk-based databases [1, 2]. Given their low latency, they are primarily used for caching, improving responsiveness and overall system efficiency [3, 4]. Redis, in particular, has become a standard for in-memory key-value datastores over the years due to its simplicity, performance, and extensive ecosystem [5, 6].

However, Redis has recently undergone a licensing change that has impacted its adoption, particularly among communities and organisations prioritising open-source principles [7, 8]. Following the Redis licensing change, Valkey, KeyDB, and Garnet have stepped up as open-source contenders for its replacement [9, 10, 11]. While this study was conducted, Redis underwent another licensing change [12], returning to a fully open-source license. However, the damage was already done, Redis had lost the trust of a community that values stable, transparent governance and a consistent commitment to open-source principles [8, 13].

This thesis presents a comparative evaluation of these emerging systems, assessing their potential to serve as open-source successors to Redis. Three key aspects form the basis of this evaluation: performance under Redis-typical workloads, compatibility with existing Redis deployments and ease of migration, as well as long-term viability in terms of development activity, governance, and licensing.

To ensure a fair and representative comparison, each system is deployed under identical conditions in a Kubernetes environment and benchmarked using realistic, Redis-like workloads that reflect typical usage patterns. Metrics such as throughput, tail latency, CPU and memory efficiency are analysed across a range of concurrency levels. The study also considers operational and migration challenges, examining how well each system integrates into existing Redis-based workflows and observability stacks.

Given Redis's central role in distributed systems, this study seeks to equip engineers and decision-makers with a clear understanding of the available alternatives. Evaluating performance, compatibility, and long-term sustainability holistically provides data-driven insights to support informed decision-making.

1.1 Research Scope and Objectives

This thesis aims to systematically evaluate Valkey, KeyDB, and Garnet as open-source successors to Redis, focusing on their suitability for cloud-native deployments. Given Redis's widespread use in modern infrastructure and recent licensing changes that have prompted a shift toward open and stable alternatives, this study provides a structured and evidence-based evaluation of these emerging systems. The goal is ultimately to identify the most

performant system and the most sustainable and operationally feasible option for real-world use in Kubernetes-based environments.

To achieve this, the study will answer the following research questions:

Performance. How do Redis alternatives perform under realistic Redis-like workloads in terms of throughput, latency, and resource efficiency when deployed in a Kubernetes environment?

Migration Complexity and Ecosystem Compatibility. To what extent are Redis alternatives compatible with Redis in terms of API support, deployment tooling, and operational workflows, and what migration challenges arise when replacing Redis with each system?

Long-Term Viability. How do the development activity, governance models, community support, and open-source licenses of Redis alternatives impact their long-term viability as Redis successors?

2 Background

This study relies on several fundamental areas of technology. This chapter introduces the core concepts and technologies used, including distributed systems, Kubernetes, Helm, and open-source licensing. It also presents an overview of in-memory key-value datastores, specifically Redis and its successors, and introduces the concept of benchmarking as the approach used to evaluate their performance.

This background provides the necessary context for understanding the systems evaluated and the methods used in the study.

2.1 In-Memory Key-Value Datastores

An in-memory key-value datastore is a database that stores all data directly in Random Access Memory (RAM), bypassing the traditional bottlenecks associated with disk I/O [2]. This results in significantly lower latency and higher throughput, making them ideal for performance-critical applications.

In a key-value datastore, data is organised as a collection of key-value pairs:

$$\text{Datastore} = \{k_1 \to v_1, k_2 \to v_2, \ldots, k_n \to v_n\}$$

Each unique key k_i maps to a corresponding value v_i.

Unlike traditional disk-based (e.g., relational) databases, which focus on persistence and complex querying, in-memory key-value stores prioritise speed and simplicity. They are commonly used in caching layers, session management, and real-time analytics, where rapid data access is essential.

2.1.1 Redis

Redis is an in-memory key-value store built around a single-threaded event loop, where one thread handles client requests from start to finish. This design simplifies concurrency by ensuring operations are executed sequentially and atomically, but it also limits vertical scalability since only one CPU core is utilised. Horizontal scaling is supported through clustering [14].

Redis [15] is the most widely adopted in-memory key-value store and ranks at the top of key-value database popularity lists [6]. It belongs to the family of NoSQL databases and is designed for high performance and architectural simplicity, making it a key component in modern cloud-native and distributed systems [5, 16].

Its primary strength lies in caching. Redis is commonly used to accelerate data access by storing frequently retrieved or precomputed results [1, 3]:

Storage caching. Where the whole dataset is stored in a disk-based database, temporarily holding the most frequently accessed data in-memory. This reduces the need for expensive queries to the primary database and improves overall system responsiveness.

Computation caching. Caches the results of computationally expensive operations, such as machine learning predictions or stream processing outputs, to avoid repeated calculations and reduce latency.

Transient data caching. Stores temporary data that exists only in memory and is not persisted in any form. This is typically used for short-lived values that does not need to be permanently stored.

Furthermore, Redis has a mature observability ecosystem. The built-in `INFO` command exposes a broad set of runtime statistics, including memory usage, command rates, CPU load, and keyspace activity [17]. These metrics are commonly collected using a Prometheus-compatible Redis exporter and visualised with Grafana dashboards [18], which are standard tools in modern monitoring stacks. This tight integration with the Prometheus and Grafana ecosystem is widely adopted in Kubernetes environments and production deployments, allowing operators to monitor system health, resource usage, and performance trends over time.

2.1.2 Successors to Redis

Due to Redis's licensing changes [7], several open-source alternatives have emerged to provide long-term, unrestricted alternatives. Although Redis has since returned to an OSI-approved open-source license (AGPL) [12], the abrupt shifts have already impacted trust within the community. This underscores the importance of long-term stability and transparent governance in critical infrastructure components.

Valkey

Valkey [19] is a community-driven fork of Redis, created from version 7.2.4 with the aim of preserving its open-source foundation. It is designed to be a drop-in replacement for Redis, with the goal of maintaining compatibility with its ecosystem. The project also positions itself as a stable and long-term solution for users seeking continuity. Like Redis, Valkey is single-threaded and relies on horizontal scaling through sharding, having multiple instances of the system working together. Valkey is actively developed under the Linux Foundation and has gained support from major industry stakeholders [20].

KeyDB

KeyDB [21] is a fork of Redis that addresses the limitations of Redis's single-threaded architecture by implementing multithreading. Each client connection is assigned to a thread when accepted, and network I/O and query parsing are handled concurrently. Instead of a single global event loop, KeyDB runs a separate Redis-style event loop on each thread to manage its assigned connections [22]. Access to the shared data structures is coordinated using spinlocks, allowing multiple threads to process commands in parallel while preserving

data consistency. This enables vertical scaling and improves throughput while maintaining compatibility with the Redis ecosystem [10].

Garnet

Garnet [23] aims to be a high-performance, open-source alternative for Redis developed by Microsoft Research. Unlike Valkey and KeyDB, which are forks of Redis, Garnet is a full reimplementation with a different architecture. It is designed for multithreading from the start and does not use Redis's event loop model. Instead, it uses I/O completion threads, where each thread handles network communication, command execution, and response independently [11].

Internally, Garnet uses a custom storage engine called Tsavorite, based on the FASTER key-value store [24]. Rather than relying on locks or spinlock, like KeyDB, to protect shared data structures, Tsavorite uses latch-free designs and coordination-free access patterns. It applies atomic operations to update shared structures, allowing multiple threads to operate in parallel without blocking each other. These techniques help reduce contention and are designed to support high-throughput, concurrent workloads [25].

2.2 Distributed Systems

Distributed systems are systems in which a collection of autonomous computers, often referred to as nodes, are located on separate hardware and networks and work together as if it was a single system [26].

Many modern key-value datastores, such as Redis and its successors, are designed to function in a distributed system. This enables them to scale horizontally as concurrency increases, by so-called sharding, having multiple instances of the system working together and distributing the load among themselves. It may also enable high availability by having multiple nodes ensure that the system remains up and running even if something happens, like hardware failure on one node. Systems like Redis are often used as components in larger distributed architectures [27].

2.2.1 Kubernetes

Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerised applications. It organises containers, lightweight, standalone units that run applications and their dependencies, into logical groups for easier management and discovery. Kubernetes ensures consistent and reliable execution of these containers, handling tasks such as scheduling, load balancing, and self-healing [28].

2.2.2 Helm

Helm is a Kubernetes package manager that simplifies application deployment by using reusable configuration templates, called charts. It is commonly used in both production and development environments to manage complex Kubernetes setups more easily [29].

2.3 Open-Source and License Model

Open-source software refers to software that has its source code publicly available and that it may be freely used, modified, and redistributed by anyone [30]. This model promotes transparency and collaboration, which in turn accelerates innovation, particularly within the software development community.

What makes software truly open-source is the license model under which the software was released. It is the license that determines whether users are legally granted the freedoms typically associated with open-source. For a license to be recognised as fully open-source, it is often expected to be approved by the Open Source Initiative (OSI) [30].

The OSI outlines ten criteria a license must fulfil to qualify as truly open-source, covering aspects such as free redistribution, access to source code, permission to modify and share derivative works, and non-discrimination against users or fields of use. These criteria aim to ensure that open-source software remains accessible for anyone, in any community or commercial context [31].

2.4 Benchmarking In-Memory Datastore Systems

Benchmarking is a process of evaluating how a system performs under controlled and repeatable workloads. In the context of in-memory datastores, benchmarking is essential for measuring systems performance under different scenarios. It reveals a system's capacity when it comes to throughput, latency, and resource efficiency, which are crucial metrics for any in-memory datastores [32].

Benchmarking workloads are often designed to reflect typical usage patterns [33]. Different systems are often optimised for specific workload types and may perform well under certain conditions but poorly under others. For example, a system designed for read-heavy traffic might perform efficiently in retrieval-focused scenarios but struggle with write-heavy loads. In addition, the pattern in which keys are accessed, such as uniform, normal, latest, or zipfian distributions, can significantly affect performance. Skewed patterns, where a small set of keys are accessed much more often, can create uneven load and put extra pressure on parts of the system. Therefore, identifying what constitutes a typical workload for the intended use case is crucial for benchmarking. Without this, results may fail to reflect real-world behaviour and lead to misleading conclusions when comparing systems.

Several tools have been developed for benchmarking in-memory systems. One widely recognised example is the Yahoo! Cloud Serving Benchmark (`YCSB`), designed as a standard tool to evaluate and compare NoSQL databases [33]. It includes a set of predefined workloads simulating various real-world scenarios. Another tool is `memtier_benchmark`, developed by Redis, which is tailored for benchmarking Redis and similar in-memory stores [34]. Unlike `YCSB`, it is better suited for high-concurrency and stress testing, offering more control over workload configuration through custom setups rather than predefined ones.

3 Related Work

In-memory data stores like Redis are widely used to provide fast, low-latency access to data in real-time systems. They are often embedded in infrastructure as caches. While Studies such as *A Survey of Pipeline Tools for Data Engineering* [35] and *Data Pipeline Management in Practice* [36] focus on general data engineering challenges, they help define the broader system context in which Redis-like systems operate. Their insights into integration, automation, monitoring, and scalability highlight demands that Redis-like systems should meet, making it essential to evaluate them under realistic and operationally relevant conditions.

Benchmarking plays a central role in evaluating the performance of a system. *Benchmarking Cloud Serving Systems with YCSB* [37] introduced a widely adopted framework for assessing NoSQL databases using standardised workloads. It emphasises the need for workload consistency, such as controlling read and write ratios and access patterns. It has become the de facto standard for benchmarking NoSQL systems, especially across different NoSQL categories. However, as discussed in *On the State of NoSQL Benchmarks* [38], *YCSB* follows a general-purpose approach focusing primarily on generic workloads. While this broad applicability is useful, it often comes at the cost of overlooking system-specific features, specialised capabilities, and intended use cases. This underscores the need for more targeted benchmarking approaches when evaluating systems designed for particular operational roles, such as Redis and its successors. In this direction, *Bench-Ranking* [39] argued that benchmarking should go beyond purely reporting performance metrics, and instead aim to support context-aware evaluation that helps guide real-world decision-making.

Real-world workload studies further reinforce the need for system-aware benchmarking. *Workload Analysis of a Large-Scale Key-Value Store* [40] and *A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter* [4] provide data from production environments, revealing significant differences across deployments, including variations in read and write ratios, access patterns, and object sizes. The studies show that different workloads lead to various performance outcomes, and that no single workload model can capture the full range of real-world behaviour. Therefore, benchmarks should be tailored to reflect the specific role and use cases when evaluating systems.

Redis has frequently been included in academic benchmarking studies, often compared with other NoSQL systems such as MongoDB, Cassandra, and HBase. Tools like *YCSB* and *memtier_benchmark* are commonly used to evaluate their performance. For instance, Kausar et al. [41] evaluate MongoDB, Cassandra, and Redis under various *YCSB* workloads over different data sizes. Ben Seghier et al. [42] focus on execution time comparisons between the different *YCSB* workloads. Alzaidi and Vagner [43] investigate concurrency performance in Redis and HBase using *YCSB*. Although these studies provide valuable performance insights, they tend to treat Redis as a generic NoSQL system rather than accounting for its unique architecture and the specific use cases it is typically deployed for.

Complementary architectural insights are provided in *Evaluation and Analysis of In-Memory Key-Value Systems* [44], which compares Redis and Memcached to demonstrate how internal design choices, such as data structures, memory usage, and persistence models, impact performance. The study shows that even minor system-level architectural decisions can significantly affect throughput and resource efficiency. In contrast, *Big Data Resource Management & Networks* [45] highlights how external factors, such as infrastructure architecture and configuration, can influence system performance and behaviour. Together, these perspectives underline the importance of evaluating systems with awareness of their internal design and their deployed environments.

Several industry white papers [46, 47, 48] have evaluated Redis under production-like conditions, offering practical insights. Although these reports often lack methodological transparency, they highlight real-world performance insights that complement academic benchmarking.

Although NoSQL and in-memory databases have been widely benchmarked, much of this research consists of genetic benchmarking across different NoSQL families and often is solely focused on performance metrics. However, the literature highlights the need for system-aware benchmarking that reflects the system's typical real-world scenarios. This is especially important for Redis-compatible systems like Valkey, KeyDB, and Garnet, which aim to be replacements. Despite their growing relevance, particularly after Redis's licensing change, these alternatives remain underexplored in academic research. Broader factors such as governance models, development activity, community involvement, and migration complexity are also often overlooked in the literature. These influence how well Redis-compatible systems can replace Redis in existing infrastructures.

This study addresses that gap by conducting a focused evaluation of Valkey, KeyDB, and Garnet. These systems are benchmarked under realistic Redis-like workloads, with attention to migration complexity, ecosystem compatibility, and long-term sustainability. By focusing on Redis's operational context, this work aims to provide actionable insight for developers and architects who are evaluating alternatives within the in-memory key-value store landscape [49].

4 Methodology

This chapter outlines the methodology used to systematically evaluate Redis successors ValKey, KeyDB, and Garnet, with Redis as a baseline. Comparative benchmarking is the core of this methodology: The performance of each system was measured under the same environment and conditions and repeated multiple times to ensure statistical confidence.

This methodology aims to identify how each system performs under typical Redis use cases, based on different metrics of interest, and to pinpoint which systems are best suited for various scenarios.

4.1 Test Environment

All systems was installed in a Kubernetes-based environment to replicate a realistic deployment scenario. The configuration of each system was kept identical. Each system is deployed as a standalone instance, with one running process per system, isolating performance variables.

For the Kubernetes environment, Kubeadm [50] was deployed and installed on a virtual machine. Since each system will be deployed as a standalone, the Kubernetes environment was configured with a single node acting as the control plane and worker. The benchmark client runs on a separate virtual machine, hosted on the same physical server, and communicates with the system under benchmarks through NodePort over an internal network. This setup simulates a distributed, cloud-native architecture while remaining controlled and reproducible.

Table 1 provides a detailed overview of the configurations of the test environments.

Table 1 Specification of the Kubernetes and benchmarking environment.

Kubernetes Environment	
Distribution	Kubeadm
Kubernetes Version	v1.31.5
Kubeadm Version	v1.31.5
Operating System	Ubuntu 24.04 LTS
Resources	4 vCPUs, 12 GiB RAM
Container runtime	Containerd
CNI-plugin	Calico
Service Exposure	NodePort
Host Environment	Same physical server as the benchmarking VM
Benchmarking Environment	
Machine Type	Virtual Machine
Operating System	Ubuntu 24.04 LTS
Resources	10 vCPUs, 16 GiB RAM
Host Environment	Same physical server as the Kubernetes VM

4.2 Configuration of Redis and Its Alternatives

All systems were deployed in Kubernetes using Helm [29]. To ensure consistency and reproducibility, each system was installed using an official or community-maintained Helm chart: Redis via Bitnami's chart [51], Valkey via Bitnami's Valkey chart [52], KeyDB via Enapter's Helm chart [53], and Garnet using Microsoft's official Helm chart [54]. Each deployment was configured through a YAML file specifying resource constraints, networking parameters, and service exposure. Table 2 summarises these settings, with complete configurations included in Appendix A.

Table 2 The configurations of the YAML files used for Redis, Valkey, KeyDB, and Garnet.

Configuration Summary	
CPU Requests	1 core
CPU Limits	2 cores
Memory Requests	6 GiB
Memory Limits	8 GiB
Persistence	Disabled
Service Type	NodePort

Each system was deployed in a standalone mode, with a single instance per deployment.

This design was chosen to isolate performance characteristics and eliminate variability introduced by clustering, replication, or high-availability setups. The configurations follow Redis' official production recommendations [55], ensuring realistic deployment standards and enabling a fair baseline for comparative benchmarking.

Furthermore, according to Redis' hardware guidelines [55], no more than 70% of the total RAM should be allocated to Redis itself, with the remaining 30% remaining for the operating system and background processes. Additionally, Redis recommends using no more than 80% of available CPU to maintain system stability under load. Based on these guidelines, the memory limit was set to 8 GiB and the CPU limit to 2 cores.

All systems were deployed using the same configuration parameters to ensure a fair and meaningful comparison. Since this study aims to evaluate potential Redis replacements in a production-like environment, a consistent setup across all candidates was employed to assess their performance under equivalent conditions.

The specific versions of software used during the benchmarking for each system are presented in Table 3.

Table 3 Software stack and version details for the evaluated systems.

Software Stack	
Helm	v3.17.1
Redis	v7.2.7
Valkey	v8.0.2
KeyDB	v6.3.4
Garnet	v1.0.62

4.3 Workload and Benchmarking

This study aims to evaluate potential Redis successors by measuring different metrics under workloads that reflect typical Redis usage patterns. To conduct the benchmarks, the widely recognised utility `memtier_benchmark` [56] was employed. Developed and maintained by Redis Inc., `memtier_benchmark` enables the generation of high-throughput, configurable load against in-memory key-value stores. It supports precise control over workload characteristics, including read/write ratios and the number of threads, making it well-suited for simulating real-world Redis use cases.

`memtier_benchmark` has seen widespread adoption in industry and academic settings. It is recommended by Redis [57] and several Redis-compatible systems, including KeyDB [58] and Valkey [59], and has been used in other studies [48, 47].

Yahoo! Cloud Serving Benchmark (`YCSB`) [60] was initially considered, as it is a well-established and widely adopted framework for benchmarking NoSQL databases [61]. However, `YCSB` was initially designed for disk-based systems such as Cassandra and HBase [62], and has been reported to scale poorly under high-concurrency workloads [63]. Given the Redis-centric scope of this study and that all evaluated systems are in-memory and potential Redis replacements where concurrency and low latency are key performance metrics,

it was concluded that `YCSB` was deemed less suitable. Instead, `memtier_benchmark`, being purpose-built for Redis, was selected as the most appropriate tool. Furthermore, it was explicitly designed to handle Redis-relevant workloads and is the preferred choice among most systems considered in this study.

This methodology used a fork of `memtier_benchmark` as the official version does not natively support Zipfian key-access distributions. This modification was necessary to replicate the skewed access patterns observed in real-world Redis-like deployments, discussed further in Section 4.3.1.

4.3.1 Key Access Pattern

A Zipfian key-access distribution was used to simulate realistic production traffic. This distribution reflects typical use cases for Redis, which means that a small number of keys, often referred to as ”hot keys”, receive a disproportionately large number of accesses. It is commonly used when benchmarking Redis and similar systems [64]. Furthermore, YCSB also uses a Zipfian in all its preset workloads [65].

A large-scale study by X (formerly known as Twitter) [4] analysed hundreds of in-memory cache clusters and concluded that key-access patterns frequently follow a Zipfian distribution. They concluded that the mean of the Zipfian skew parameter α for heavy-read and -write workloads:

$$\alpha = \begin{cases} 1.4, & \text{for read-heavy workloads} \\ 0.9, & \text{for write-heavy workloads} \end{cases}$$

This methodology uses these α values to reflect realistic Redis-like access patterns.

4.3.2 Data Size and Memory Pressure

Each stored value was set to 1 KiB in size, which is commonly used in similar benchmarks [62]. YCSB also uses a default key-value size of 1 KiB in all its preset workloads [65]. The systems were pre-loaded with data before running any tests to make the benchmark more accurate and reflect realistic scenarios.

Microsoft recommends loading the system to about 75% of its available memory before benchmarking [66]. In this setup, each system had access to 8 GiB of RAM (see Section 4.2). Each key was 1,024 bytes in size, and an average overhead of 300 bytes was assumed. The number of keys was calculated as shown in Equation 4.1.

$$\text{number of keys} = \frac{(8 \times 2^{30} \times 0.75)}{1024 + 300} \approx 4{,}800{,}000 \tag{4.1}$$

This setup ensures that the systems are under realistic memory pressure during benchmarking, rather than running with mostly empty memory.

4.3.3 Workload Characteristics

An essential part of any benchmarking is selecting representative workloads that reflect real-world usage. The following subsection outlines how the workloads were defined, what

parameters were considered when shaping them, and how they aim to capture both read-heavy and write-heavy scenarios relevant to typical Redis deployments.

Workload Types and Read/Write Ratios

Two workloads were selected to evaluate each system's performance under realistic conditions: a write-heavy 50% read / 50% write workload and a read-heavy 95% read / 5% write workload. This selection follows a common convention in benchmarking Redis and similar in-memory systems, where at least one write-intensive and one read-intensive scenario is included to cover a wide spectrum of operational demands [64, 46, 67].

This approach is further motivated by a large-scale study conducted by X (formerly Twitter), which analysed hundreds of in-memory cache clusters [4]. The study found that most cache workloads in production are strongly read-heavy, with average read ratios around 90%. However, it also revealed that approximately 35% of clusters qualify as write-heavy, defined as having more than 30% write operations.

The following two workloads will be used in this study:

Workload A (50% reads / 50% writes) is a write-heavy workload that simulates environments where data is frequently updated and read, such as session management, collaborative editing, or stateful service caches.

Workload B (95% reads / 5% writes) represents the more common read-heavy usage of Redis, such as classic caching layers for web applications or microservices, where the cache is primarily used for fast data retrieval.

These workload configurations ensure a realistic and comprehensive evaluation of each system's performance under different read/write pressures and access distributions. The full benchmarks are provided in Table 4.

Table 4 Workload characteristics used in the benchmarking tests.

Workload Configuration Summary	
Workload A	50% reads / 50% writes (write-heavy)
Workload B	95% reads / 5% writes (read-heavy)
Access Pattern	Zipfian distribution
Zipfian α value (A)	0.9
Zipfian α value (B)	1.4
Key Size	1 KiB
Number of Keys	4.8 million
Warm-Up Duration	60 seconds
Test Duration	5 minutes per run
Repetitions per Test	3

Execution Parameters and Benchmarking Assumptions

Each test was repeated three times, and the mean values from these runs were used for analysis, ensuring statistical confidence.

Each workload was executed across increasing levels of concurrency, where each level represents the number of simultaneous client connections. Concurrency is a key factor in benchmarking in-memory systems and is commonly used to measure scalability and saturation points. Pipelining was deliberately disabled, as typical cache usage involves one request per connection at a time, waiting for a response before the next. This behaviour aligns with how `memtier_benchmark` operates by default and better reflects real-world Redis usage [68, 34].

Concurrency Levels and Infrastructure Limits

The different concurrency levels were determined through a small preliminary test to identify the key stages of Redis performance: the point of low concurrency, the point at which throughput rapidly increases, the peak throughput, and finally, the saturation point. Based on these observations, the following concurrency levels were selected: 10, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1 200, 1 500, and 2 000.

Notably, even higher concurrency levels were attempted during the testing phase. However, pushing beyond 2 000 concurrent connections resulted in crashes within the Kubernetes environment, particularly impacting the `etcd` cluster, as indicated by repeated warnings about request timeouts and processing delays. These errors suggested that the control plane was becoming overloaded, making further increases in concurrency unreliable and unrepresentative of application-layer performance. Therefore, 2 000 concurrent connections were chosen as the maximum tested level to ensure the validity and reliability of the benchmarking results without overwhelming the underlying orchestration infrastructure.

4.3.4 Resource Monitoring During Benchmarking

During the benchmarking procedure, a custom resource monitoring script was executed in the background throughout each test. This script periodically samples CPU usage and memory consumption of both the system under test and the Kubernetes node. While the benchmarking tool measured throughput and latency, the monitoring script provided additional insight into the system's resource efficiency and operational cost under load. The full script is included in Appendix

During the benchmarking procedure, a custom resource monitoring script was executed in the background throughout each test. This script periodically samples CPU usage and memory consumption every 10 seconds for both the system under test and the Kubernetes node. For each three minute benchmark run, the collected data was aggregated by calculating the average CPU and memory usage, providing a representative view of sustained resource utilisation. While the benchmarking tool measured throughput and latency, the monitoring script offered complementary insights into how efficiently each system used its resources under load. The full script is included in Appendix B.

4.3.5 Warm-Up Phase

A 60-second warm-up phase preceded each benchmark to ensure consistent and reliable performance measurements. During this period, the system under test was subjected to the same workload configuration as the actual benchmark, but all results were discarded. This warm-up period allowed internal components to stabilise before data collection began. Eliminating the effects of cold-start behaviour, with increased latency and reduced throughput [69]. Similar warm-up phases are commonly employed in other in-memory benchmarking studies [46].

4.3.6 Benchmarking Procedure

A shell script (see Appendix B) was utilised to automate the benchmarking process to ensure consistency and reproducibility. The script executes all workload configurations across various concurrency levels, handles warm-up and data collection, and logs structured results to CSV files for analysis. Each system (Redis, ValKey, KeyDB, Garnet) was benchmarked separately by modifying the script's connection parameters before execution.

The entire benchmarking process is summarised in Algorithm 1, which outlines the structured sequence used for each system.

Algorithm 1 Benchmark Execution Loop (per system).

Manual Step: Deploy target system (e.g., Redis, ValKey) to Kubernetes using Helm
Preload $\approx$ 4.8M keys into the system
for workload $\in$ {Workload A (50/50), Workload B (95/5)} **do**
    for concurrency $\in$ {10, 100, 200, ..., 2000} **do**
        Warm-up Phase: run workload for 60 seconds (discard results)
        for repetition = 1 to 3 **do**
            Run `memtier_benchmark` for 3 minutes
            Record throughput and latency metrics
            Sample CPU and memory usage
        end for
    end for
end for
Manual Step: Delete Helm release and clean up Kubernetes resources

Each benchmark followed the structured sequence shown above. Preloading, warm-up, and multiple repetitions were used to ensure fair and stable system comparisons. Approximately 4.8 million key-value pairs were inserted before testing, filling 75% of the allocated memory to create realistic memory pressure (see Section 4.3.2). The 60-second warm-up phase helped eliminate cold-start effects (see Section 4.3.5). Tests were repeated three times for each concurrency level and workload, and the results were averaged to ensure statistical reliability (see Section 4.3.3).

4.3.7 Benchmarking Limitations

While this benchmark setup for this study aims to reflect realistic Redis-like deployments, several limitations may influence how the results should be interpreted. These constraints

are primarily related to resource allocation, deployment configuration, the architectural nature of the systems, and reliance on default settings provided by Helm charts.

Notably, the benchmarking client machine was provisioned with 10 vCPUs and 16 GiB RAM. No signs of CPU or memory saturation were observed on the client during testing, and resource usage remained consistently low. This ensured that all observed bottlenecks originated from the system under test and not from the client.

Limited CPU Allocation

All evaluated systems, including Redis, were deployed with a CPU limit of two cores, as specified in Section 4.2. This decision was based on Redis's own production recommendations. While this limit may be appropriate for single-threaded systems such as Redis and Valkey, it can underrepresent the performance potential of multithreaded systems like Garnet and KeyDB, which are designed to leverage multiple CPU cores. Although applying the same CPU constraint ensures fairness in comparison, it may constrain systems optimised for parallel execution, particularly at higher concurrency levels.

Standalone Deployment

To isolate performance characteristics and ensure a controlled benchmarking environment, each system was deployed in standalone mode with a single instance and no replication or clustering, as described in Section 4.2. While this decision eliminates variables introduced by a cluster distribution, it also limits the generalisability of the results. Many production deployments of Redis rely on sharding, high availability, and clustering. As such, the results presented in this study may not fully capture how the evaluated systems behave in more complex distributed setups.

Furthermore, while multithreaded systems such as KeyDB and Garnet are designed to scale vertically by utilising additional CPU cores through threading, single-threaded systems like Redis and Valkey typically scale horizontally by deploying multiple instances through sharding. In a standalone benchmarking setup, where only one instance is deployed per system, multithreaded systems may gain a relative advantage by better utilising available CPU resources. For single-threaded systems, achieving similar scaling would require a deployment with multiple instances, which was not considered in this evaluation. As a result, this configuration may unintentionally favour multithreaded systems and underrepresent the potential of single-threaded systems in clustered environments.

Potential Bias from Helm Chart Defaults

All systems were deployed using Helm charts, as specified in Section 4.2. While CPU and memory limits were manually set, many other values were left to chart defaults. Since these defaults can vary between charts and are not always transparent, they may introduce unintended biases. This means that some performance differences observed in this evaluation could partly be a result of differences in default settings rather than the systems themselves.

Resource Metrics Sampling and Aggregation

To assess each system's resource efficiency during benchmarking, CPU and memory usage were periodically sampled using a custom monitoring script. For each test run, these samples were collected every 10 seconds. The data were then aggregated by calculating the average CPU and memory usage over the full duration of each three minute benchmark run. This averaging approach captures sustained resource utilisation rather than transient peaks, making it suitable for comparative analysis of steady-state efficiency. However, it may obscure short-lived spikes or saturation points that could indicate instability or impact overall efficiency and performance.

4.4 Evaluating Benchmark Performance

After data collection, each system's performance across several key metrics was analysed. The primary metrics of interest were:

Throughput. The total number of operations the system can handle per second at each tested concurrency level. This gives a direct measure of overall system capacity under load.

Latency. Measured as the 99^{th} and 99.9^{th} percentiles to capture tail latency, which reflects the worst-case response times users may experience. This is more relevant than average or median latency, where even rare slowdowns can significantly impact overall performance [70, 71].

Resource Usage. The CPU usage (in cores) and memory usage (as a percentage of allocated limits) during each test. These values indicate the resource efficiency of each system under stress.

Scalability. How throughput and latency evolve as concurrency increases, revealing when a system saturates and how gracefully (or not) it handles increasing load beyond that point.

With these definitions and the methodology established, Chapter 5 presents the performance results of each evaluated system. The results provide a detailed comparison of Redis and its potential successors under realistic conditions by examining how throughput, latency, and resource usage evolve under different concurrency levels and workload types.

4.5 Evaluating Migration Effort and Redis Compatibility

When migrating from Redis to an alternative system, the overall migration effort becomes a critical consideration. While some candidates function as near drop-in replacements, others may require configuration changes, code adaptations, or even substantial architectural adjustments. Beyond API compatibility, it is equally important to assess how well each system integrates into Kubernetes-based environments, how straightforward the deployment process is, and whether standard Redis-related tools, such as client libraries, command-line interfaces, and observability stacks, remain compatible.

To assess the migration complexity for each alternative, the following criteria are considered:

API Compatibility. Redis uses a wire protocol called Redis Serialisation Protocol (RESP) [72] for client-server communication, which encodes commands and responses as a text-based structure. The evaluation considers whether Redis alternatives support RESP, and to what extent. Also, compatibility with Redis command semantics and supported data types, such as integers, strings, arrays, and sets, should be considered. Partial or no RESP support may require replacing client libraries, adapting APIs, or modifying application logic, which increases migration complexity.

Configuration and Deployment. This criterion considers how easily the alternative system can be installed, configured, and deployed in a cloud-native environment. The evaluation includes the availability of container images and Helm charts. Significant differences in deployment tooling or infrastructure requirements compared to Redis may increase migration effort.

Operational Integration. This criterion evaluates how well the alternative systems integrate with existing operational tooling and workflows. The assessment includes support for observability through Prometheus and Grafana, compatibility with command-line tools such as `redis-cli` or equivalents, and runtime backup and restore mechanisms (e.g., snapshotting or persistence). Systems that require significant changes to monitoring, alerting, or maintenance routines compared to Redis are considered to have higher operational migration complexity.

The following Chapter 5 applies these criteria to each evaluated system to assess how well they align with Redis regarding compatibility, deployment practices, and operational integration. By analysing these dimensions, insights regarding the practical challenges and feasibility of replacing Redis with Valkey, KeyDB, or Garnet in production environments can be gained.

4.6 Evaluating Long-Term Viability

In addition to performance and migration effort, long-term viability is essential when selecting a replacement for Redis. A system may be suitable and perform well today, but without ongoing development, transparent governance, and a clear open-source license, that might not remain true in the future. These factors are critical to ensure the system stays maintained, legally usable, and compatible with modern cloud-native environments. That is why it is important to evaluate each system to determine if it is positioned to meet these requirements long-term.

This study will assess each alternative's long-term viability with a focus on its licensing model, ownership, control, development activity and community support. A set of open-source health metrics will be used to assist in the evaluation of the development activity and community support. These metrics are adapted from the CHAOSS [73] (Community Health Analytics in Open Source Software) project, an initiative under the Linux Foundation that defines metrics to evaluate the sustainability and health of open-source projects. While

CHAOSS provides a wide range of metrics, only a handful were selected for this study. These include release frequency, code activity, contributor count, and issue resolution time.

The following will be considered to assess long-term viability:

Development Activity. Capturing how actively a project is maintained and developed by combining two CHAOSS metrics: *Release Frequency*[74] and *Code Changes Commits*[75]. Frequent stable releases and a high number of commits over the past 12 months indicate ongoing development, responsiveness to bugs and feature requests, and timely delivery of updates. In contrast, infrequent releases or low commit activity may suggest stagnation or reduced project support.

Community Support. Evaluates the health of the developer community by combining two CHAOSS metrics: *Committers* [76] and *Issues Closed* [77]. A higher number of unique committers over the past 12 months indicates broader community involvement and reduces reliance on a few individuals, contributing to the project's resilience. The number of closed issues during the same period reflects how engaged the community is in actively responding to issues. Together, these metrics offer insight into how engaged, responsive, and sustainable the project's contributor base is.

Governance and Ownership. Each project's ownership and control will be examined to determine who maintains it, such as a community, a foundation, or a single company. The goal is to understand whether the project is independently governed or controlled by a company, as this can affect long-term sustainability and openness. Projects with company control may be more prone to sudden licensing changes or shifts in direction, while foundation or community governance may offer greater stability and transparency [8].

License Model. Each system's overall licensing model will be examined and discussed, focusing on what the license allows in terms of usage, modification, and redistribution. The goal is to determine whether the system is fully open-source and whether the license supports long-term use in both community-driven and commercial deployments.

With these dimensions defined, Chapter 5 evaluates each system, Valkey, KeyDB, and Garnet, to assess how well they align with the criteria for long-term viability.

5 Results and Discussion

This chapter presents a detailed analysis of the benchmarking results for Redis and its successors, Valkey, KeyDB, and Garnet, under both write-heavy and read-heavy workloads. The evaluation focuses on throughput, latency, and resource efficiency across varying concurrency levels to assess how each system performs under load. These results are based on the benchmarking methodology (Chapter 4, particularly Section 4.4) and the performance metrics defined therein.

5.1 Throughput under Write-Heavy and Read-Heavy Workloads

Throughput performance is analysed under varying concurrency levels. Throughput, measured in operations per second, is used to evaluate how well each system handles increasing client load. Results are presented separately for two types of workloads: a write-heavy workload, which consists of 50% reads and 50% writes, and a read-heavy workload, consisting of 95% reads and 5% writes. These configurations were selected to reflect common real-world usage patterns, further explained in Section 4.3.

Throughout this chapter, several tables present percentage differences relative to Redis. For each system and concurrency level, the percentage indicates how that system performed compared to Redis for the corresponding metric. A positive value means better performance than Redis, while a negative value suggests worse performance than Redis.

5.1.1 Write-Heavy Workload

Figure 1 shows the trends in throughput for a write-heavy workload in increasing numbers of concurrent connections. The percentage differences relative to Redis are provided in Table 5.

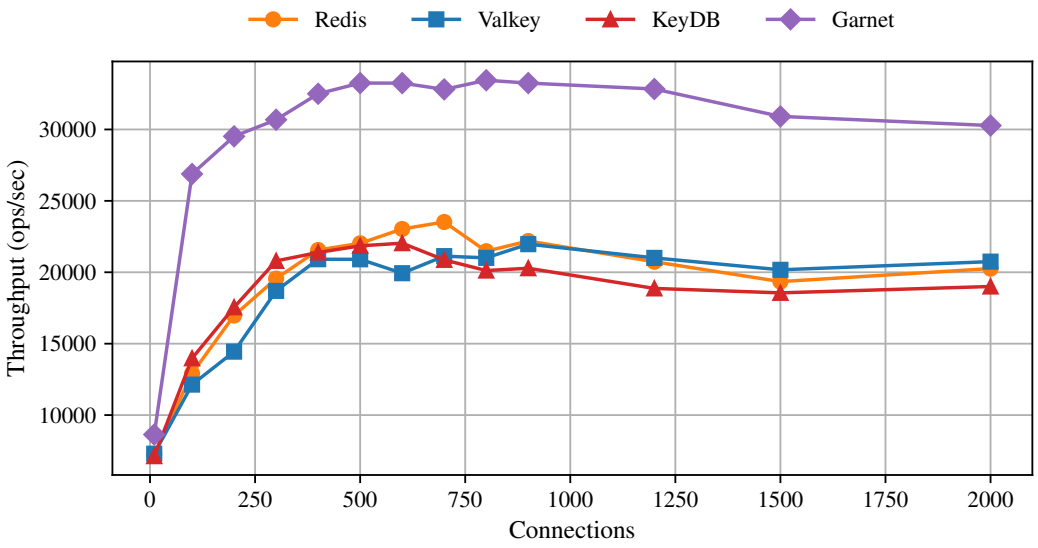

Figure 1: Throughput for write-heavy workload as a function of concurrent connections.

Table 5 Percentage difference in throughput compared to Redis for write-heavy workload.

Connections	Valkey vs Redis (%)	KeyDB vs Redis (%)	Garnet vs Redis (%)
10	+1	-2	+19
100	-6	+8	+108
200	-15	+3	+74
300	-4	+6	+57
400	-3	-1	+51
500	-5	-1	+51
600	-13	-4	+44
700	-10	-11	+39
800	-2	-6	+56
900	-1	-9	+50
1200	+1	-9	+58
1500	+4	-4	+60
2000	+2	-6	+49

All systems demonstrated increasing throughput as concurrency increased, until reaching a saturation point between 600 and 800 connections, beyond which performance began to level off or decline.

Redis and Valkey followed similar throughput trends, with Redis maintaining a clear advantage in most of the range. Between 100 and 700 connections, Redis consistently outperformed Valkey, peaking at a 15% lead. At very high concurrency levels, 1 200 and above, Valkey managed to slightly surpass Redis, indicating a recovery of some efficiency under extreme concurrency.

KeyDB initially matched Redis in throughput, showing a slight advantage at low concurrency. However, this gain was short-lived. From 400 connections onward, KeyDB con-

sistently performed worse, with throughput falling up to 11% at higher levels. Despite its multithreaded design, KeyDB did not deliver improved throughput under write-heavy conditions and failed to scale effectively.

Garnet stood out across the board, delivering the highest throughput at every concurrency level. At 100 connections, it achieved more than double the throughput of Redis and maintained significant margins throughout. On average, it sustained nearly 55% higher throughput than Redis across the entire test range. These results highlight Garnet's scalability and underline the efficiency of its multithreaded architecture under parallel write-intensive workloads.

5.1.2 Read-Heavy Workload

Figure 2 shows the throughput results for the read-heavy workload across increasing concurrency levels. Table 6 provides the corresponding percentage differences in throughput related to Redis.

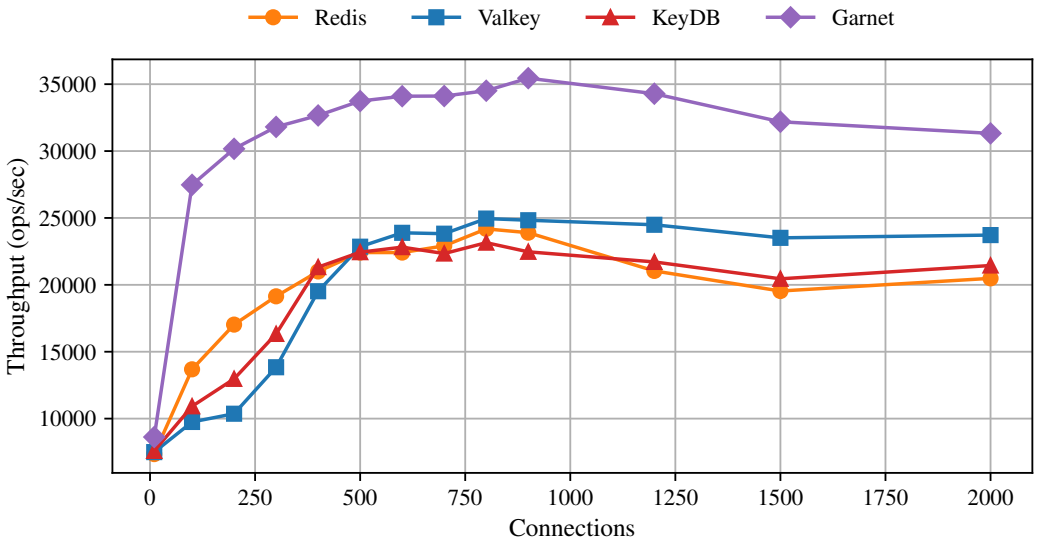

Figure 2: Throughput (ops/sec) for read-heavy workload as a function of concurrent connections.

Table 6 Percentage difference in throughput compared to Redis for read-heavy workload.

Connections	Valkey vs Redis (%)	KeyDB vs Redis (%)	Garnet vs Redis (%)
10	+2	+3	+17
100	-29	-20	+101
200	-39	-24	+77
300	-28	-15	+66
400	-7	+2	+56
500	+2	0	+51
600	+7	+2	+52
700	+4	-3	+49
800	+3	-4	+43
900	+4	-6	+48
1200	+16	+3	+63
1500	+20	+5	+65
2000	+16	+5	+53

All systems showed increasing throughput with increasing concurrency, up to a saturation point around 800 to 900 connections, beyond which performance plateaued or declined.

Redis outperformed Valkey at low to moderate concurrency levels, particularly between 100 and 500 connections. However, Valkey scaled more effectively at higher loads and consistently exceeded Redis from 500 connections onward, reaching up to 20% higher throughput at 1 500 connections. These results suggest that, while Redis handles light traffic better, Valkey becomes more efficient as demand increases.

KeyDB remained below Redis across nearly all concurrency levels, despite slightly better performance than Valkey at the lower end. It peaked at a 24% deficit compared to Redis at 200 connections and did not show a substantial advantage even as concurrency increased. This indicates that KeyDB's multithreaded architecture does not translate into improved throughput in this read-heavy scenarios.

Garnet consistently delivered the highest throughput across the full range. At 100 connections, it already outperformed Redis by over 100%, and it maintained an average improvement of roughly 57% throughout. This consistent lead underlines the efficiency of Garnet's multithreaded design and its suitability for high-throughput, read-intensive workloads.

5.1.3 Summary

Garnet consistently achieved the highest throughput across all concurrency levels and under both workloads, maintaining large performance margins over Redis. Valkey underperformed compared to Redis at low to moderate concurrency, especially in the read-heavy workload, but surpassed it at high concurrency levels, indicating better scalability. KeyDB followed a similar trend to Redis but generally underperformed across both workloads and did not demonstrate any meaningful advantage.

5.2 Tail Latency Performance

This section analyses the tail latency behaviour of Redis and its successors under varying concurrency levels, with a focus on the 99th (p99) and 99.9th (p99.9) percentiles. These metrics capture the slowest response times observed and are essential for evaluating worst-case performance. While throughput reflects overall system capacity, it does not reveal how consistently a system can respond to individual requests. High tail latency can degrade user experience even when throughput remains high. Results are reported separately for two workload types: a write-heavy workload consisting of 50% reads and 50% writes, and a read-heavy workload consisting of 95% reads and 5% writes.

5.2.1 Latency Under Write-Heavy Workload

Figures 3 and 4 show the p99 and p99.9 latency trends under increasing concurrency. Detailed percentage differences relative to Redis are provided in Tables 7 and 8.

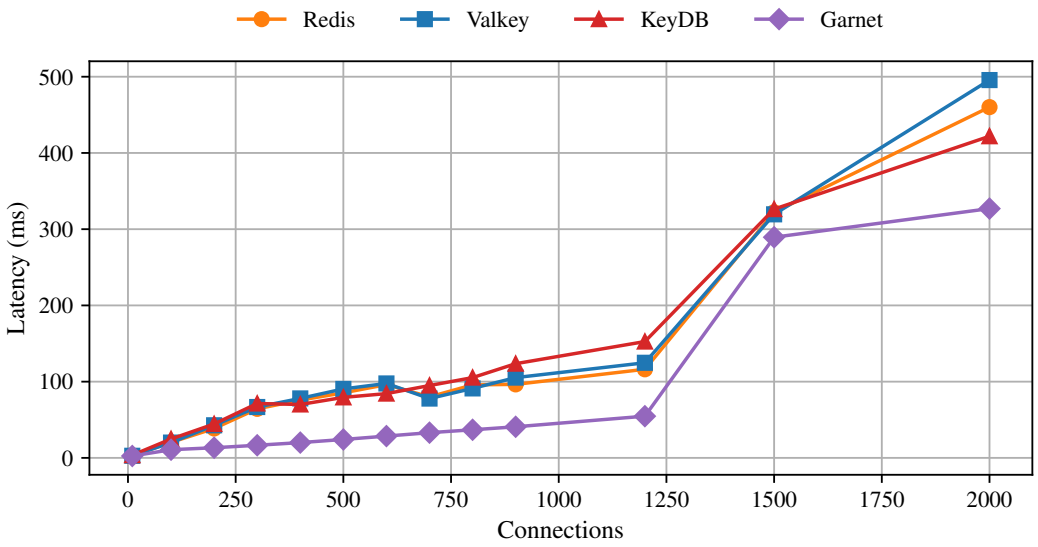

Figure 3: P99 latency vs. connections for the write-heavy workload.

Table 7 Percentage difference in p99 latency compared to Redis for the write-heavy workload.

Connections	Valkey vs Redis (%)	KeyDB vs Redis (%)	Garnet vs Redis (%)
10	-3	+26	-13
100	-1	+22	-47
200	+11	+15	-65
300	+4	+11	-74
400	+3	-8	-74
500	+6	-7	-72
600	+2	-12	-70
700	-3	+19	-59
800	-5	+10	-62
900	+9	+28	-58
1200	+7	+31	-53
1500	-1	+1	-10
2000	+8	-8	-29

All systems showed increasing latency with higher concurrency, with the sharpest rises occurring beyond 900 connections, likely indicating saturation. Valkey remained close to Redis across the range, typically within ±10%, but showed no consistent advantage. While it briefly outperformed Redis at 700 and 800 connections, the improvements were marginal. KeyDB mostly underperformed, with slightly better results at 200 to 600 connections, but higher latency elsewhere, including a peak of 31% above Redis at 1 200 connections. Garnet clearly outperformed all systems, maintaining over 70% lower latency than Redis between 300 and 600 connections. Its linear and predictable growth curve reflects efficient multi-threaded scheduling and strong scalability under write-heavy load.

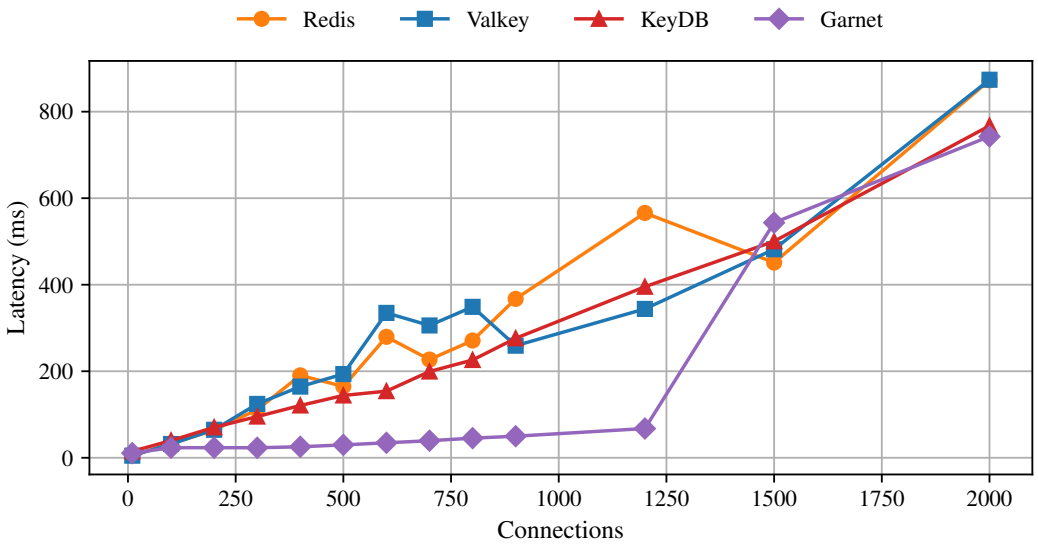

Figure 4: P99.9 latency vs. connections for the write-heavy workload.

Table 8 Percentage difference in p99.9 latency compared to Redis for the write-heavy workload.

Connections	Valkey vs Redis (%)	KeyDB vs Redis (%)	Garnet vs Redis (%)
10	-12	+170	+94
100	-1	+23	-28
200	0	+9	-64
300	+12	-14	-79
400	-14	-37	-87
500	+17	-13	-82
600	+20	-45	-88
700	+35	-12	-83
800	+29	-17	-83
900	-29	-25	-86
1200	-39	-30	-88
1500	+7	+11	+20
2000	0	-12	-15

Redis exhibited fluctuating p99.9 latency, with no clear scaling trend. Valkey mostly fell short of Redis up to 800 connections and only outperformed it at 900 and 1 200 connections. KeyDB showed very poor performance at low concurrency (169.91% higher latency at 10 connections), but otherwise consistently maintained lower p99.9 latency than Redis. Notably, its latency scaled near-linearly without large fluctuations, outperforming Redis from 300 connections onward. Garnet performed exceptionally well up to 1 200 connections, with low and stable latency and a nearly flat growth curve compared to other systems. At the lowest concurrency level (10 connections), however, its latency was nearly twice as high as Redis, though this discrepancy disappeared immediately as load increased. Garnet peaked sharply at 1 500 connections, recording the highest latency of all systems at that level, before recovering again at 2 000 connections.

5.2.2 Latency Under Read-Heavy Workload

Figures 3 and 4 show the p99 and p99.9 latency trends under increasing concurrency. Detailed percentage differences relative to Redis are provided in Tables 9 and 10.

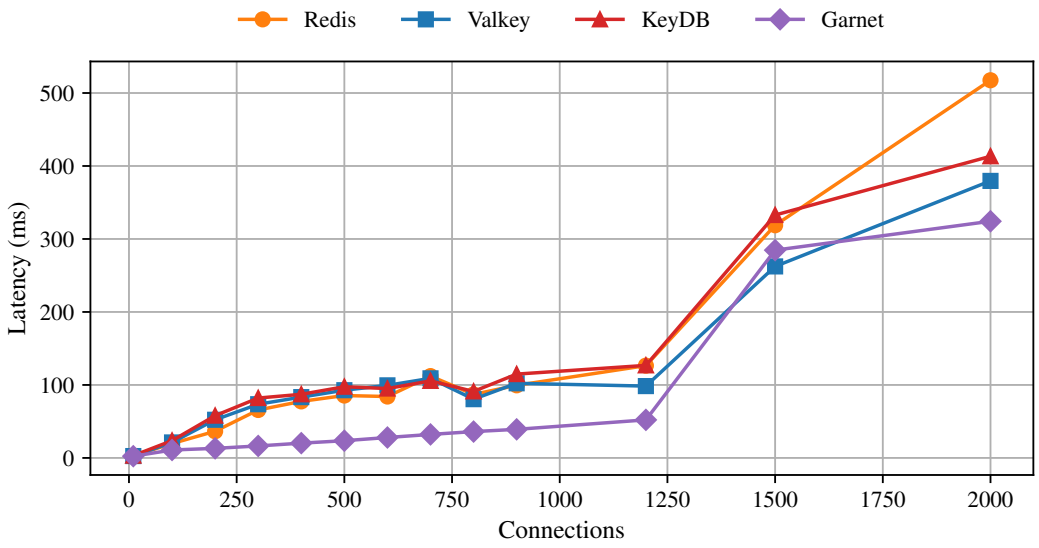

Figure 5: P99 latency vs. connections for the read-heavy workload.

Table 9 Percentage difference in p99 latency compared to Redis for the read-heavy workload.

Connections	Valkey vs Redis (%)	KeyDB vs Redis (%)	Garnet vs Redis (%)
10	-10	+10	-16
100	+8	+20	-45
200	+43	+59	-64
300	+12	+25	-75
400	+8	+13	-74
500	+9	+14	-72
600	+18	+13	-67
700	-3	-6	-71
800	-8	+5	-59
900	+3	+15	-61
1200	-22	+0	-59
1500	-18	+4	-11
2000	-27	-20	-37

All systems exhibited relatively stable p99 latency up to 900 concurrent connections, beyond which latency increased sharply, likely indicating saturation. Valkey underperformed compared to Redis at lower concurrency levels, but demonstrated stronger performance at higher loads. From 700 connections onward, it consistently achieved lower latency than Redis, highlighting its ability to scale under higher concurrency. KeyDB consistently recorded higher latency than Redis across most of the range, only surpassing it at the maximum tested level of 2 000 connections. Garnet outperformed all other systems throughout the test. It maintained a smooth and linear increase in latency up to 900 connections, and continued to lead beyond this point. Although Valkey briefly achieved lower latency at 1 500 connections, Garnet regained the advantage at 2 000, highlighting its efficiency under read-intensive loads.

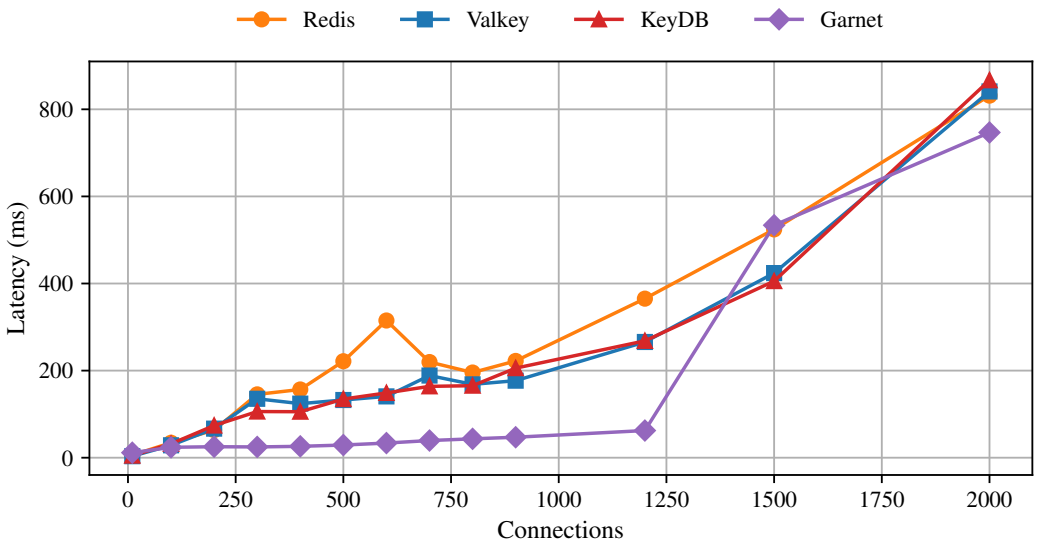

Figure 6: P99 latency vs. connections for the read-heavy workload.

Table 10 Percentage difference in p99.9 latency compared to Redis for the read-heavy workload.

Connections	Valkey vs Redis (%)	KeyDB vs Redis (%)	Garnet vs Redis (%)
10	-40	-16	+99
100	-17	-7	-30
200	+5	+16	-60
300	-7	-27	-83
400	-21	-33	-83
500	-40	-39	-87
600	-55	-53	-89
700	-14	-25	-82
800	-14	-15	-78
900	-20	-7	-79
1200	-27	-27	-83
1500	-19	-23	+2
2000	+1	+4	-10

Redis exhibited the highest p99.9 latency across nearly all concurrency levels, displaying notable fluctuations. In contrast, both Valkey and KeyDB demonstrated more stable and linear latency curves, consistently outperforming Redis. Garnet delivered the strongest overall performance, maintaining the lowest p99.9 latency at nearly all levels. While it briefly underperformed at 10 concurrent connections, recording almost double the latency of Redis, it quickly stabilised and remained dominant up to 1 200 connections. At 1 500 connections, Garnet experienced a temporary spike, resulting in the highest latency among all systems. However, this was short-lived, as it once again achieved the lowest latency at 2 000 connections.

5.2.3 Summary

Garnet achieved the lowest tail latencies under both workloads, with a flat and linear latency curve up to 1 200 connections. Its p99 and p99.9 values remained closely aligned, demonstrating predictable performance. Valkey exhibited slightly higher tail latency than Redis at low to moderate concurrency but consistently outperformed Redis at higher levels, particularly in the read-heavy workload. KeyDB generally showed higher p99 latency than Redis but maintained stable and competitive p99.9 latency, especially at high concurrency.

5.3 Resource Efficiency: CPU and RAM Usage

This section presents the CPU and RAM usage observed during benchmarking. Showcasing how efficiently each system utilises available resources under different load conditions. CPU usage is shown as a percentage of the total available cores, where each system was limited to a maximum of 2 CPUs. RAM usage is presented as a percentage of the 8 GiB memory limit configured for each system.

5.3.1 Write-Heavy Workload

This subsection presents the CPU and RAM usage results from the write-heavy workload. The workload consists of 50% reads and 50% writes, representing a balanced access pattern. CPU usage is shown in Figure 7, and memory consumption is displayed in Figure 8.

CPU Usage

Figure 7 shows the average CPU usage of each system as concurrency increases during the write-heavy workload. It illustrates how many CPU cores are utilised under load, with all systems limited to 2 available cores.

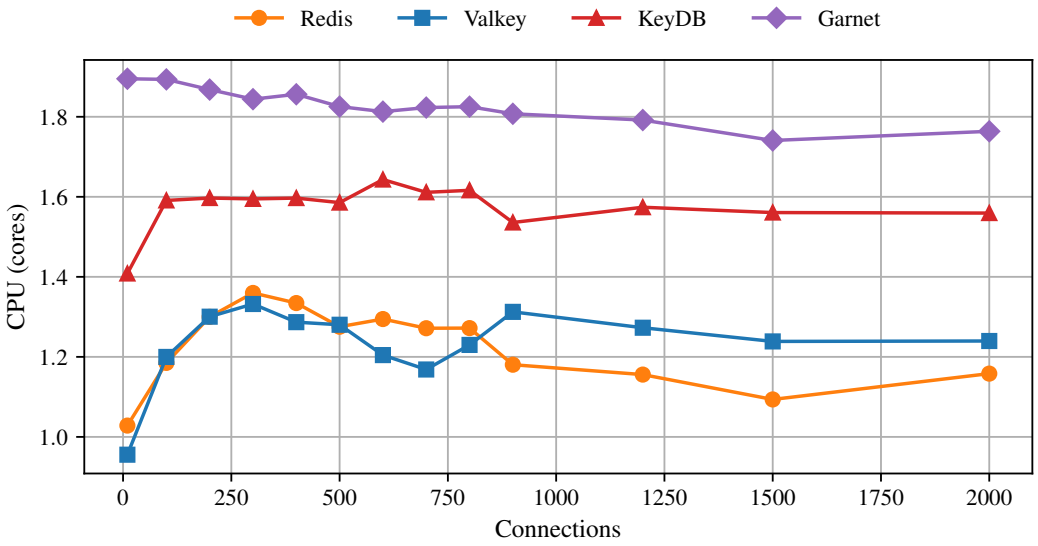

Figure 7: Average CPU usage vs. connections for write-heavy workload.

In general, all systems showed decreased CPU usage as concurrency increased. This likely results from increased context switching, thread contention, and queueing overhead. As the system becomes saturated, more CPU time is spent on coordination rather than executing useful work, reducing overall efficiency. Redis and Valkey, both single-threaded for command execution, plateaued around 1.3 cores and exhibited more fluctuation compared to the other systems. The additional usage beyond one core likely stems from protocol parsing and I/O handling, which utilise multiple threads. KeyDB reached approximately 1.6 cores but did not fully utilise the available CPUs, suggesting limited scalability. Garnet achieved the highest utilisation, peaking at 1.9 cores, indicating efficient multithreaded execution.

RAM Usage

Figure 8 displays the percentage of allocated memory consumed by each system during the write-heavy workload. It shows how memory usage develops as concurrency increases, with all systems limited to 8 GiB of memory.

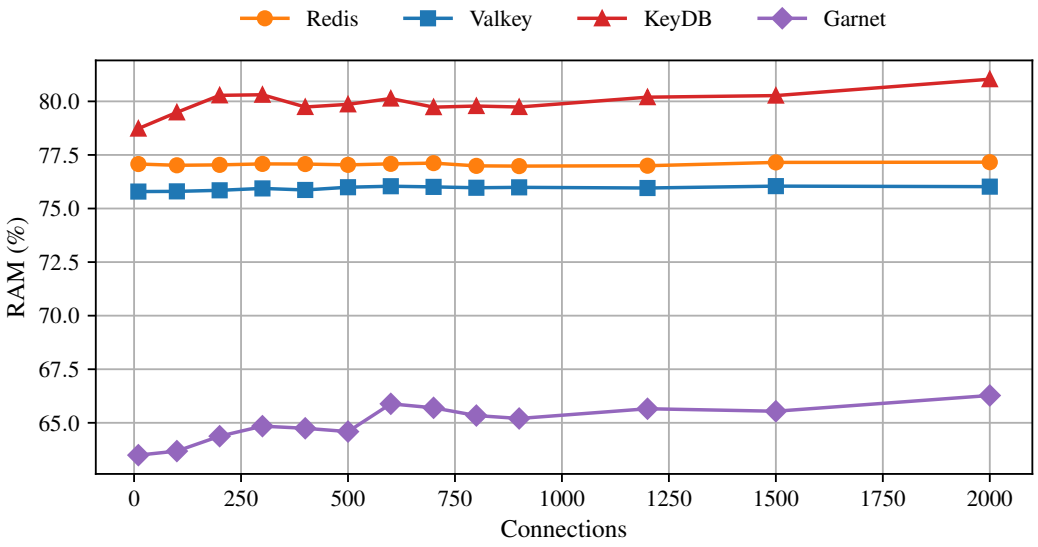

Figure 8: RAM usage vs. connections for write-heavy workload.

Each system was preloaded with 4.8 million key–value pairs, with the aim to fill about 75% of the 8 GiB memory limit (see Section 4.3.2). Since each write was on existing keys, the dataset size remained constant. These results display overhead differences rather than memory growth.

There are clear differences in baseline memory consumption. This indicates that each system handles in-memory data structures with varying efficiency and overhead. Garnet used significantly less memory than the others, pointing to a more compact or efficient internal representation. Redis, Valkey, and KeyDB consumed over 75% of the 8 GiB memory limit, while Garnet remained closer to 65%.

Memory usage stayed mostly stable across all concurrency levels. Redis and Valkey showed minimal to no variation. While KeyDB and Garnet, both multithreaded, exhibited small increases as the concurrency levels rose. This may reflect the additional overhead of manag-

ing parallel execution, such as coordination, caching, or memory allocation across multiple threads.

5.3.2 Read-Heavy Workload

The following subsection examines CPU and memory consumption as concurrency increases during a read-heavy workload. This workload, consisting of 95% reads and 5% writes, reflects common cache-heavy usage patterns where low latency and efficient resource utilisation are critical.

CPU Usage

Figure 9 shows the average CPU usage of each system as concurrency increases during the read-heavy workload. It illustrates how many CPU cores each system utilises under load, with all systems having access to a maximum of 2 cores.

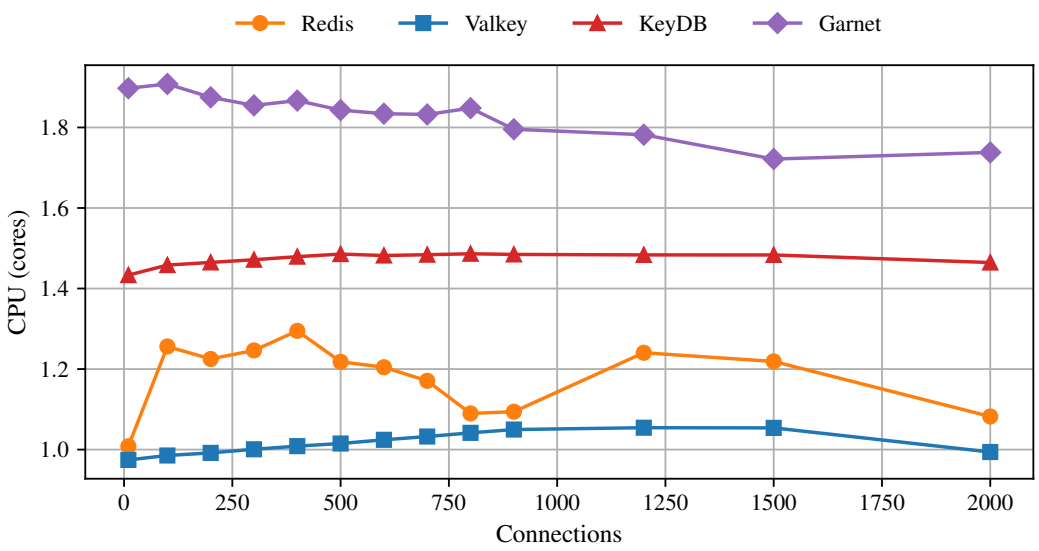

Figure 9: Average CPU usage vs. connections for read-heavy workload.

Under the read-heavy workload, Valkey and KeyDB showed stable and nearly linear CPU usage as concurrency increased, indicating consistent resource utilisation. KeyDB consistently utilised around 1.5 cores throughout the test. Redis, by contrast, exhibited noticeable fluctuations with no clear scaling pattern. Garnet started at approximately 1.9 cores but steadily declined to around 1.7 cores as concurrency increased. This decline likely reflects overhead introduced by high concurrency, including context switching, thread contention, and queuing. As systems become saturated, more CPU time is spent on coordination rather than executing useful work, reducing overall efficiency. Although Redis and Valkey are single-threaded for command execution, their CPU usage exceeded one core, likely due to protocol parsing and I/O handling, which utilise multiple threads.

RAM Usage

Figure 10 displays the percentage of allocated memory consumed by each system during the read-heavy workload. It shows how memory usage evolves as concurrency increases, with all systems limited to 8 GiB of memory.

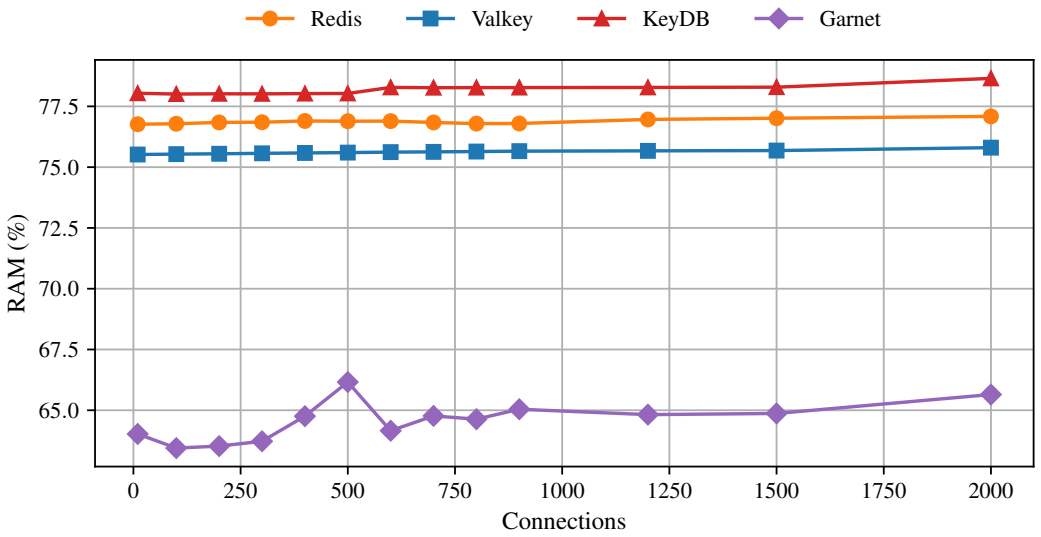

Figure 10: RAM usage vs. connections for read-heavy workload.

Each system was preloaded with 4.8 million key–value pairs, with the aim to fill about 75% of the 8 GiB memory limit (see Section 4.3.2). Since each write was on existing keys, the dataset size remained constant. These results display overhead differences rather than memory growth.

There are clear differences in baseline memory consumption. This indicates that each system handles in-memory data structures with varying efficiency and overhead. Garnet used significantly less memory than the others, pointing to a more compact or efficient internal representation. Redis, Valkey, and KeyDB consumed over 75% of the 8 GiB memory limit, while Garnet remained closer to 65%.

Memory usage stayed mostly stable across all concurrency levels. Redis, KeyDB and Valkey showed minimal to no variation. While Garnet exhibited fluctuation and small increases as the concurrency levels rose. This may reflect the additional overhead of managing parallel execution, such as coordination, caching, or memory allocation across multiple threads. However, interestingly, KeyDB did not share the same pattern for the read-intensive workload, showcasing a stable usage.

5.3.3 Summary

Garnet demonstrated the most efficient use of CPU and memory across workloads. It consistently used more CPU than other systems while maintaining the lowest memory footprint. Valkey and Redis showed similar resource usage, though Valkey had more stable CPU utilisation under read-heavy load. KeyDB showed higher CPU usage than Redis and Valkey but never fully utilised its available resources, and it consistently consumed the most memory.

5.4 Overall Performance Analysis

This section analyses the benchmarking results across the different workloads and performance metrics to identify overarching patterns. By comparing throughput, latency, CPU usage, and memory efficiency under varying concurrency levels, providing a holistic view of each system.

5.4.1 Garnet: Best Performance Across All Metrics

Garnet consistently delivered the highest throughput and lowest latency across both workloads compared to all other systems. In terms of throughput, it outperformed Redis by an average of 55% in the write-heavy workload and 57% in the read-heavy workload. It also achieved near-full CPU utilisation and followed a stable and linear curve, while Redis displayed a more unstable and fluctuating pattern. This indicates efficient multithreaded execution in Garnet's architecture. Although Garnet's memory usage increased slightly under higher concurrency, likely due to context switching, it remained the most memory-efficient overall. Redis, in contrast, showed more stable and nearly constant memory usage across all concurrency levels, but consistently consumed around 77% of its 8 GiB available memory. Garnet, despite being preloaded with the same 4.8 million keys, used only around 65%. This notable difference highlights Garnet's efficient internal data structures and low overhead. Overall, the results suggest that Garnet is well-optimised for parallel workloads and scales effectively under both read-heavy and write-heavy workloads.

5.4.2 Valkey: Similar to Redis, Outperforms It at Scale

Valkey performed similarly to Redis in most tests, reflecting their shared single-threaded architecture, but small differences emerged under specific conditions. Redis had an advantage in write-heavy workloads at low to moderate concurrency, achieving up to 15% higher throughput, while Valkey pulled ahead slightly at high concurrency levels. Redis outperformed Valkey at low concurrency in the read-heavy workloads, achieving up to 40% higher throughput. However, Valkey scaled more effectively as concurrency increased, ultimately surpassing Redis by up to 20% and achieving generally lower latency. CPU usage was similar between Redis and Valkey in the write-heavy tests, though Valkey showed more stable and linear usage under read-heavy conditions. It used slightly less CPU while still achieving higher throughput at high concurrency, indicating more consistent resource utilisation. RAM usage was comparable, with Valkey demonstrating somewhat better efficiency. In conclusion, Redis performed better at low concurrency overall, and particularly in write-heavy workloads. Valkey, on the other hand, performed slightly better under high concurrency in both workloads and clearly outperformed Redis in the read-heavy workload at higher concurrency across all measured metrics.

5.4.3 KeyDB: Promising Design, Underwhelming Execution

KeyDB generally underperformed compared to the other systems, including Redis. Although its multithreaded design allowed for higher CPU usage than both Redis and Valkey, it never fully utilised the two available cores. In terms of throughput, KeyDB did not demonstrate any consistent advantage over Redis. There were only two brief instances where it outperformed Redis, at the lowest concurrency levels in the write-heavy workload and the

highest levels in the read-heavy workload. In both cases, the improvement was marginal, with a maximum gain of 8%, and these gains were short-lived. Overall, Redis consistently outperformed KeyDB across the majority of the test spectrum. It also had the highest memory consumption of all systems, often exceeding 80% of the 8 GiB limit, which points to possible overhead from thread management or internal buffering. In terms of latency, KeyDB offered no clear advantage over Redis, with the exception of p99.9 latency, where it, in general, showed lower latencies with relatively stable and linear scaling. These results suggest that while KeyDB benefits from a multithreaded architecture in theory, it struggles to convert that architectural potential into practical performance gains in this benchmarking setup.

5.4.4 Summary

Garnet emerged as the most performant and scalable system across all scenarios. Valkey followed similar trends to Redis across all metrics, although Redis generally performed better. KeyDB, while promising in design, failed to capitalise on its multithreading potential in practice and performed the worst across all metrics.

5.5 Migration Effort and Redis Compatibility Analysis

In addition to raw performance, the ease of migrating from Redis to an alternative system is a critical factor for adoption. As outlined in Section 4.5 of the Methodology, this section's evaluation considers API compatibility, deployment effort in Kubernetes environments, and operational integration with Redis tooling. These aspects determine how seamlessly Redis can be replaced without disrupting existing workflows, infrastructure, or observability stacks. Given Redis's wide adoption and mature ecosystem, even small incompatibilities can introduce significant migration overhead.

5.5.1 Valkey

This subsection provides an overview of Valkey's compatibility with Redis and the migration effort required.

API Compatibility

Valkey is a direct fork of Redis [78] and is a drop-in replacement [9]. It fully supports the Redis Serialization Protocol (RESP) and maintains the complete Redis command set and data types. As a result, applications typically require no code changes when migrating to Valkey, existing Redis client libraries and command semantics continue to function identically. However, compatibility with newer Redis versions released after its licensing change may not be guaranteed.

Deployment and Configuration

Valkey integrates well with Kubernetes-based and containerised environments. Official Docker images are published at Docker Hub [79], and a Helm chart maintained by Bitnami is available and recommended by the Valkey project [52, 80]. The chart enables straightforward deployment in Kubernetes environments, with a process similar to deploying Redis. While minor adjustments, such as updating image names or chart references, may be required, migrating existing Redis deployments to Valkey requires minimal effort.

Operational Integration

Valkey maintains full compatibility with the Redis ecosystem, allowing standard administration and monitoring tools to function out of the box. Tools such as `redis-cli` are fully supported, and metric exporters like the Redis Prometheus Exporter continue to operate seamlessly by relying on Redis commands such as `INFO`, which Valkey also implements [81]. This ensures that existing observability stacks, including those built with Grafana, require no modification. Regarding runtime backup and restore, Valkey supports the exact persistence mechanisms as Redis, namely RDB and AOF, allowing existing backup scripts and failover strategies to be reused without change [82, 83]. In conclusion, replacing Redis with Valkey does not disrupt operational workflows.

5.5.2 KeyDB

This subsection provides an overview of KeyDB's compatibility with Redis and the migration effort required.

API Compatibility

KeyDB is a multi-threaded fork of Redis that is fully compatible with the Redis API and protocol [13]. It implements the complete set of Redis commands and data types and fully supports RESP [84], allowing any Redis client to connect and execute standard commands without application-level modifications. It also supports the same persistence formats as Redis, including RDB and AOF, which means existing Redis backup files can be restored into KeyDB without modification. Keydb includes some optional, KeyDB-specific features, but these do not interfere with standard Redis usage. However, these additions may not be backwards-compatible with other Redis-compatible systems. In practice, KeyDB functions as a true drop-in replacement for Redis, with full ecosystem compatibility.

Deployment and Configuration

KeyDB integrates well with Kubernetes-based and containerised environments. Official Docker images are available on Docker Hub [85], and community-supported Helm charts, such as those maintained by Bitnami [53], can be used for deployment. The process is similar to deploying Redis. While minor changes, such as switching image names or Helm chart references, may be necessary, migrating existing Redis deployments to KeyDB requires minimal effort.

Operational Integration

KeyDB maintains operational compatibility with the Redis ecosystem, enabling existing tooling and workflows to function with minimal to no change. It fully supports `redis-cli`. For instance, supporting the same persistence mechanics as Redis, both RDB and AOF, ensuring compatibility with existing backup and failover procedures. Monitoring remains straightforward, KeyDB exposes runtime metrics via the `INFO` command, allowing Prometheus exporters and Grafana dashboards designed for Redis to operate without modification [86]. While KeyDB introduces additional operational features, such as configurable threading, these are optional and do not interfere with a standard migration, but may not be backwards-compatible with other Redis-compatible systems. In practice, replacing Redis with KeyDB does not disrupt operational workflows.

5.5.3 Garnet

This subsection provides an overview of Garnet's compatibility with Redis and the migration effort required.

API Compatibility

Garnet is a reimplementation that uses the RESP protocol for client communication, but it is not intended to be a perfect drop-in replacement for Redis. The team behind Garnet describes it as a "close-enough starting point" [87]. This means that while most Redis clients can connect to a Garnet server and execute basic commands (e.g., GET, SET) using the same syntax and data types, full compatibility is not guaranteed. Garnet does not support the entire Redis API, meaning that applications that rely on more advanced features, such as modules or less-common data structures, may require code changes or workarounds. For simple caching use cases, migration is likely to be straightforward, but more complex Redis deployments may be incompatible. In summary, Garnet offers partial API compatibility, sufficient for many scenarios, but not a seamless drop-in replacement unless Redis usage is limited to core functionality.

Deployment and Configuration

Garnet integrates well with Kubernetes-based and containerised environments. Official Docker images are available via the GitHub Container Registry [88], and a Helm chart is provided and maintained in the official repository [54]. The deployment process is similar to that of Redis. While minor modifications, such as adjusting image sources or resource settings, may be necessary, deploying Garnet in Kubernetes environments requires minimal effort.

Operational Integration

Garnet provides partial operational compatibility with the Redis ecosystem, meaning existing tooling and workflows may need modifications. It supports `redis-cli` and common administrative commands, including AOF persistence, offering familiar mechanisms for

durability [87]. However, its persistence files are not fully compatible with the format Redis has, and cannot be directly exchanged between Redis and Garnet [89]. This means that migrating data between the systems may require intermediate steps, such as exporting data from Redis and reloading it into Garnet using standard commands. Garnet also lacks support for the Redis `INFO` command, which limits compatibility with Prometheus exporters and Grafana dashboards designed for Redis observability. The community has acknowledged this limitation, and interest in adding these monitoring features has been raised in the official repository [90]. Garnet is not a drop-in replacement for Redis and may interfere with existing workflows that depend on Redis-specific monitoring or backup tools.

5.5.4 Summary

The analysis shows that Redis alternatives vary significantly regarding compatibility and migration effort. Valkey and KeyDB closely align with Redis in protocol support, tooling, and deployment practices, making them suitable for environments where minimal disruption is desired. Garnet requires more consideration due to its partial compatibility and limitations in operational integration. While it supports core Redis functionality, it lacks full command coverage and integration with common observability tools, which may require adjustments to existing workflows. These differences highlight that migration effort is not uniform across Redis alternatives.

A summary of the compatibility and integration characteristics for each system is presented in Table 11.

Table 11 Summary of Compatibility and Migration Effort.

Dimension	**Valkey**	**KeyDB**	**Garnet**
RESP Support	Yes	Yes	Partial
Docker/Helm Support	Yes	Yes	Yes
`redis-cli` Support	Yes	Yes	Partial
Prometheus Integration	Yes	Yes	No
Migration Effort	Low	Low	Moderate–High

5.6 Long-Term Viability Analysis

Long-term viability is a crucial factor when evaluating Redis alternatives, as it ensures the future sustainability of each system. Several aspects are of particular interest: active development, community support, project governance, and the licensing model. These topics are further outlined in Section 4.6 of the methodology.

5.6.1 Development Activity

To evaluate how actively each system's active development, this section examines two CHAOSS metrics: *Release Frequency* [74] and *Code Changes Commits* [75]. Release Frequency measures how often a project publishes stable versions of its software, while Code Changes Commits captures the volume of code-altering commits.

This analysis is based on a 12 month period, from May 1, 2024 to May 1, 2025. To determine the number of published stable releases, each project's GitHub repository was manually inspected. To count commits during this period, the Git command shown in Appendix D, Listing D.1, was used.

Valkey

Valkey maintained a steady development pace. A total of seven stable releases were published during this period, averaging one release approximately every two months. In terms of code activity, the repository recorded 758 commits to the main branch, resulting in an average of approximately 63 commits per month. This reflects a consistent update cycle and regular maintenance of the codebase.

KeyDB

KeyDB released zero stable versions, with the most recent release published on 30 October 2023. During the same period, the repository recorded zero commits to the main branch, with the latest commit made on 5 April 2024. This represents a notable decline in visible development activity. Since its initial release in 2019, the project has accumulated a total of 27 522 commits, but none during the examined 12 month period. Upon further inspection, a comment from one of the main contributors on 30 April 2024 indicated that development is ongoing and that current development is being conducted on a separate branch outside of the main branch [91]. As such, current development efforts may not be reflected in the main branch activity.

Garnet

Garnet showed a highly rapid release rate, with 65 stable versions published, equivalent to more than five releases per month. The repository also recorded 549 commits to the main branch, corresponding to an average of approximately 46 commits per month. The project exhibits a notably high release frequency and a consistent commit rate.

5.6.2 Community Support

To evaluate the health of the developer community for each system, this subsection examines two CHAOSS metrics: *Committers* [76] and *Issues Closed* [77]. These metrics provide insight into the size of the active contributing part of the community and the project's responsiveness to user-reported issues.

The analysis is based on the same 12 month period, from May 1, 2024 to May 1, 2025. To count the number of unique contributors (committers), the Git command in Appendix D, Listing D.2, was used. To obtain the number of closed issues during the same period, the GitHub issue query shown in Appendix D, Listing D.3, was used via the repository's issue tracker.

Valkey

Valkey had 131 unique contributors who made commits to the repository. This suggests a relatively broad and active developer base. In the same period, a total of 247 issues were closed, indicating a high level of responsiveness to reported problems and user feedback.

KeyDB

KeyDB recorded zero unique contributors to the main branch, indicating an absence of visible code contributions during the period. However, 11 issues were closed in the same timeframe. This suggests that while there was no activity development in the main branch, some level of issue management or administrative maintenance was still ongoing.

Garnet

Garnet had 45 unique contributors who committed to the repository, reflecting an active developer community. During the same period, 181 issues were closed, indicating a high level of responsiveness and ongoing engagement with users and contributors.

5.6.3 Governance and Ownership

This section examines the governance and ownership structure of each project to evaluate the level of independence, transparency, and potential risks related to long-term sustainability. Understanding whether a project is governed by a single company, community, or foundation is important, as it influences decision-making processes and the likelihood of sudden shifts in direction or licensing [8].

Valkey

Valkey is governed under an open, vendor-neutral model established by the Linux Foundation. Its technical leadership committee, composed of former Redis core contributors, guides the project's technical direction and ensures it aligns with the community's goals. Multiple major companies such as AWS, Google Cloud, Oracle, Ericsson, and Snap Inc. have pledged long-term support for Valkey's independence and sustainability. The project is not controlled by any single entity. Instead, its governance process is publicly documented and emphasises community participation, transparency, and long-term stability [92]. With its firm commitment to remaining open-source, community-led, and backed by influential contributors and organisations, Valkey stands on a solid foundation.

KeyDB

KeyDB's governance is company-driven. After Snap Inc. acquired the KeyDB project in 2022, development has been maintained under Snap's GitHub organisation since. The project's direction is determined by Snap-employed maintainers, and there is no independent governance structure or public decision-making process. While external contributions are welcomed, the project's long-term roadmap and priorities remain closely tied to

Snap's internal strategy. Although Snap emphasises its commitment to keeping KeyDB open-source, the company-governed model may carry a higher risk of future changes in direction, licensing, or development focus that reflect corporate interests over community needs [93, 8].

Garnet

Garnet is governed by Microsoft Research under a corporate open-source model. The project is hosted on Microsoft's GitHub organisation and developed primarily by internal Microsoft teams. While governance remains centralised, Microsoft publishes a public roadmap on the Garnet website [94], outlining short-, medium-, and long-term goals, and invites community feedback on feature prioritisation and the project's long-term direction [95]. Despite the absence of a formal independent governance structure, Garnet actively considers community input when shaping its future. However, overall control of the project's direction and decision-making ultimately resides with Microsoft.

5.6.4 Licensing Model

This section examines the license models under which each project is released, as licensing influences openness, commercial usability, and long-term viability. A license is considered fully open-source if it allows users to freely use, modify, and distribute the software, including for commercial purposes, usually under terms approved by the Open Source Initiative (OSI). If a license is approved by the OSI, it implies that the system meets all criteria of the Open Source Definition and is therefore considered fully open-source [30, 31]. Comparing the licenses of Valkey, KeyDB, and Garnet highlights their relative openness and the conditions they set for adoption.

All three alternatives have OSI-approved licenses, specifically the MIT License [96] and the BSD 3-Clause License [97], which allow free use, modification, and redistribution in both source and binary forms. Meaning users are permitted to share the original or modified source code, as well as distribute compiled versions as part of a product, without any obligation to disclose the source code. These licenses impose minimal requirements, users must give credit to the original copyright holder and include the full license text and disclaimer in any redistribution. They also include a broad warranty disclaimer and limit liability.

Valkey (BSD 3-Clause License)

Valkey is released under the BSD 3-Clause License [98], an OSI-approved license [97]. The only significant difference between the BSD 3-Clause License and the MIT License is that the BSD 3-Clause License includes a clause prohibiting the use of the copyright holder's or contributors' names to endorse or promote products derived from the software without specific written permission.

KeyDB and Garnet (MIT License)

KeyDB [99] and Garnet [100] are both released under the MIT License [96], another OSI-approved license. The MIT License offers nearly identical freedoms to BSD 3-Clause but

does not include the endorsement clause.

5.6.5 Summary

The long-term viability analysis reveals clear differences between the three Redis alternatives, particularly in development activity, community support, and governance. A summary of the findings is displayed in Table 12, with the numeric data based on data collected from a 12 month period from May 1 2024 to May 1 2025.

Table 12 Summary of long-term viability for Redis alternatives.

Dimension	**Valkey**	**KeyDB**	**Garnet**
Development Activity	High	None	High
Community Support	Strong	Absent	Moderate
Governance	Foundation-led	Corporate	Corporate
Licensing	Open-Source	Open-Source	Open-Source

Table 13 Summary of long-term viability for Redis alternatives, the numeric data is based on data collected from May 1 2024 to May 1 2025

Category	**Valkey**	**KeyDB**	**Garnet**
Development Activity	7 releases 758 commits	0 releases 0 commits	65 releases 549 commits
Unique Contributors	131 contributors	0 contributors	45 contributors
Issues Closed	247 closed issues	11 closed issues	181 closed issues
Governance	Linux Foundation Community-led	Snap Company-led	Microsoft Company-led
License	BSD 3-Clause	MIT	MIT
Overall Viability	Strong and sustainable	Weak and unclear direction	Promising but centralised

Based on the findings, Valkey stands strong in terms of long-term sustainability. It showed a high development activity, with the most commits and the largest number of closed issues. This reflects an active and responsive contributor community. Although it had fewer stable releases than Garnet, the high number of commits suggests a lot of work happens between releases. Valkey seems more community-driven, with the largest number of contributors, where many different people help improve the project over time. Its governance under the Linux Foundation ensures that development stays open and transparent, without being controlled by any company. Together, these factors make Valkey the most viable option for the long-term.

5.7 Overall Summary of Findings

The findings concluded that Garnet delivered the highest overall performance across all tested workloads and metrics, consistently achieving significantly higher throughput and lower tail latencies compared to Redis and the other alternatives. It also demonstrated stable and efficient use of CPU and memory, notably handling the same dataset while using over 10% less memory than that of Redis. In addition to its technical strengths, Garnet displayed a strong and rapid development pace, releasing 65 stable versions in the past year and resolving many community-reported issues, highlighting strong maintenance and a clear commitment to the open-source community. However, Garnet is not a drop-in replacement; it only partially supports the Redis API and lacks full integration with Redis tooling, which presents migration challenges for systems relying on advanced features or established observability stacks. While Microsoft has shown a willingness to engage with the community, the project remains company-led, with final decision-making ultimately falling under Microsoft's control. In summary, Garnet shows exceptional promise, but its adoption must be carefully weighed against its current limitations.

KeyDB showed zero development activity during the evaluation period, with no new releases, commits, or active contributors. Although a few issues were closed, they did not lead to any visible changes in the codebase. Snap governs The project entirely, with minimal to no sign of ongoing engagement. As a result, KeyDB currently lacks the transparency, activity, and community involvement needed to be considered a viable long-term alternative.

From a licensing perspective, all three projects are released under OSI-approved licenses. Valkey uses the BSD 3-Clause License, while KeyDB and Garnet use the MIT License. The only distinction is that BSD 3-Clause prohibits using the copyright holder's or contributors' names to endorse or promote products derived from the software without specific written permission, which the MIT license allows. In most cases, this distinction is most probably negligible for typical users.

In conclusion, Valkey emerges as the most sustainable long-term alternative to Redis, combining active development, strong community involvement, and open, community-led governance. Garnet also presents a reliable and well-maintained option, with fast release cycles and high responsiveness to the community's needs, but its company-led model can limit the community influence and long-term transparency. In contrast, KeyDB currently shows minimal to no signs of active development or community engagement, making it difficult to recommend as a viable long-term alternative.

6 Conclusion

This thesis evaluated Valkey, KeyDB, and Garnet as potential open-source successors to Redis, focusing on performance characteristics, compatibility, and long-term viability. The goal was to determine which system is the most suitable overall, and under what scenarios each alternative may be a better fit.

The findings concluded that Garnet delivered the highest overall performance across all tested workloads and metrics. Its multithreaded architecture enabled significantly higher throughput and lower tail latencies than Redis and the other alternatives. It simultaneously exhibits stable and efficient use of CPU and memory, handling the same dataset while using over 10% less memory than that of Redis. In addition to its technical strengths, Garnet displayed a highly rapid development, releasing 65 stable versions in the past year and actively resolving many community-reported issues. It highlights strong maintenance and a clear commitment to the overall community. However, while Microsoft showed a strong willingness to engage with the community, the project remains company-led, with final decision-making ultimately falling under Microsoft. Furthermore, Garnet is not a drop-in replacement. It only partially supports the Redis API and lacks full integration with Redis tooling, which presents migration challenges for systems relying on advanced features or established observability stacks. In summary, Garnet shows exceptional promise, but its adoption must be carefully weighed against its limitations.

Valkey delivered performance comparable to that of Redis, with clear advantages emerging at higher concurrency levels, particularly in read-heavy scenarios, where it consistently achieved higher throughput, lower tail latencies, and more efficient CPU utilisation. While it does not introduce new architectural innovations, Valkey's key strength lies in its full API compatibility. It is a genuine drop-in replacement that supports all Redis commands, client libraries, and observability tools. This seamless integration allows existing Redis deployments to transition with minimal effort. Furthermore, Valkey demonstrated high development activity and community support, with over 750 commits, seven stable releases, 131 unique contributors, and 247 resolved community-reported issues over the past year. Governed by the Linux Foundation and backed by major cloud providers, the project is community-led and built on transparent, vendor-neutral governance. In conclusion, Valkey stands out as the most sustainable long-term alternative to Redis, offering operational continuity, full ecosystem compatibility, and the reliability of a well-supported open-source project.

KeyDB, while promising in design due to its multithreading capabilities, failed to deliver consistent performance improvements and generally failed to outperform Redis. It is a complete drop-in replacement for Redis, with full API compatibility and integrates seamlessly with existing Redis tooling. More concerning is the lack of development activity over the past year, with no commits, releases, or active contributors. Governed solely by Snap, the project shows little sign of ongoing maintenance or community engagement. In summary, KeyDB's stagnant development and underwhelming performance make it an unsuitable suc-

cessor for Redis.

Several limitations regarding the benchmarking must be acknowledged, as discussed in Section 4.3.7. First, the CPU limit of two cores was uniformly applied to all systems, which may underrepresent the performance potential of multithreaded systems like Garnet and KeyDB. Second, all deployments were configured as standalone instances without clustering or replication, which does not fully reflect how Redis and its successors are typically used in production environments. Finally, default parameters from Helm charts may have introduced configuration biases, and the averaging method used for resource usage could obscure transient spikes. These limitations mean that results should be interpreted as baseline indicators under controlled conditions rather than fully reflecting how the systems would behave in real-world production environments.

Future work should address these limitations by exploring clustered or multi-instance deployments, especially for systems like Redis and Valkey that rely on sharding for scalability. Benchmarking these systems in distributed setups would provide more realistic insights into horizontal scalability, high availability, and resilience under node failure. Additionally, observability integration deserves further investigation. While this thesis considered metric exposure and compatibility at a high level, future studies should validate how well Redis alternatives integrate with established monitoring stacks, such as Prometheus exporters and Grafana dashboards. This includes testing metric coverage, reliability of alerting mechanisms, compatibility with existing operational workflows, and key requirements for production adoption.

This work provides the first comprehensive evaluation of Redis-compatible successors from a holistic perspective. By combining performance benchmarks, migration analysis, and long-term viability metrics within a controlled Kubernetes environment, this thesis offers data-driven insights for engineers and decision-makers considering open-source alternatives to Redis. It helps identify which system is best suited for high-throughput workloads, seamless integration, or long-term sustainability. This structured evaluation fills a gap in the existing research and provides a solid foundation for future studies seeking to explore Redis alternatives in a more complex or distributed settings.

Based on this study's findings, the following recommendations can be made: Valkey is the most suitable open-source successor to Redis for general-purpose use, especially in environments that prioritise full compatibility and minimal operational disruption. For use cases where maximum performance is critical and Redis usage is limited to core functionality, Garnet is the preferred choice. While Garnet requires more migration effort, its substantial performance advantages may justify the trade-off in some scenarios.

Bibliography

[1] Showan Esmail Asyabi, "A Survey on In-Memory KV Store Designs for Today's Data Centers," in *Proceedings of the 2022 International Conference on Advanced Computer Science and Information Systems (ICACSIS)*, 2022, accessed: 2025-04-29. [Online]. Available: https://api.semanticscholar.org/CorpusID:247863259

[2] Martin Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Sebastopol, CA: O'Reilly Media, 2017.

[3] Redis. (2025) Redis Caching Solutions. Accessed: 2025-04-29. [Online]. Available: https://redis.io/solutions/caching/

[4] Juncheng Yang and Yao Yue and K. V. Rashmi, "A Large-Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, 2020, pp. 191–208, accessed: 2025-04-29. [Online]. Available: https://www.usenix.org/system/files/osdi20-yang.pdf

[5] Lindsay Clark. (2020) Redis: The Most Popular Database on AWS – Ahead of MySQL and PostgreSQL. The Register. Accessed: 2025-04-29. [Online]. Available: https://www.theregister.com/2020/11/23/redis_the_most_popular_db_on_aws/

[6] DB-Engines, "DB-Engines Ranking - Key-Value Stores," https://db-engines.com/en/ranking/key-value%2Bstore, accessed: 2025-04-29.

[7] Redis. Redis Adopts Dual Source-Available Licensing. Accessed: 2025-04-29. [Online]. Available: https://redis.io/blog/redis-adopts-dual-source-available-licensing/

[8] Dawn Foster, "THE NEW DYNAMICS OF OPEN SOURCE: RELICENSING, FORKS, AND COMMUNITY IMPACT," accessed: 2025-05-11. [Online]. Available: https://chaoss.community/?p=3630

[9] Amazon Web Services, "What is Valkey?" accessed: 2025-05-06. [Online]. Available: https://aws.amazon.com/elasticache/what-is-valkey/

[10] KeyDB. Our Story - KeyDB Documentation. Accessed: 2025-04-29. [Online]. Available: https://docs.keydb.dev/docs/about#our-story

[11] Microsoft Research. (2024) Introducing Garnet: An Open-Source, Next-Generation, Faster Cache Store for Accelerating Applications and Services. Accessed: 2025-04-29. [Online]. Available: https://www.microsoft.com/en-us/research/blog/introducing-garnet-an-open-source-next-generation-faster-cache-store-for-accelerating-applications-and-services/

[12] Rowan Trollope, “Redis is now available under the AGPLv3 open source license,” 2025, accessed: 2025-04-23. [Online]. Available: https://redis.io/blog/agplv3/

[13] The Linux Foundation, “Linux Foundation Launches Open Source Valkey Community,” accessed: 2025-05-12. [Online]. Available: https://www.linuxfoundation.org/press/linux-foundation-launches-open-source-valkey-community

[14] Redis, “Redis Race Condition,” accessed: 2025-05-24. [Online]. Available: https://redis.io/glossary/redis-race-condition/

[15] Redis. Redis. Accessed: 2025-04-29. [Online]. Available: https://redis.io

[16] Will Johnston. (2022) Redis as a Cache vs Redis as a Primary Database in 90 Seconds. Accessed: 2025-04-29. [Online]. Available: https://redis.io/blog/redis-cache-vs-redis-primary-database-in-90-seconds/

[17] Redis, “INFO,” accessed: 2025-04-29. [Online]. Available: https://redis.io/docs/latest/commands/info/

[18] Redis, “Redis Enterprise Software observability and monitoring guidance,” accessed: 2025-05-18. [Online]. Available: https://redis.io/docs/latest/integrate/prometheus-with-redis-enterprise/observability/

[19] Valkey. FAST. RELIABLE. OPEN SOURCE, FOREVER. Accessed: 2025-04-29. [Online]. Available: https://valkey.io/

[20] The Linux Foundation. Accessed: 2025-04-29. [Online]. Available: https://www.linuxfoundation.org/projects/hosting

[21] KeyDB. KeyDB - A Database Built for Scale. Accessed: 2025-04-29. [Online]. Available: https://docs.keydb.dev/

[22] KeyDB, “A Multithreaded Fork of Redis That’s 5X Faster Than Redis,” accessed: 2025-05-18. [Online]. Available: https://docs.keydb.dev/blog/2019/10/07/blog-post/

[23] Microsoft. Garnet: A High-Performance Key-Value Store. Accessed: 2025-04-29. [Online]. Available: https://microsoft.github.io/garnet/

[24] Garnet, “Introduction to Tsavorite,” accessed: 2025-05-24. [Online]. Available: https://microsoft.github.io/garnet/docs/dev/tsavorite/intro

[25] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett, “Faster: A Concurrent Key-Value Store with In-Place Updates,” accessed: 2025-05-24. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3183713.3196898

[26] M. van Steen and A. S. Tanenbaum, “A brief introduction to distributed systems,” *Computing*, vol. 98, pp. 967–1009, 2016.

[27] Redis, “Scaling with Redis,” accessed: 2025-05-16. [Online]. Available: https://redis.io/docs/latest/operate/oss_and_stack/management/scaling/

[28] Kubernetes. Production-Grade Container Orchestration. Accessed: 2025-05-16. [Online]. Available: https://kubernetes.io/

[29] Helm, “Helm - The Kubernetes Package Manager,” [Online]. Available: https://helm.sh/, accessed: 2025-04-29.

[30] Open Source Initiative, “The Open Source Definition,” accessed: 2025-05-12. [Online]. Available: https://opensource.org/osd

[31] Open Source Initiative, “OSI Approved Licenses,” accessed: 2025-05-12. [Online]. Available: https://opensource.org/licenses

[32] Datadog. (2015) How to Monitor Redis Performance Metrics. Accessed: 2025-04-29. [Online]. Available: https://www.datadoghq.com/blog/how-to-monitor-redis-performance-metrics/

[33] Brian F. Cooper and Adam Silberstein and Erwin Tam and Raghu Ramakrishnan and Russell Sears, “Benchmarking Cloud Serving Systems with YCSB,” accessed: 2025-05-16. [Online]. Available: https://courses.cs.duke.edu/fall13/compsci590.4/838-CloudPapers/ycsb.pdf

[34] Redis. memtier_benchmark: A High-Throughput Benchmarking Tool for Redis & Memcached. Accessed: 2025-04-29. [Online]. Available: https://redis.io/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/

[35] Anthony Mbata, Yaji Sripada, and Mingjun Zhong, “A Survey of Pipeline Tools for Data Engineering,” accessed: 2025-05-25. [Online]. Available: https://arxiv.org/pdf/2406.08335

[36] Abdulrahman Munappy and John Grundy and Steve Howard and Hoa Khanh Dam, “Data Pipeline Management in Practice: Challenges and Opportunities,” in *Product-Focused Software Process Improvement (PROFES)*, vol. 12562. Springer, 2020, pp. 168–184. [Online]. Available: https://doi.org/10.1007/978-3-030-64148-1_11

[37] Brian F. Cooper and Adam Silberstein and Erwin Tam and Raghu Ramakrishnan and Russell Sears, “Benchmarking Cloud Serving Systems with YCSB,” in *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*. ACM, 2010, pp. 143–154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[38] Vincent Reniers and Dimitri Van Landuyt and Ansar Rafique and Wouter Joosen, “On the State of NoSQL Benchmarks,” in *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering Companion (ICPE)*. ACM, 2017, pp. 107–112. [Online]. Available: https://doi.org/10.1145/3053600.3053622

[39] Ragab, Mohamed and Awaysheh, Feras M. and Tommasini, Riccardo, “Bench-Ranking: A First Step Towards Prescriptive Performance Analyses For Big Data Frameworks,” in *2021 IEEE International Conference on Big Data (Big Data)*, 2021, pp. 241–251.

[40] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny, “Workload Analysis of a Large-Scale Key-Value Store,” in *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, 2012, pp. 53–64. [Online]. Available: https://doi.org/10.1145/2254756.2254766

[41] Mohammad Abu Kausar, Mohammad Nasar, and Arockiasamy Soosaimanickam, "A Study of Performance and Comparison of NoSQL Databases: MongoDB, Cassandra, and Redis Using YCSB," 2022, accessed: 2025-05-19. [Online]. Available: https://doi.org/10.17485/IJST/v15i31.1352

[42] Nadia Ben Seghier and Okba Kazar, "Performance Benchmarking and Comparison of NoSQL Databases: Redis vs MongoDB vs Cassandra Using YCSB Tool," 2021, accessed: 2025-05-19. [Online]. Available: https://doi.org/10.1109/ICRAMI52622.2021.9585956

[43] Mustafa Alzaidi, Aniko Vagner, "Benchmarking Redis and HBase NoSQL Databases using Yahoo Cloud Service Benchmarking tool," accessed: 2025-05-12. [Online]. Available: https://real.mtak.hu/156196/1/AMI_56_from1to9.pdf

[44] Wenqi Cao, Semih Sahin, Ling Liu, and Xianqiang Bao, "Evaluation and Analysis of In-Memory Key-Value Systems." [Online]. Available: https://istc-cc.cmu.edu/publications/papers/2016/Cao-IEEEBigData2016.pdf

[45] Awaysheh, Feras M. and Alazab, Mamoun and Garg, Sahil and Niyato, Dusit and Verikoukis, Christos, "Big Data Resource Management Networks: Taxonomy, Survey, and Future Directions," *IEEE Communications Surveys Tutorials*, vol. 23, no. 4, pp. 2098–2130, 2021.

[46] Aerospike, "Ultra High Performance NoSQL Benchmark Report," https://aero-media.aerospike.com/2018/05/UltraHighPerformanceNoSQL.pdf, 2013, accessed: 2025-04-29.

[47] MemVerge, "Memory Machine and In-Memory Databases," https://memverge.com/wp-content/uploads/2021/03/White-Paper_Memory-Machine-and-In-memory-Databases.pdf, accessed: 2025-04-29.

[48] Vladimir Starostenkov, "Redis-as-a-Service Benchmark," https://www.altoros.com/wp-content/uploads/pdf/Redis-as-a-Service_Benchmark.pdf, 2015, accessed: 2025-04-29.

[49] Ahmed Awad and Feras Awaysheh and Hugo A. López, "BEST: A Unified Business Process Enactment via Streams and Tables for Service Computing," 2025. [Online]. Available: https://arxiv.org/abs/2501.14848

[50] Kubernetes. Kubeadm. Accessed: 2025-04-29. [Online]. Available: https://kubernetes.io/docs/reference/setup-tools/kubeadm/

[51] Bitnami. Redis Helm Chart. Accessed: 2025-04-29. [Online]. Available: https://artifacthub.io/packages/helm/bitnami/redis

[52] Bitnami. Valkey Helm Chart. Accessed: 2025-04-29. [Online]. Available: https://github.com/bitnami/charts/blob/main/bitnami/valkey/README.md

[53] Enapter, "keydb," https://artifacthub.io/packages/helm/enapter/keydb, accessed: 2025-04-29.

[54] Microsoft. Garnet Helm Chart. Accessed: 2025-04-22. [Online]. Available: https://github.com/microsoft/garnet/tree/main/charts/garnet

[55] Redis. Hardware Requirements. Accessed: 2025-04-29. [Online]. Available: https://redis.io/docs/latest/operate/rs/installing-upgrading/install/plan-deployment/hardware-requirements/

[56] Redis. memtier_benchmark. Accessed: 2025-04-29. [Online]. Available: https://github.com/redislabs/memtier_benchmark

[57] Redis. Benchmarking Redis Enterprise. Accessed: 2025-04-29. [Online]. Available: https://redis.io/docs/latest/operate/rs/clusters/optimize/memtier-benchmark/

[58] KeyDB. Putting KeyDB to the Test! Accessed: 2025-04-29. [Online]. Available: https://docs.keydb.dev/docs/benchmarking/

[59] Valkey. Documentation: Benchmarking Tool. Accessed: 2025-04-29. [Online]. Available: https://valkey.io/topics/benchmark/

[60] Brian Frank Cooper. Yahoo! Cloud Serving Benchmark (YCSB). Accessed: 2025-04-29. [Online]. Available: https://github.com/brianfrankcooper/YCSB

[61] ScyllaDB. (n.d.) YCSB Definition. Accessed: 2025-04-29. [Online]. Available: https://www.scylladb.com/glossary/ycsb/

[62] Babuji, Yadu N. and Chard, Kyle and Gerow, Aaron and Duede, Eamon, "Cloud Kotta: Enabling secure and scalable data analytics in the cloud," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 302–310.

[63] Marc Brooker. Hot Keys, Scalability, and the Zipf Distribution. Accessed: 2025-04-29. [Online]. Available: https://brooker.co.za/blog/2023/02/07/hot-keys.html

[64] Qian Liu and Haolin Yuan, "A High-Performance Memory Key-Value Database Based on Redis," https://scispace.com/pdf/a-high-performance-memory-key-value-database-based-on-redis-280vpqx4nt.pdf, 2019, accessed: 2025-04-29.

[65] YCSB. Git Repository - YCSB Workloads. Accessed: 2025-04-29. [Online]. Available: https://github.com/brianfrankcooper/YCSB/tree/master/workloads

[66] Microsoft. Performance Testing with Azure Managed Redis (Preview). Accessed: 2025-04-29. [Online]. Available: https://learn.microsoft.com/en-us/azure/redis/best-practices-performance

[67] Rachit Pandey, "Performance Benchmarking and Comparison of Cloud-Based Databases MongoDB (NoSQL) Vs MySQL (Relational) using YCSB," https://doi.org/10.13140/RG.2.2.10789.32484, 2020, accessed: 2025-04-29.

[68] Redis. Redis benchmark. Accessed: 2025-04-25. [Online]. Available: https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/

[69] Baeldung, "Caching: Cold Cache vs. Warm Cache," https://www.baeldung.com/cs/caching-cold-cache-vs-warm-cache, 2024, accessed: 2025-04-23.

[70] Daniel S. Berger and Benjamin Berg and Timothy Zhu and Mor Harchol-Balter and Siddhartha Sen, "RobinHood: Tail Latency-Aware Caching—Dynamically Reallocating from Cache-Rich to Cache-Poor," 2018, accessed: 2025-04-30. [Online]. Available: https://www.cs.cmu.edu/~harchol/Papers/OSDI18.pdf

[71] Last9, “Tail Latency: The Silent Killer of Distributed Systems,” https://last9.io/blog/tail-latency/, 2024, accessed: 2025-04-30.

[72] Redis, “Redis serialization protocol specification,” accessed: 2025-05-07. [Online]. Available: https://redis.io/docs/latest/develop/reference/protocol-spec/

[73] CHAOSS, “CHAOSS,” accessed: 2025-05-10. [Online]. Available: https://chaoss.community/

[74] CHAOSS, “Release Frequency,” accessed: 2025-05-10. [Online]. Available: https://chaoss.community/?p=4765

[75] CHAOSS, “Code Changes Commits,” accessed: 2025-05-10. [Online]. Available: https://chaoss.community/?p=4707

[76] CHAOSS, “Committers,” accessed: 2025-05-10. [Online]. Available: https://chaoss.community/?p=3945

[77] CHAOSS, “Issues Closed,” accessed: 2025-05-10. [Online]. Available: https://chaoss.community/?p=3633

[78] Valkey, “API Compatibility,” accessed: 2025-05-07. [Online]. Available: https://valkey.io/commands/info/

[79] Valkey, “valkey/valkey,” accessed: 2025-05-06. [Online]. Available: https://hub.docker.com/r/valkey/valkey

[80] Valkey. Using Bitnami’s Valkey chart. Accessed: 2025-05-06. [Online]. Available: https://valkey.io/blog/using-bitnami-valkey-chart/

[81] Valkey, “INFO,” accessed: 2025-05-07. [Online]. Available: https://valkey.io/commands/info/

[82] Redis, “Redis persistence,” accessed: 2025-05-07. [Online]. Available: https://redis.io/docs/latest/develop/reference/protocol-spec/#request-response-model

[83] Valkey, “Documentation: Persistence,” accessed: 2025-05-07. [Online]. Available: https://valkey.io/topics/persistence/

[84] KeyDB, “Protocol Specification,” accessed: 2025-05-07. [Online]. Available: https://docs.keydb.dev/docs/compatibility/#:~:text=Compatibility%20with%20Redis

[85] Eqalpha, “eqalpha/keydb,” accessed: 2025-05-08. [Online]. Available: https://hub.docker.com/r/eqalpha/keydb

[86] KeyDB, “INFO,” accessed: 2025-05-08. [Online]. Available: https://docs.keydb.dev/docs/commands/#info

[87] Garnet, “API Compatibility,” accessed: 2025-05-08. [Online]. Available: https://microsoft.github.io/garnet/docs/welcome/compatibility

[88] Microsoft, “Garnet,” accessed: 2025-05-08. [Online]. Available: https://docs.keydb.dev/docs/commands/#info

[89] Garnet, “Can I quickly import RDB data?” accessed: 2025-05-09. [Online]. Available: https://valkey.io/commands/info/

[90] Garnet, “Garnet Monitoring Integration in Helm Chart,” accessed: 2025-05-09. [Online]. Available: https://github.com/microsoft/garnet/issues/629

[91] John Sully, “Is the project no longer in development? 788,” accessed: 2025-05-11. [Online]. Available: https://github.com/Snapchat/KeyDB/issues/788

[92] Redis, “Redis persistence,” accessed: 2025-05-07. [Online]. Available: https://redis.io/docs/latest/develop/reference/protocol-spec/#request-response-model

[93] KeyDB, “KeyDB (YC S20) Joins Snap,” accessed: 2025-05-12. [Online]. Available: https://docs.keydb.dev/news/2022/05/12/keydb-joins-snap/

[94] Garnet, “Garnet Roadmap,” accessed: 2025-05-12. [Online]. Available: https://microsoft.github.io/garnet/docs/welcome/roadmap

[95] Garnet, “Welcome to Garnet,” accessed: 2025-05-12. [Online]. Available: https://microsoft.github.io/garnet/docs

[96] Open Source Initiative, “The MIT License,” accessed: 2025-05-12. [Online]. Available: https://opensource.org/license/mit

[97] Open Source Initiative, “The 3-clause bsd license,” accessed: 2025-05-12. [Online]. Available: https://opensource.org/liRedisandValkeynativelyexposeinternalmetricsviathecense/bsd-3-clause

[98] Valkey, “License,” accessed: 2025-05-12. [Online]. Available: https://github.com/valkey-io/valkey?tab=License-1-ov-file

[99] KeyDB, “License,” accessed: 2025-05-12. [Online]. Available: https://github.com/DjDeveloperr/Keydb/blob/main/LICENSE

[100] Garnet, “License,” accessed: 2025-05-12. [Online]. Available: https://github.com/microsoft/garnet/blob/main/LICENSE

A Appendix: YAML Configurations

Listing A.1: Helm values used to deploy Redis and KeyDB in Kubernetes

```
architecture: standalone

auth:
  enabled: true
  password: pass

master:
  persistence:
    enabled: false
  service:
    type: NodePort
    nodePort: 30001
  resources:
    requests:
      cpu: "1"
      memory: "6Gi"
    limits:
      cpu: "2"
      memory: "8Gi"
```

Listing A.2: Helm values used to deploy Valkey in Kubernetes.

```
architecture: standalone

auth:
  enabled: true
  password: pass

primary:
  persistence:
    enabled: false
  service:
    type: NodePort
    nodePort: 30001
  resources:
    requests:
      cpu: "1"
      memory: "6Gi"
    limits:
      cpu: "2"
      memory: "8Gi"
```

Listing A.3: Helm values used to deploy Garnet in Kubernetes.

```
auth:
  enabled: true
  password: pass

persistence:
  enabled: false

service:
  type: NodePort
  port: 6379
  nodePort: 30001

resources:
  requests:
    cpu: "1"
    memory: "6Gi"
  limits:
    cpu: "2"
    memory: "8Gi"
```

B Appendix: Resource Monitoring Script

The placeholder <POD_NAME> should be replaced with the actual name of the running pod (e.g., redis-master-0) prior to execution.

Listing B.1: Bash script for collecting resource usage in Kubernetes.

```
#!/bin/bash

mkdir -p results
resource_log="results/resource_usage.csv"
echo "timestamp,redis-master-cpu,redis-master-mem,k8s-master-cpu,k8s-master-mem" >
     "$resource_log"

collect_resources() {
  timestamp=$(date +"%Y-%m-%d %H:%M")

  redis_master_cpu=$(kubectl top pod <POD_NAME> --no-headers | awk '{print $2}')
  redis_master_mem=$(kubectl top pod <POD_NAME> --no-headers | awk '{print $3}')

  k8s_master_cpu=$(kubectl top node k8s-master-new --no-headers | awk '{print $3}'
    )
  k8s_master_mem=$(kubectl top node k8s-master-new --no-headers | awk '{print $5}'
    )

  echo "$timestamp,$redis_master_cpu,$redis_master_mem,$k8s_master_cpu,
    $k8s_master_mem" >> "$resource_log"
}

while true; do
  sleep 10
  collect_resources
done
```

C Appendix: Benchmark Script

This shell script automates the benchmarking workflow, including warm-up, benchmarking execution, and resource monitoring. It calls a separate script `get_CPU_RAM.sh` (see Appendix B) to record CPU and memory usage during each benchmark continuously. The main script iterates over multiple concurrency levels and workloads, saving raw output and structured CSV data for analysis.

Instructions

Before running the script, update the following placeholders:

- `<IP>` – the IP address of the system's endpoint
- `<PORT>` – the port the system is exposed on

Make sure `get_CPU_RAM.sh` (see Appendix B) is in the same directory. Results will be saved in `results/WA/` and `results/WB/`.

Listing C.1: Benchmark automation script

```
#!/bin/bash

REDIS_IP=<IP>
REDIS_PORT=<PORT>

# Define workload mappings
declare -A workloads
workloads["WB"]="--ratio=5:95"
workloads["WA"]="--ratio=1:1"

declare -A zipf
zipf["WB"]="1.4"
zipf["WA"]="0.9"

connections_list=(10 100 200 300 400 500 600 700 800 900 1200 1500 2000)

# Start resource monitoring
./get_CPU_RAM.sh &
RESOURCE_PID=$!

for workload_key in "${!workloads[@]}"; do
  workload_ratio="${workloads[$workload_key]}"
  zipf_exp="${zipf[$workload_key]}"
  result_dir="results/$workload_key"
  mkdir -p "$result_dir/all"

  datafile="$result_dir/data.csv"
```

```
  echo "connections,run,start_time,end_time,type,ops/sec,hits/sec,misses/sec,
    avg_latency,p50_latency,p99_latency,p999_latency,KB/sec" > "$datafile"

  for connections in "${connections_list[@]}"; do
    for run in {1..3}; do
      echo "Warmup before run $run with $connections connections for workload
    $workload_key ($workload_ratio)"

      memtier_benchmark -s $REDIS_IP -p $REDIS_PORT \
         $workload_ratio \
         --key-maximum=4800000 \
         --key-pattern=Z:Z \
         --data-size=1024 \
         --key-zipf-exp=$zipf_exp \
         --clients=$((connections / 10)) \
         --threads=10 \
         --test-time=60 > /dev/null

      ALL_OUT="$result_dir/all/${connections}_run${run}.txt"
      echo "Benchmarking run $run with $connections connections for workload
    $workload_key"

      START_TIME=$(date +"%Y-%m-%d %H:%M")

      memtier_benchmark -s $REDIS_IP -p $REDIS_PORT \
        $workload_ratio \
        --key-maximum=4800000 \
        --key-pattern=Z:Z \
        --data-size=1024 \
        --key-zipf-exp=$zipf_exp \
        --clients=$((connections / 10)) \
        --test-time=180 \
        --threads=10 > "$ALL_OUT"

      END_TIME=$(date +"%Y-%m-%d %H:%M")

      awk -v connections="$connections" -v run="$run" -v start="$START_TIME" -v
    end="$END_TIME" '
        BEGIN { OFS="," }
        /^Sets|^Gets|^Totals/ {
            for (i=1; i<=NF; i++) if ($i == "---") $i = "0.00"
            print connections, run, start, end, $1, $2, $3, $4, $5, $6, $7, $8,
    $9
        }
      ' "$ALL_OUT" >> "$datafile"

    done
  done
done

kill -9 $RESOURCE_PID
echo "Benchmarking complete. Results saved in results/WA/ and results/WB/"
```

D Appendix: Git Commands for Viability Analysis

The following Git commands were used to assess development activity and contributor metrics for each Redis alternative, as described in Section 5.6.5.

Listing D.1: Count number of commits in the last 12 months.

```
git log --since=2024-05-01 --until=2025-05-01 --oneline | wc -l
```

Listing D.2: Count unique committers in the last 12 months.

```
git log --since="2024-05-01" --until="2025-05-01" --format='%aN' | sort -u | wc -l
```

Listing D.3: GitHub issue query to count closed issues in the last 12 months.

```
is:issue is:closed closed:2024-05-01..2025-05-01
```