A Comparative Analysis of Application-Level and System-Level Container Runtimes for State-of-the-Art Data Deduplication Techniques

by

AL-NAHIAN BIN SARWAR
21201519
ROHIT CHOWDHURY
21301020
SADMAN TAUFIQ MAHIN
21201759
MOHAMMED AKIB
21201760

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science and Engineering

Department of Computer Science and Engineering
Brac University
October, 2025.

Declaration

It is hereby declared that

1. The thesis submitted is my/our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Al-Nahian Bin Sarwar 21201519	Rohit Chowdhury 21301020
Sadman Taufiq Mahin 21201759	Mohammed Akib 221201760

Approval

The thesis/project titled " A Comparative Analysis of Application-Level and System-Level Container Runtimes for State-of-the-Art Data Deduplication Techniques " submitted by

1. AL-NAHIAN BIN SARWAR (21201519)
2. ROHIT CHOWDHURY (21301020)
3. SADMAN TAUFIQ MAHIN (21201759)
4. MOHAMMED AKIB (21201760)

of Summer, 2025 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on 10th October, 2025.

Examining Committee:

Supervisor:
(Member)

Dr. Jannatun Noor Mukta

Associate Professor, CSE
Director, BSDS
United International University

Thesis Coordinator:
(Member)

Dr. Md. Golam Rabiul Alam

Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Dr. Sadia Hamid Kazi

Associate Professor; Chairperson
Department of Computer Science and Engineering
Brac University

Abstract

Containerization has become a cornerstone of modern software deployment, offering lightweight isolation and rapid scalability across diverse environments. However, the growing variety of container runtimes introduces uncertainty regarding their behavior under data-intensive workloads such as deduplication, where computational efficiency and resource utilization directly affect scalability and responsiveness. To investigate this, we design a structured experimental pipeline that executes three hash-based deduplication algorithms - CRC32, MD5, and SHA-256; within three container runtimes: Docker, LXC, and Podman. Each algorithm is run ten times across datasets of 1M, 5M, and 10M records to ensure statistical consistency, generating over 3,700 performance samples consolidated into 180+ representative instances. Building on this pipeline, we develop a holistic scalability assessment framework that quantifies container efficiency through throughput trends, variability in CPU and memory usage, and collision rates, offering a comprehensive perspective on runtime behavior. Experimental findings show that Docker maintains balanced scalability with stable throughput growth through efficient daemon-managed scheduling, while LXC delivers superior computational efficiency under heavy workloads due to its direct kernel namespace access. Podman, though optimized for lightweight and security-focused tasks, demonstrates performance variability when scaled. Finally, we introduced a decision tree to assist in selecting optimal container–algorithm configurations tailored to workload requirements. This work establishes an empirical foundation for understanding container performance in deduplication contexts, providing actionable insights for building efficient and resilient cloud-native data processing infrastructures.

Keywords: MD5, SHA-256, CRC32, Data Deduplication, LXC, Docker, Podman, Containerization

Acknowledgement

We owe all we have accomplished so far to the Almighty Allah, who gave us the means, and the focus we needed to finish our study on schedule. We owe a great deal of gratitude to our supervisor Dr. Jannatun Noor Mukta for her valuable assistance and support. We would also like to extend our gratitude to Md. Farhadul Islam, Research Assistant, at BRAC University, who helped us improve the quality of our research. Finally, we would want to thank our caring parents for always being there for us and praying for us.

Table of Contents

Declaration i

Approval ii

Abstract iv

Acknowledgment v

Table of Contents vi

List of Figures viii

List of Tables ix

Nomenclature ix

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Rational of the Study or Motivation . . . 2
- 1.3 Problem Statement . . . 2
  - 1.3.1 Research Questions . . . 3
- 1.4 Objective . . . 3
- 1.5 Methodology in Brief . . . 4
- 1.6 Scopes and Challenges . . . 4
- 1.7 Contribution . . . 5
- 1.8 Thesis Organization . . . 6

2 Literature Review 7
- 2.1 Preliminaries . . . 7
  - 2.1.1 Brief History of Containers and Definition . . . 7
  - 2.1.2 Understanding Data Deduplication . . . 8
  - 2.1.3 Virtual Machines for Virtualization . . . 9
  - 2.1.4 Containers for Virtualization . . . 10
- 2.2 Review of Existing Research . . . 10

3 Environmental & Societal Impact 19

4 Research Methodology 20
- 4.1 Workflow . . . 20
- 4.2 Design Process and Evaluation Methodology Overview . . . 21

4.2.1 System's Internal Framework . . . 21
4.2.2 Performance evaluation metrics . . . 22
4.3 Model Specification with Algorithm and Container Framework . . . 24
4.3.1 Experimental Procedures . . . 24
4.3.2 Algorithmic Structure of Hash-based Deduplication Methods . 25
4.3.3 Container Architecture . . . 27

5 Experiment and Result Analysis 28
5.1 Experimental Setup . . . 28
5.2 Workload Characteristics . . . 29
5.3 Results and Analysis . . . 29
5.3.1 Outcome Overview . . . 29
5.3.2 Performance Benchmarks and Impact . . . 30
5.4 Practical Takeaways . . . 37

6 Discussion 38
6.1 Key Findings and Significance . . . 38
6.1.1 Stability Analysis . . . 38
6.1.2 Implications for Deployment Scenarios . . . 38
6.1.3 Decision Framework . . . 40
6.2 Limitations and Future Work . . . 42

7 Conclusion 44

Bibliography 49

List of Figures

2.1 Container (particularly docker) architecture [9] 8
2.2 General architecture of deduplication techniques [16] 9
2.3 Comparing various application runtime models: (a) type 1 hypervisor, (b) type 2 hypervisor, and (c) container [5]. 10

4.1 Flow chart depicting the work plan 21
4.2 Internal architecture of containerised deduplication running inside a virtual machine 22
4.3 Algorithmic flowchart of a hash based deduplication method 26
4.4 Comparative architecture of LXC, Docker, and Podman container engines used in this study and how containers interact with the guest OS and the Linux kernel. 27

5.1 Used experimental setup specification 28
5.2 Scatter plot of execution time (normalized) 32
5.3 Average memory usage by each algorithm 32
5.4 Throughput comparison by algorithm and container 33
5.5 Heatmaps for cpu time (in seconds) 34
5.6 Collision rate by algorithm 35
5.7 Scalability 36

6.1 Cpu time (sec) and memory usage (MB) across containers and algorithms for varying data scales 40
6.2 Decision tree visualization for choosing containers based on algorithm type and data scale 41

List of Tables

2.1 Table of related paper reviewed and their key findings 18

5.1 Table of average performance metrics for the workload of 1 million records 29
5.2 Table of average performance metrics for the workload of 5 million records 29
5.3 Table of average performance metrics for the workload of 10 million records 30

6.1 Our work compared to other related work 43

Chapter 1

Introduction

1.1 Background

The amount and complexity of digital data have increased to an unprecedented level due to the growing reliance on cloud-based infrastructure and the growth of data-driven applications. Modern computing ecosystems are now faced with the challenge of storing, processing, and analyzing large datasets in a way that is both scalable and resource-efficient, whether they are semi-structured logs, structured tabular records, or sensor outputs [43].

Containerization technologies like LXC, Docker, and Podman have emerged as crucial instruments for establishing lightweight, portable, and reproducible computing environments in answer to these demands [7]. Applications and their dependencies can be packaged into separate pieces using these container engines, providing uniform behavior across a range of infrastructures, from high-performance computing (HPC) clusters to personal workstations [41], [29]. Researchers and developers are now able to create easily scalable and maintainable modular, microservice-based architectures thanks to this change. At the same time, the necessity for efficient storage management strategies has increased due to the exponential development in data volume. Redundant data, frequently in the form of duplicate files or segments, is responsible for a sizable amount of storage waste in large-scale systems. And this redundancy not only leads to inefficient utilization of disk space but also increases data transfer overheads and processing latency.

To reduce these inefficiencies, data deduplication has become a popular remedy. The main concept is to recognize and save only distinct data instances, substituting references to the original for any later copies. One of the most effective and scalable techniques for this is hash-based deduplication, which creates distinct fingerprints of data blocks using cryptographic or non-cryptographic hash algorithms. In this area, algorithms like MD5 and SHA-256 have been widely utilized, while CRC32 provides a lightweight substitute for first non-duplicate data filtering [19]. Using lightweight hash-based algorithms is one useful way to lower the overhead of deduplication. While faster, lighter options like CRC32 can serve as a first-pass filter to remove obviously unique entries at a minimal cost, hash functions like MD5 and SHA-256 have long been used to create digital fingerprints of data for deduplication [42]. Redundancy can be decreased without causing a large processing burden by carefully

combining these strategies.

These deduplication approaches has been investigated in a variety of storage systems, but no impactful research on how they behave and operate in various containerization contexts has often been overlooked. Because LXC, Docker, and Podman have different architectures, especially when it comes to system call handling, process scheduling, and I/O management, these platforms could result in quantifiable differences in how well deduplication workloads execute. This is particularly important for use cases that call for reliable and repeatable performance in deployments that are resource-constrained and cloud-native.

The purpose of this study is to compare three hash-based deduplication methods that are used in LXC, Docker, and Podman containers: SHA-256, MD5, and CRC32. The study aims to assess how container architecture and deduplication performance interact by benchmarking important performance metrics like execution time, CPU and memory use, and I/O overhead. The results are meant to help choose the best container engines and deduplication techniques for data processing systems that are scalable, portable, and resource-efficient.

1.2 Rational of the Study or Motivation

Using a dataset without redundant data or sorted, clean and superior data can greatly improve computing performance all in all enhancing productivity. In cloud servers, where an immense amount of data is stored and referred to due to its massive network traffic it is crucial to choose the optimal method to allocate the hardware and software resources. Moreover, unnecessary amounts of redundant data removed from datasets also helps in reducing computation load which affects the overall energy consumption, thinking about that while reviewing papers we came across generalized deduplication, which is quite resource and computational cost friendly [6].

To evaluate the performance of modern day deduplication techniques we have to simulate the operation in a cloud environment to make fair comparisons. Modern day cloud computing includes containerised or cloud native microservices of resource allocation [18], [11]. The lightweight category virtualization like application and system-level containers such as LXC, Docker, Padman are vastly used in public and private clouds. Thus concluding to the idea of a model prototype of running hash based deduplications such as SHA 256, MD5 and CRC32 inside a containerized environment to benchmark the overall performance.

1.3 Problem Statement

The increasing need for effective data processing has led developers and system architects to concentrate more on streamlining algorithmic processes in order to lower computing load and energy consumption [4]. With its lightweight, isolated environments that are perfect for scalable deployments, containerization has become a game -changing technique in cloud computing [7]. However, when used

for computationally demanding activities like data deduplication, the fundamental performance features of several container engines—in particular, Linux Containers (LXC), Docker, and Podman — remain understudied.

Data deduplication, especially hash-based methods like SHA-256, MD5, and CRC32 plays a pivotal role in enhancing storage efficiency and reducing redundancy across applications [38]. While existing research has investigated deduplication algorithms [37], [49], there is limited understanding of how containerization choices impact their runtime performance and operational overhead. This vacancy is notable in hybrid and cloud-native environments where containers often operate atop virtualized infrastructure.

Thus, this research aims to systematically benchmark and compare LXC, Docker, and Podman containers under identical workloads executing the aforementioned deduplication techniques. The objective is to quantify variations in resource utilization, deduplication latency, and I/O performance - ultimately guiding practitioners toward container technologies that balance performance with operational efficiency.

1.3.1 Research Questions

- Among the three hash-based deduplication methods (CRC32, MD5 and SHA-256), which method is comparably better for use based on the different performance matrics in individual and overall containers?
- How do LXC, Docker, and Podman differ in their execution efficiency when performing hash-based data deduplication tasks?
- What trade-offs exist in CPU usage, memory consumption, and I/O performance among the three container platforms?
- Which container engine is best suited for deduplication-centric applications in virtualized cloud environments?
- Which containerized environment along with which deduplication algorithm is suited for which system and which kind of needs or priorities?

1.4 Objective

The primary objective of this research is to conduct a comparative analysis of three container platforms called LXC, Docker, and Podman; along with which executing three widely used hash-based data deduplication techniques: SHA-256, MD5, and CRC32. This investigation will evaluate the performance of each container platform in terms of computational overhead, I/O throughput, and system resource utilization.
To achieve this, we will:

- Set up controlled experimental environments using virtual machines provisioned via Oracle VirtualBox to eliminate host-side interference.

- Execute deduplication tasks inside LXC, Docker, and Podman containers (running inside the controlled VR environment), each running the same dataset and hash algorithm.
- Benchmark and record system metrics such as CPU usage, memory footprint, deduplication latency, and container startup overhead using standardized profiling tools.
- Compare insights which will support in making more informed decisions on container selection for performance-critical and deduplication-driven cloud workloads.

1.5 Methodology in Brief

This study conducts a comparative performance analysis of three containerization platforms — LXC, Docker, and Podman; along with which executing hash-based data deduplication techniques: SHA-256, MD5, and CRC32. The objective is to assess how container architecture influences deduplication efficiency in terms of execution overhead, resource usage, and I/O performance.

To ensure consistency and isolate host-specific variables, all containers are run inside virtual machines using Oracle VirtualBox. This nested virtualization approach provides a standardized benchmarking environment representative of hybrid and cloud-native deployments [41]. Each input dataset is divided into fixed-size blocks by the deduplication workflow segment, which then uses one of the hash functions to create a fingerprint for each block. Unique blocks are kept with their hashes, while duplicates are recognized by matching hashes and replaced with references. SHA-256 and MD5 are commonly used in secure deduplication systems and offer cryptographic robustness [19]. CRC32, while not cryptographically secure, is efficient for pre-filtering non-duplicate blocks and reducing overhead [49]. Each technique is implemented and tested across all three platforms under identical hardware and dataset conditions. Performance metrics like the - CPU load, memory consumption, deduplication time, container startup latency, and I/O throughput are collected using standard profiling tools.

In addition to highlighting performance issues and trade-offs in containerized deduplication processes, this methodological framework offers recommendations for the best containerization platform to use for storage-efficient applications in cloud-native or restricted environments.

1.6 Scopes and Challenges

There is always room for improvement, and same goes for our research and experiment here too. As there are some problems in each of the deduplication algorithms that are being used in the experiment (like MD5, SHA-256 and CRC32), there are also some strong points to each of them. It's more of a like a trade of between different properties in the algorithms but there is no universal best or always use

method out there, so this remains as a scope for further future development to make better deduplication algorithm. The challenges and scopes we have see are -

- The three algorithms each have trade of between speed, collision rate and suitability where CRC32 has best throughput and speed of performing deduplication but has the worst collision resistance and security, on the other hand SHA-256 is best is security and collision resistance but lacks speed and throughput. And the MD5 is the middle performer but it also needs more improvement.
- The overhead on the execution environment for container is high, so more research is needed on lowering overhead and making containers more flexible like virtual machine.
- The cost of running applications should be as low as possible in terms of resources, for that is why we are trying to shift from VM's to containers in the first place.
- There is a need of further improvement in strong isolation between multiple applications that are running in the same OS in containers so that no kinds of malicious, compromised or attack can be propagated on other applications. There should be Opaqueness (one application is not aware of other applications running in the same OS).

1.7 Contribution

This research presents key contributions toward achieving the goals which are -

- We have conducted a comparative performance analysis of three container platforms: LXC (system level), Docker and Podman (application level); also three data deduplication techniques: SHA-256, MD5, and CRC32 by evaluating their efficiency in terms of computational overhead, IO throughput, and system resource utilization.
- Most of the similar container performance evaluation papers used applications-layer protocol or database management systems like HTTP, MySQL etc. services for benchmarking, however, we have presented an algorithm-based benchmarking where we used three deduplication algorithms to be run into our system while we observe our resource usage to benchmark the containers, which makes our work novel.
- We have used container on VM as benchmarking environment to simulate a hybrid and cloud native workspace which is widely used now-a-days.
- We have provided empirical insights into performance trade-offs between application and system level containers as well as provided a comprehensive comparison between the three deduplication algorithms.
- We have structured a decision tree for selecting the appropriate container engine and deduplication algorithm based on criteria such as data scale, speed priority, memory constraints, and collision sensitivity; thereby supporting practitioners or researchers who are working with containers or deduplication algorithms.

- Through our paper, we have shown how CRC32 to be more suited as a pre-filter deduplication due to its non-zero collision rates despite high throughput, while also demonstrating MD5 and SHA-256 to be more relievable due to its guaranteed data integrity even though its slow nature when it comes to data deduplication.

1.8 Thesis Organization

The upcoming part of this thesis has been constructed as follows:

- In the first part of chapter 2, the background information is given on various architecture of virtualization like virtual machines (VM), containers; along with understanding of data deduplication and containers. After that, we discuss the different papers which focuses on containers performance and deduplication methods.
- Next, chapter 3 section talks about how our research impacts society and environment.
- chapter 4 includes workflow and details about internal frameworks for the algorithms and containers that we used, as well as the performance metrics based on which we are benchmarking the systems.
- chapter 5, we provide the results of our experiments as well as the experimental process we utilize.
- In chapter 6, we go through the main takeaways and findings that we made from our experiments and provided a decision tree on selecting containers and deduplication algorithms as we need.
- With chapter 7, we wrapped up the whole research with the conclusion.

Chapter 2

Literature Review

2.1 Preliminaries

The exponential growth of digital data has created significant challenges in terms of storage efficiency, especially in cloud and containerized computing environments. This raises significant challenges in efficient data storage, processing and management particularly within cloud-based and containerized computing environments. This chapter provides the foundational concept necessary to understand the scope and methodology of this thesis. It begins with an overview of software containerization including its origins, core principles, and widespread adoption.

2.1.1 Brief History of Containers and Definition

Software containerization was initially introduced in the FreeBSD operating system in 2000 to enhance the portability and usability of computational tools. In 2010 companies like Docker carried out the widespread use of container adoptions [1] . It enabled software to run in an isolated environment which limited interaction with the underlying host system [12] . The application sees a file system separate from the host. As a result, it allows the container to be packaged with all necessary dependencies, making it transferable across various computing environments such as personal workstations or high-performance computing clusters, all of this without the requirement of modification of the code or additional installations. Typically, users download a container image that encapsulates the application, its dependencies and any required resources. The container engines also referred to as container runtimes execute these images, which initiates the container as a running instance [41] . Thus, a user's requirement is to install a single container engine to launch multiple container images, each image can contain a distinct set of software tools. Container images become highly portable and can be readily shared over standard network connections due to this architecture. As a result, containerization has been widely adopted by both users and developers, particularly in cloud computing. Figure 2.1, showcases the architecture of Docker container which is similar for almost all the contains, where client sends the container images to Daemon (container server), depending on these images downloads required libraries, drivers and binaries, which ultimately creates individual and isolated containerized environments.

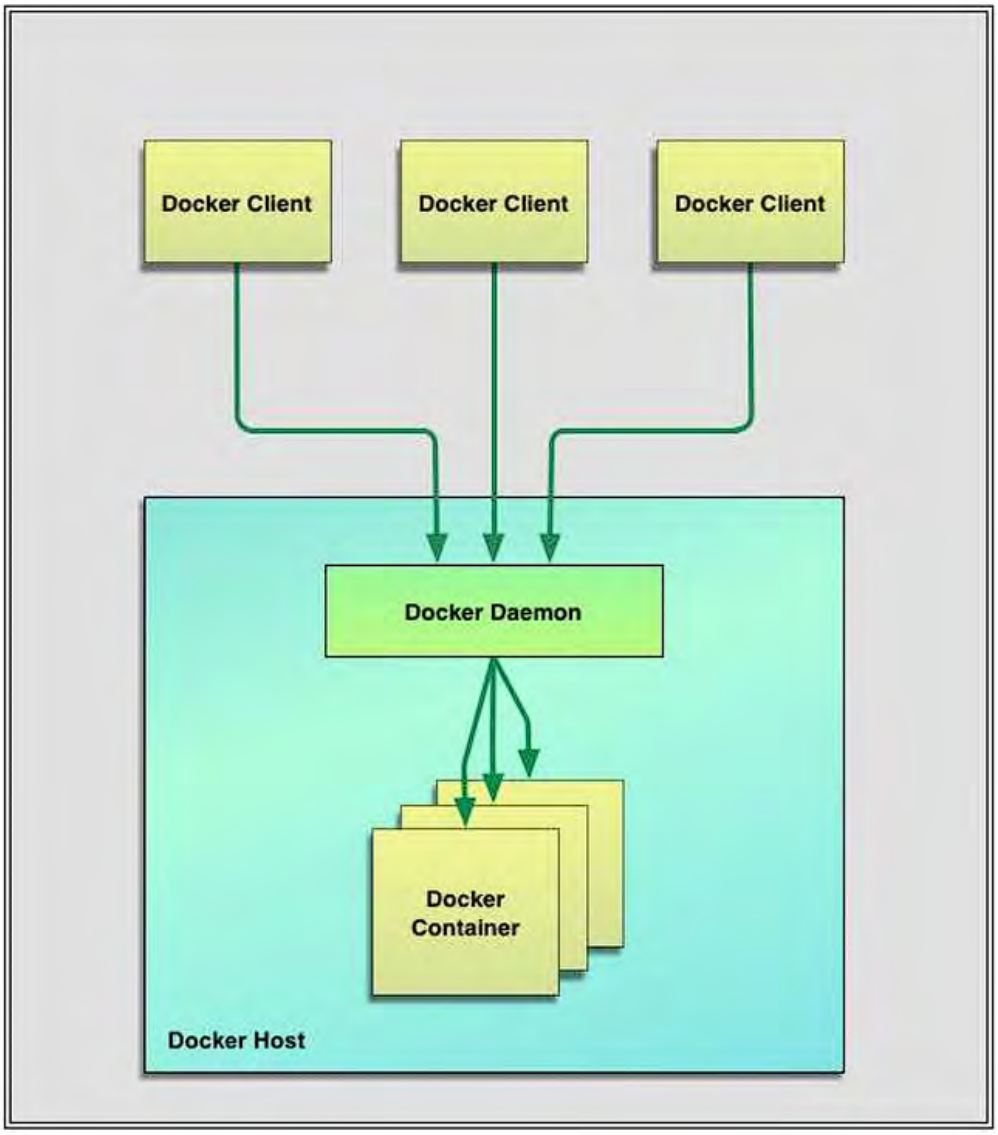


Figure 2.1: Container (particularly docker) architecture [9]

2.1.2 Understanding Data Deduplication

Data Deduplication is the concept of the breaking a large sum of data into chunks and then match them with other data to keep only unique data and only one instance of that data [45]. It uses cryptographically secure hash-based fingerprints of the files or chunks and then identifies the duplicates by matching their fingerprints [16]. This technique is basically a special compression technique to reduce redundancy and network transmission rate for cloud big data [8]. This technique first came up for implementing in cloud storage and data centers where huge amount of data resides and there can be multiple copy of same data present. So, to store data more efficiently, it is a very important technique. Now-a-days, this technique has been modified for many ways to be implemented in many places including real-time data too. Hash algorithms such as MD5, SHA-256 successor of SHA-1 and Rabin fingerprinting are executed to detect similar fingerprints to reference or point it towards the original location. On the other hand, write down unique ones into disk and its index is recorded [19]. Another commonly used hash function is the Cyclic

redundancy check or CRC32 to filter the non-duplicate data which utilizes 32bit hash values. While CRC32 is not cryptographically secure, it is highly effective in quickly identifying non-duplicate data blocks [24]. All three techniques provide a potential scope to study the performance tradeoffs while executing in containerized environments. The following Figure 2.2 shows a brief idea of how a deduplication algorithm is generally structured to identify unique data from redundant data using chunking and fingerprinting.

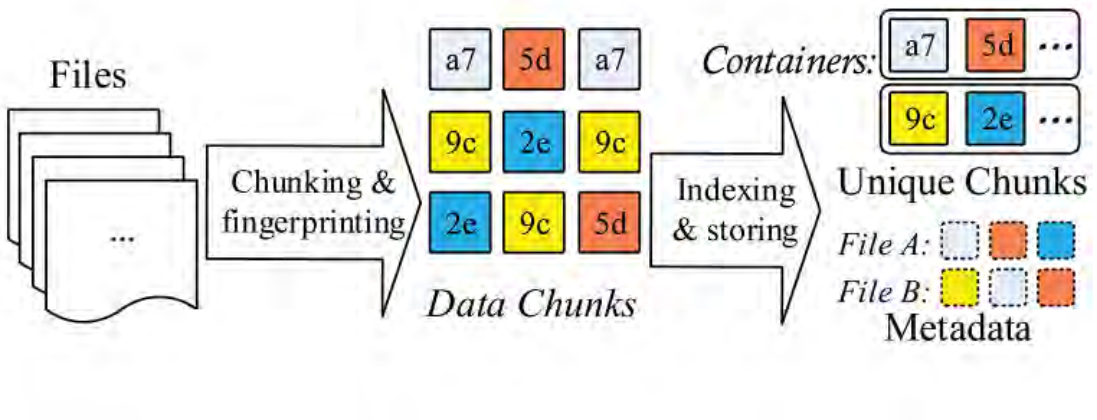


Figure 2.2: General architecture of deduplication techniques [16]

2.1.3 Virtual Machines for Virtualization

Virtual machines (VMs) are a foundational technology in cloud computing that enable multiple isolated operating system instances to coexist on a single physical host. Each virtual machine abstracts the underlying hardware through a hypervisor, which allocates and manages hardware resources such as CPU cycles, memory blocks, storage, and network interfaces among guest operating systems. This abstraction layer ensures that each VM operates as an independent entity, capable of running its own kernel and system libraries [5]

There are two primary types of hypervisors:

- Type 1 (Bare-metal): Runs directly on physical hardware, providing superior performance and isolation (e.g., VMware ESXi, Microsoft Hyper-V).
- Type 2 (Hosted): Runs on a conventional OS, offering ease of deployment at the cost of performance overhead (e.g., VirtualBox, VMware Workstation).

Each VM carries its own OS kernel, leading to greater resource isolation but also increased redundancy and startup overhead. In large-scale cloud infrastructures, such as those powered by OpenStack or VMware vSphere, hypervisors coordinate resource scheduling, migration, and fault tolerance across distributed clusters [25]. Although VMs provide strong isolation and hardware-level abstraction, the duplication of OS kernels and the emulation of hardware devices make them relatively resource-intensive. This inefficiency paved the way for a more lightweight form of virtualization called containers.

2.1.4 Containers for Virtualization

Containers are a modern virtualization approach that isolates applications at the operating system level, rather than the hardware level. Unlike virtual machines, containers share the same kernel of the host operating system, which drastically reduces overhead and startup time [22].
They achieve isolation using kernel features such as namespaces, control groups (cgroups), and union file systems (OverlayFS, AUFS, etc.). These mechanisms allow containers to maintain independent process spaces, resource quotas, and file system layers while still leveraging the host's kernel.

By using a single shared kernel, containers achieve near-native performance and excellent scalability, making them ideal for microservices, CI/CD pipelines, and serverless computing.
In platforms such as Docker or Podman, images are built as layered file systems, where each layer represents incremental changes. These layers are cached and reused across containers, improving storage efficiency [49].

LXC (Linux Containers), a lower-level implementation, provides a more system-oriented environment resembling traditional Linux systems, making it suitable for system containers. Conversely, Docker and Podman focus on application containers, emphasizing portability and developer-centric workflows [22].

Conceptually, containerization can be visualized as lightweight, modular units running atop a single OS kernel compared to VMs, which replicate entire OS stacks. This architectural efficiency makes containers essential in cloud-native computing, Kubernetes orchestration, and hybrid cloud deployments. Figure 2.3, showcases the 2 types of hypervisors and the containers for visual aids.

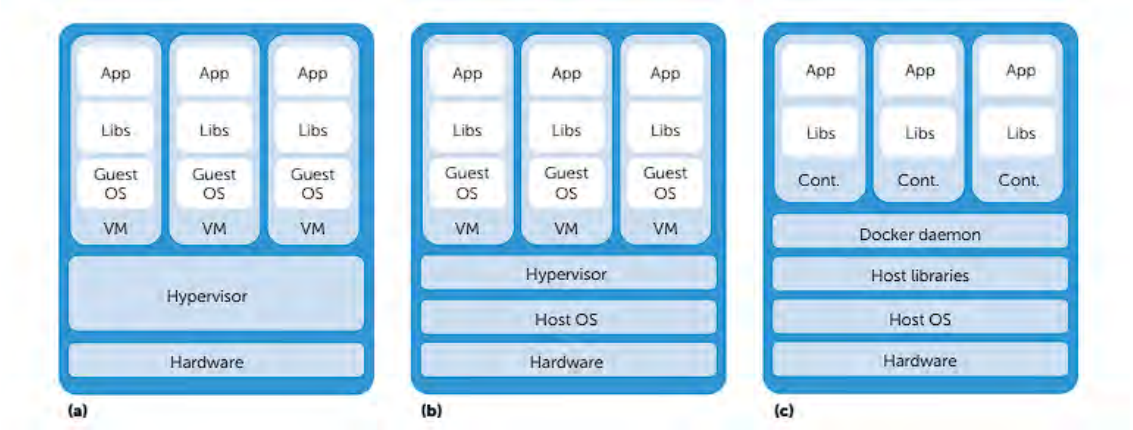


Figure 2.3: Comparing various application runtime models: (a) type 1 hypervisor, (b) type 2 hypervisor, and (c) container [5].

2.2 Review of Existing Research

Firstly, Zhang et al. [21] provides a systematic comparative evaluation of virtual machine (VM)-based and container-based infrastructure for big data environments with a focus on Apache Spark workloads. Spurred on by the increasing popularity of

containers as light and easy-to-deploy alternatives to traditional VMs, the authors study three key dimensions: deployment simplicity, bootup performance, and the performance and scalability of representative Spark workloads like K-means, Logistic Regression, PageRank, and SQL Join. Experimental results from a five-node physical cluster consistently demonstrate that containers are more efficient to deploy, reducing total setup time for a Spark cluster by half compared to VMs, mainly due to sharing and stacking image structures. Containers also show bootup times significantly faster and deployment density: e.g., booting 256 containers is over 50 times faster than an equivalent number of VMs. Containers outperform VMs in the performance test by recording less runtime overhead and providing greater scalability, especially with an increase in cluster size. Spark jobs on container environments record less execution time and better CPU and memory usage, whereas big VM clusters fail due to resource conflicts. The paper concludes that containers are overall more helpful for big data workloads in ease of management, performance, and scalability and provides valuable contributions to researchers and practitioners designing large-scale data processing systems.

According to Vestergaard et al. [27], [28] explained Generalized Deduplication (GD) as a lossless compression technique tailored for large volumes of small IoT data. Unlike conventional deduplication methods that struggle with near-identical data, GD effectively compresses such data by leveraging a chunk-based decomposition approach. In this method, data chunks are decomposed into base and deviation components, where redundancy is eliminated by storing only one base copy and replacing duplicates with pointers. If two chunks are similar but not identical, they can be represented using a single base and different deviation values, thereby significantly reducing storage overhead. By applying deduplication to the base components, GD optimizes data storage efficiency without losing information integrity. This approach is particularly beneficial in IoT environments, where massive volumes of sensor data generate substantial redundant information. The study highlights GD's potential in enhancing storage optimization and transmission efficiency, making it a valuable technique for resource-constrained IoT systems.

Potdar et al. [31] tested the performance of Docker containers and virtual machines (VMs) using a variety of tests such as Sysbench, Phoronix, and Apache Benchmark to examine CPU, memory, storage I/O, load handling, and speed of operation. The study concludes that Docker containers are better performing than VMs across all categories of tests due to their lightweight architecture devoid of the overhead of running individual guest operating systems found in VMs. In CPU-intensive operations like prime number computation and compression of files, Docker completes operations significantly quicker than VMs. Memory throughput tests also reflect Docker to be more efficient. Disk read/write access benchmarking of Docker indicates its disk access to be roughly twice as fast as VMs. Load testing with Apache Benchmark proves Docker to be able to serve more requests per second than VMs because of lower network latency. Operational speed benchmarks with computational loads also gain Docker faster speeds of execution. Statistical t-tests analysis indicates such performance deficiencies are quite large. The paper concludes Docker containerization offers a faster, more efficient virtualization technology with better resource allocation than conventional VM setups, making it particularly suitable

to cloud and microservices-based deployment. Future areas of work include Docker container scheduler optimization and enhancing container security to protect against existing vulnerabilities.

Verma et al. [48] compares the performance of microservices deployed on Docker containers and virtual machines (VMs) within the Amazon AWS cloud platform, analyzing important metrics such as throughput, response time, and CPU utilization under various deployment configurations. In contrast to typical assumptions that containers will perform better than VMs, the study determines VM-based web services to always be superior performers compared to container-based services deployed on AWS. Specifically, VMs achieve up to 125% response times better than containers for the same workloads. Three scenarios of experiments are included in the research simulating increasing numbers of web services being run on AWS EC2 (VM) and ECS (container) platforms. In all cases, VMs have higher throughput and lower response latency, with CPU utilization reaching earlier peaks for containers, indicating faster resource saturation. These gains in performance are primarily the result of AWS's architecture, wherein only ECS containers run inside EC2 VMs, adding an additional layer of virtualization that is subject to overhead and reduced performance over running containers directly on hardware. Authors recommend the use of EC2 VMs for deployment of high-performance applications instead of ECS containers. They further recommend conducting performance studies on other cloud platforms like Google Cloud and Microsoft Azure in subsequent work. This paper disproves the long-held wisdom concerning the dominance of containers in cloud-native deployments on the basis of deployment environment impacts on actual performance, hence informing cloud architecture and microservice deployment decisions.

Göttel et al. [30] introduce Hermes, a Generalized Deduplication (GD) framework designed to improve computational power efficiency in IoT-based Digital Twin (DT) systems. Unlike traditional deduplication methods that focus solely on exact redundancy elimination, Hermes is capable of identifying and eliminating similar, but not necessarily identical, data chunks, making it particularly effective for time-series IoT applications. The study demonstrated that Generalized Deduplication (GD) reduces the CPU workload by 26%, resulting in significant energy savings in IoT-edge computing. Additionally, transmission costs decrease by 43% as GD minimizes redundant data transfer between IoT nodes and the cloud, optimizing network bandwidth usage. Furthermore, the framework enhances DT synchronization, ensuring real-time updates with minimal resource consumption. The authors emphasized Hermes' suitability for large-scale IoT deployments, particularly in smart cities and industrial automation, where reducing the network and computational loads is essential for system efficiency. However, the study also highlights the complexity introduced by pre-processing requirements for similarity detection, suggesting the need for hybrid deduplication models to facilitate real-world implementations.

In a recent study, Kathiravan et al. [40] proposes an enhanced cloud-based duplicate elimination and file management system based on the MD5 algorithm to efficiently identify and eliminate duplicate files from cloud storage. Having understood the huge expenses and administrative burden caused by duplicate data, research in this

paper focuses on optimizing deduplication operations for decreased storage requirements and improved backup performance without interfering with users to a large extent. The system utilizes MD5 hashing to create unique digital signatures on files to facilitate speedy duplicate identification. Files that are being uploaded are searched against stored hashes within a hashed mapping table, whereby the system is able to reject duplicates before they are stored. The system has secure ownership validation and accommodates multiple forms of deduplication, including file-level, block-level, and local vs. global deduplication. Experimental outcomes indicate that the system can significantly reduce storage space and accelerate duplicate elimination, and therefore it is beneficial for big data cloud environments. An admin interface for managing user files and accounts, and horizontal scaling functionality on cloud platforms are provided. The authors observe that MD5's efficiency cannot be disputed with acknowledged vulnerabilities in cryptography, since it still performs well with checksums in deduplication scenarios. Future work involves adding encryption for enhanced file security and user-controlled access to decryption. In general, the paper provides a feasible, scalable solution to cloud storage deduplication that solves both performance and security issues and simplifies administrative burden.

Here, Hu et al. [24] propose D-pblk, a novel deduplication approach designed for Open-Channel SSDs, which are often used in data-intensive environments. Recognizing the limitations of performing complex hash computations like SHA-1 and MD5 within SSDs due to hardware constraints, the authors offload fingerprint computation to modern host CPUs capable of high-speed parallel processing. Their method incorporates a two-step fingerprinting process. Initially, a fast CRC32 hash filters out unique data, and only likely duplicates are verified using the more compute-intensive SHA-1 hash. To address the challenge of storing deduplicated data that may not align with NAND flash page sizes, the system employs two ring buffers—one for collecting incoming data and another for managing non-duplicate content before it's written to disk. The framework was tested using FEMU, a flash emulator, with datasets including GCC, APACHE, and WIKI. Results showed that D-pblk achieved deduplication rates between 4.61% and 31.63%, and reduced write latency by up to 29%, without significantly affecting read performance. The research highlights the effectiveness of combining lightweight hashing, multithreading, and efficient metadata handling to optimize both SSD performance and lifespan.

Futhermore, Silva et al. [47] work is a sound recent comparative analysis of Docker and LXD (LXC) container technology across their performance axes with respect to native and virtualized environments. The authors state the shortcomings of traditional hypervisor-based virtualization performance overhead and scalability for models like microservices that are accelerated to be deployed to introduce container-based virtualization as a developing solution. Docker as an application container and LXD as a system container are contrasted in multi-dimensional benchmark environments testing CPU, disk, and network performance on Windows as well as Linux platforms. Empirical evidence indicates that container platforms impose quite modest overall overhead compared to native execution, with Docker on Windows having measurable loss of multi-core CPU performance (ca. 30%), but Docker on Linux has only a very modest negative impact and even some performance equivalence in very highly parallel workloads. LXD on Linux was shown to either match or slightly

better native performance, particularly for multi-core workloads, which suggests its reliability and feasibility for heavy development operations. IO and network testing also confirms that both Docker and LXD have limited overhead, with the exception of write speed potentially being affected depending on platform and storage configuration suggesting use-by-use analysis for IO-intensive applications. In short, both container solutions are highly mature and scalable, with LXD on Linux consistently yielding the best and most stable result, particularly for performance-critical or multi-process use cases. Linux-based containerization is what is recommended by the authors for the optimal result but mention that technology choice should be determined by hardware, workload, and use case specifics.

Leekha and Shaikh [34] presents a scalable deduplication approach for cloud storage content through a 64-bit architecture-based SHA-256 implementation to address the growing issue of redundancy from heterogeneous sources such as text, images, and videos. The approach operates on file-level as well as block-level data, enhancing detection through a sliding window protocol, variable chunk size, and specific focus on localizing changes instead of mere copy detection counteracting manipulation types like splicing, rotation, morphing, and copy-move attacks. The system introduces pre-processes information, then generates hashes through MD5, standard SHA-256, and the new 64-bit version of SHA-256 for increased collision resistance and efficiency. Extensive testing on numerous public datasets MICC, CoMoFoD, Coverage for images and REWIND, VTD for video is demonstrated to indicate that the new hashing technique improves processing efficiency and scalability for deduplication tasks. Processing time is demonstrated to be decreased with good recall/precision in localization, specifically forgery/manipulation detection in multimedia content. The use of Hamming distance also optimizes computation and storage. Ultimately, the paper demonstrates how the combination of 64-bit SHA-256 and sliding window chunking achieves effective and fast deduplication and localization to facilitate real-time applications and cloud infrastructures on a big scale. The technique can be extended to audio and encrypted data, authors believe, pointing to the growing necessity for adaptive deduplication as digital data keeps expanding.

Moreover, Felter et al. [10] article presents a comprehensive performance comparison of traditional virtual machines (VMs) with KVM and Linux containers under the control of Docker, with an emphasis on CPU, memory, networking, IO, and real application-relevant workloads in the context of the cloud. The study finds containers to generally compete or outperform VMs in nearly all test environments on the basis of their lightness and sharing of host OS kernel, which avoids the overhead of running full guest operating systems required in VMs. Microbenchmarks reveal CPU and memory overhead from both virtualization methods to be inconsequential, but differences emerge in IO and networking. Docker containers have minimal or zero latency and throughput overhead for network and storage IO, whereas VMs have higher latency, lower IO throughput, and higher CPU overhead, especially for small-sized IO operations. Docker's use of NAT and layered file systems (AUFS) does impose some performance overhead that may be eliminated by the activation of native networking and volumes. Real application benchmarks demonstrate that containerized Redis and MySQL almost match the native Linux, while VMs introduce additional overhead, especially with network latency impacting throughput

under low concurrency. The article points out that VMs offer higher isolation and OS flexibility for multi-OS scenarios but at performance expense, whereas containers provide deployment at speed and resource efficiency as a good solution for cloud and PaaS models. The authors address practical trade-offs such as VM tuning sophistication against Docker's simplicity and suggest case-by-case workload-to-workload comparison in choosing between VMs and containers. The article in general stresses the growing legitimacy of containers as a fast and scalable alternative to traditional VM-based virtualization in cloud infrastructure.

After that, Bernstein [7] provides a comprehensive history of container technology evolution and its role in cloud computing infrastructure, emphasizing its advantages over traditional hypervisor-based virtualization for scalability, deployment speed, and resource efficiency. The author traces the evolution of containers from Unix chroot and Solaris zones to Linux Containers (LXC) and modern Docker containers, noting how containers share the host OS kernel, thus being smaller and faster deployments than VMs requiring separate OS instances. The article describes Docker's layered image model and automation using Dockerfiles, which make application packaging and deployment easier. It also discusses Kubernetes, Google's open-source cluster manager, which orchestrates container deployment, networking, and scaling in large distributed systems, extending container utility beyond single hosts. Kubernetes' novel IP allocation for each container pod simplifies networking across clusters. Bernstein analogizes containers with hypervisors in cloud setups, noting that companies use hypervisors like VMware ESXi or KVM for multi-OS support and improved isolation, whereas public clouds like Google and IBM SoftLayer utilize containers for improved efficiency. The paper identifies security concerns with containers and mentions that for high-security needs, VMs may be preferred or containers run within VMs for the sake of isolation. Finally, it highlights the industry momentum on container standardization and orchestration, positioning containers, Docker, and Kubernetes as cornerstones of next-generation cloud computing architectures, with the potential for quicker, more streamlined, and scalable application deployment environments. This text provides forward-looking context on the revolutionary impact of containers on cloud infrastructure design and deployment models.

Timčenko, V., Lazić, M., Đorđević, B. and Davidović, N. [35] compares the performance of two popular container engines, Docker and Podman, under carefully monitored test settings. In order to assess disc performance under four simulated workloads webserver, varmail, fileserver, and random file access—the authors use Filebench as the benchmarking tool and CentOS 7 with an XFS filesystem as the testing environment. The study's main finding is that Podman performs better than Docker on all workloads, especially when there are several containers running. With a single container, the performance decrease is negligible, but as the number of container instances rises, it becomes more noticeable. The reason for this discrepancy is that Podman runs daemon-less and makes more direct use of runC, which results in lower overhead, while Docker's architecture depends on a long-running daemon. A mathematical model that deconstructs file system latency and overhead components supports the study's findings. The authors also come to the conclusion that Docker and Podman only slightly increase overhead when compared to the native system, confirming the effectiveness of container-based virtualisation. However, Podman is

a better option because to its rootless operation and reduced performance degradation, especially in settings where system resources and security are an issue.

Another paper by Malviya et al. [36] compares four of the top container orchestration tools in cloud computing: Kubernetes, Docker Swarm, Apache Mesos, and Red Hat OpenShift. It contrasts these tools on parameters such as security, deployment, stability, scalability, cluster installation complexity, and learning curve. Kubernetes is identified as the most widely used platform, with praise for its robust pod scheduling, huge tool ecosystem, and ability to orchestrate container clusters at large scale. Docker Swarm is noted as being simple to use and appropriate for less demanding applications, offering native Docker integration and simple cluster management. Apache Mesos is noted as being extremely scalable, having very large clusters, and supporting heterogeneous workloads outside containers, such as big data processing. OpenShift, based on Kubernetes, notes advanced security policies, enterprise-class support, and hybrid cloud deployment. The article highlights the compromises among these platforms: Kubernetes for scale and flexibility, Docker Swarm for ease of use, Mesos for distributed processing at scale, and OpenShift for enterprise readiness and security. It concludes that the choice of an orchestrator depends on organizational needs, application complexity, and trade-off between scalability, security, and ease of use. The study also suggests possible future research on developer-friendly development of these tools and investigating other orchestration tools for deployment quality improvements. This detailed comparison assists practitioners in selecting appropriate container orchestration technologies based on their cloud infrastructure needs.

One of the most thorough experimental investigations of container performance on virtual machines (VMs) in the current year is presented by Aqasizade et al. [49]. He experimentally compares the performance of containers run on top of virtual machines (VMs) with bare-metal containers and VMs on CPU, memory, disk, and network resources, on well-known platforms including Docker, LXC, Podman, KVM, and Xen. Twelve configurations were tested on an actual setup using benchmarks such as 7Zip (CPU), STREAM (memory), IOzone (disk), Netperf (network), and Sysbench (MySQL database). The study illustrates that native machines offer optimal CPU and memory performance, with containers, specifically Podman and Docker, being close seconds. For container-on-VM configurations, Docker on KVM and LXC on Xen offer optimal CPU and memory performance, respectively. The disk I/O performance is superior in Xen-based VMs and Xen VM-based containers due to paravirtualization and hardware optimizations. Network performance is largely comparable across all configurations with the exception of those utilizing Podman, which is constrained by network interface problems. MySQL benchmarks demonstrate that Xen-based configurations are more performant than others with exponential enhancements in transactions per second as the number of threads rises. The findings affirm that while containers have lower overheads than VMs, running containers on VMs has an additional layer of virtualization overhead causing some loss of performance, though with increased isolation and management benefits. The paper concludes that virtualization configuration is a function of individual workload demands, and it determines the optimal container and VM combinations for a given set of resource requirements. It provides actionable guidance for cloud archi-

tects to deploy efficient and scalable systems and suggests further research in the field of container runtime performance in Kubernetes clusters.

Author	Containers Used	Workload/algo	Results/Findings	Limitations
Kathiravan et al. [40]	-	MD5	Cloud storage using MD5 hashing technique saves bandwidth and storage.	Several security flaws, stronger encryption required
Silva et al. [47]	Docker and LXD	Geekbench6, CrystalDiskMark	Docker and LXD's capability to meet the demands of contemporary virtualisation is tested	The performance results are based on specific hardware and software platforms
Leekha et al. [34]	-	Modified SHA256	SHA-256 (64-bit) hashing provides secure deduplication with better scalability and faster processing than MD5, SHA-256 (standard), and SHA-512 for text, images, and video data.	Evaluated on select public datasets; effectiveness may vary with other data types or in real-world cloud settings.
Aqasizade et al. [49]	Docker, LXC, Podman Containers	MySQL	Containers on VMs deliver competitive performance compared to standalone VMs, with Docker on KVM excelling in CPU, LXC on Xen in memory, Docker or LXC on KVM in networking, and Xen-based containers in disk IO.	Tested on specific public datasets; effectiveness may vary with other data or real-world cloud environments.
Bernstein et al. [7]	Docker and LXC	-	Docker enhances container deployment ease by building on LXC with a layered image system and standardized API.	Container security isolation is weaker than VMs, so combining containers with VM isolation is recommended for sensitive workloads.
Zhang et al. [21]	Docker	Hadoop	Containers achieve higher CPU and memory efficiency, showing less CPU wait time and more flexible memory use than VMs.	Only focuses on CPU and Memory
Felter et al. [10]	Docker	Redis, MySQL	Real-world tests using Redis and MySQL demonstrate containerized deployments offer near-native throughput and latency, whereas KVM VMs show greater overhead, notably in network-sensitive tasks.	The impact of nested virtualization (containers inside VMs or vice versa) was not included due to expected overheads.
Potdar et al. [31]	Docker	standard benchmark tools such as Sysbench, Phoronix, and Apache benchmark	RAM speed benchmarks reveal Docker containers have higher memory throughput than VMs, and load testing with Apache Benchmark indicates Docker manages more requests per second with lower network latency compared to VMs.	Doesn't assess performance in multi-host or real-world cloud environments using orchestration tools like Kubernetes.
Malviya et al. [36]	Kubernetes, Docker Swarm, Mesos, and Redhat OpenShift	-	Suggests orchestration tool depends on factors; user expertise, security needs, cluster scale, and application complexity.	Focuses on feature comparison and qualitative assessments, with limited empirical benchmarking of orchestration tools under varied workloads.
Bystrov et al. [33]	Docker containers of the OpenStack cloud	-	The virtualization layer (Docker or KVM) reduced the computational performance by a factor of approximately 2, compared to native hardware.	Only the performance of the DEM application is evaluated and other scientific or data processing applications are missing
Verma et al. [48]	Amazon EC2 and Docker	HTTP	Microservices deployed on AWS EC2 VMs demonstrate up to 125% better response time compared to those deployed using Docker containers on AWS EC2 Container Service (ECS).	Only Apache HTTP Server web services were tested; other microservices or workloads may perform differently.

Table 2.1: Table of related paper reviewed and their key findings

Chapter 3

Environmental & Societal Impact

Through this research, we aim to contribute to sustainable and energy-efficient computing by enabling better decision-making in container runtime selection for data-intensive workloads. By identifying container–algorithm combinations that deliver higher throughput with lower CPU and memory utilization, we promote resource-efficient configurations that help reduce power consumption and the environmental footprint of cloud infrastructures.

In practical scenarios where response time is critical or resources are limited—such as healthcare data systems, financial platforms, and edge-computing environments—our findings provide valuable guidance for deploying containers that maintain stability and responsiveness under constrained conditions.

From a societal perspective, we believe this work supports the creation of more sustainable and accessible digital infrastructures. By improving computational efficiency and promoting responsible cloud-native deployment practices, we contribute toward a greener and more equitable technological ecosystem.

Chapter 4

Research Methodology

This section provides a structured overview of the design methodology employed to implement and evaluate deduplication techniques within containerized environments. The goal is to ensure consistency across testing, reproducibility of results, and modular adaptability of the approach.

4.1 Workflow

The process initially starts with the loading of the input workload in CSV format. This step, which primarily involves file I/O operations, ensures that the raw data is made available to the system for subsequent processing. Efficient data loading is critical, as it directly affects the responsiveness of the system when handling large-scale datasets. Once the dataset has been loaded, it is partitioned into smaller data-blocks. Partitioning serves the dual purpose of reducing memory overhead and facilitating more efficient computation, as operations can be applied block-wise. This stage, also dominated by I/O activity, provides the structural foundation for the subsequent computational phase. Following partitioning, each data-block is processed by applying a hash function. The objective here is to compute the hash value for every block, thereby enabling later analysis of deduplication or integrity-checking performance. Unlike the preceding steps, this phase is CPU-intensive, as hashing operations require significant computational effort. Consequently, the performance of this stage is strongly dependent on both the efficiency of the selected hashing algorithm and the computational resources available. The system then begins recording key performance metrics, including CPU usage, execution time, and other relevant resource consumption statistics. This measurement phase is essential, as it provides the empirical evidence required for a fair comparison across different configurations and test conditions. To enhance the reliability and validity of the results, each configuration is executed for an epoch of ten times. Repetition ensures that the collected data captures consistent trends and minimizes the effect of transient system fluctuations. This approach strengthens the statistical robustness of the experimental findings. Finally, the collected metrics are subjected to systematic analysis. This concluding stage integrates the performance data and derives insights regarding the efficiency, trade-offs, and limitations of the approach under study. The workflow formally concludes after this analytical phase, providing the foundation for discussion in the results and evaluation chapters. The visual representation is shown in the Figure 4.1 (Flow Chart Depicting the work plan).

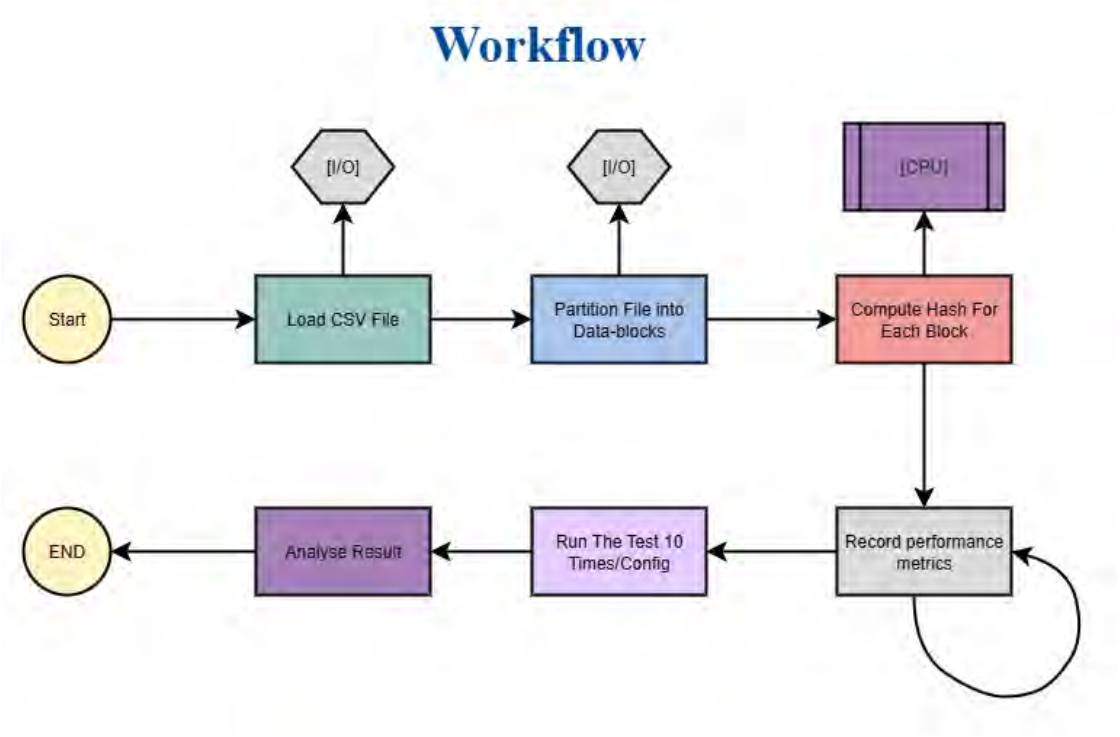


Figure 4.1: Flow chart depicting the work plan

4.2 Design Process and Evaluation Methodology Overview

The deduplication pipeline is designed as a modular Python script where the hash function can be dynamically selected. This allows consistent testing across different algorithms while maintaining uniform logic for file reading, hashing, and duplicate detection.

4.2.1 System's Internal Framework

In this study, we are focusing on performance evaluation of different containers for a particular type of deduplication algorithm. And as in recent times, more and more people are interested and relying on cloud infrastructure for running models or business, so it is more likely that the deduplication algorithm which we are working with will be ran on cloud environment. And based on this concept, we are performing our tests in an environment simulating the cloud servers. In cloud computing, virtual machines (VMs) serve as the foundation for virtualization that provide computing power, enabling cloud providers to share their physical hardware resources from a single physical server with the illusion of dividing into multiple virtual servers, each acting as an independent computer with its own operating system and applications distributed among multiple customers [2], [52]. And this server virtualization that simulate partitioning a physical server into multiple virtual servers, is achieved with the help of specialized software called "Hypervisor" or "Virtual Machine Monitor (VMM)"[50], [3]. So, we also downloaded a hypervisor software to create a smaller sized computer while allocating resources to it inside

our testing computer. And inside that hypervisor created small computer we tested the hash based deduplication techniques– SHA-256, MD5 and CRC32 which were applied inside containerization platforms: LXC, Docker and Podman. To see the visualization of our framework, please see the Figure 4.2 as a reference.

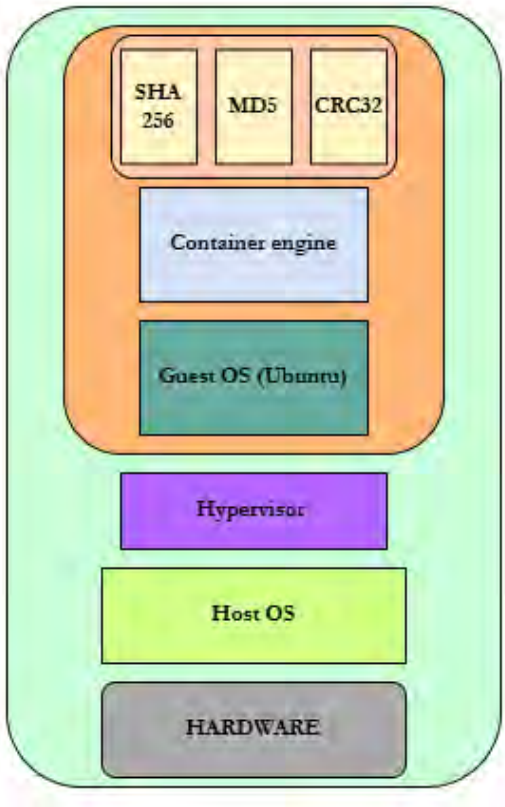


Figure 4.2: Internal architecture of containerised deduplication running inside a virtual machine

4.2.2 Performance evaluation metrics

The evaluation focuses on six key performance indicators to comprehensively assess container behavior during deduplication:

- **CPU Utilization:** Captures average and peak processor utilization through the time the algorithm needs to stay in the cpu or processor for processing. To do this, there is dedicated python library called *time* which comes with a class method called *process_time*() that we used.

$$T_{\text{CPU}} = T_{\text{user}} + T_{\text{system}} \tag{4.1}$$

Where:

T_{user} = time spent executing code in user space.

T_{system} = time spent in the OS kernel on behalf of your process (e.g., file I/O, memory allocation, system calls).

- **Execution Time:** Measures the total wall-clock time required to complete the deduplication task. We also used the library called *time* here too.

$$T = T_{\text{end}} - T_{\text{start}} \tag{4.2}$$

Where:

T_{start} = time when algorithm begins space.

T_{end} = time when algorithm finishes.

- **Memory Usage:** Tracks RAM consumption during runtime, including peak usage. We are using $memory\_info()$ class method from a python library called *psutil*.

$$M = M_{\text{peak}} - M_{\text{base}} \tag{4.3}$$

Where:

M_{peak} = peak memory during deduplication.

M_{base} = memory used before running deduplication.

- **I/O Throughput:** Assesses the volume and speed of disk read/write operations using Linux I/O counters.

$$\text{Throughput} = \frac{N}{T} \tag{4.4}$$

Where:

N = number of records processed.

T = execution time.

- **Deduplication Ratio:** Reflects the effectiveness of duplicate detection, calculated as Deduplication Ratio.

$$DR = \frac{\text{Original Data Size}}{\text{Deduplicated Data Size}} \times 100\% \tag{4.5}$$

- **Collision:** Points out how many times the deduplication algorithms hash functions generates same hash value for different blocks for data with the help of key-value pair dictionary data structure.

Each container is configured to run a dedicated hash-based deduplication workload on an identical CSV file, ensuring test isolation. Resource usage is monitored during each test iteration, and a statistical average is computed for comparison.

Algorithm 1 Performance Benchmarking for Deduplication

```
1: procedure BenchmarkDeduplication(file_path, hash_func, output_file, hash_name)
2:     Start CPU and wall timers
3:     Initialize duplicates, unique_lines, hash_map
4:     total_lines ← 0, collision ← 0
5:     for all lines in file (skipping header) do
6:         total_lines ← total_lines + 1
7:         h ← hash_func(line)
8:         if h in hash_map then
9:             Append line to duplicates
10:            if hash_map[h] ≠ line then
11:                collision ← collision + 1
12:            end if
13:        else
14:            hash_map[h] ← line
15:            Append line to unique_lines
16:        end if
17:    end for
18:    Write unique_lines to output file
19:    Compute statistics (memory, CPU, deduplication ratio, throughput)
20:    Print and save report
21: end procedure
```

4.3 Model Specification with Algorithm and Container Framework

The core experimental model is based on deploying three parallel container workflows (LXC, Docker, and Podman), each running implementations of SHA-256, MD5, and CRC32-based deduplication logic written in Python. The data stream is segmented into fixed-size chunks, each of which is hashed and indexed. Duplicate entries are flagged and excluded from final storage. The approach reflects typical deduplication use cases in data-intensive cloud workloads.

4.3.1 Experimental Procedures

- Load the input CSV file .
- Partition the file into fixed-size or individual blocks (e.g., 4 KB).
- Compute the hash of each block using the selected algorithm.
- Store hashes in a dictionary to detect duplicates.

- Record performance metrics (time, CPU, memory, I/O).
- Repeat the test 10 times for each configuration.
- Aggregate and analyze results.

4.3.2 Algorithmic Structure of Hash-based Deduplication Methods

In case of general algorithmic design of hash-based deduplication, each data block goes through each of the algorithms that we mentioned (SHA-256, MD5 and CRC32), which each create a unique sequence of fixed size string for each data blocks which are called "Fingerprints" and these fingerprints then get stored in a "Fingerprint Table". Each of the algorithms are different when it comes to calculating and generating fingerprints and the length of the fingerprints itself (like for SHA-256, the generated fingerprint for each data block is always 256 bits; for MD5, that is 128 bits and finally for CRC32, 32 bits), however the fundamental structure for all of them are somewhat same with slight difference where SHA256 and MD5 both are cryptographic hash functions with high (SHA256) to moderate (MD5) data integrity and security but low speed, where as CRC32 is non-cryptographic checksum based using polynomial division which is almost same as hashing that creates fingerprint with drawbacks likes higher collision chance and preliminary deduplication algorithm tags but with fastest of speed among all three. Each time a new fingerprint is created, it is matched with the other fingerprints that is saved in a table. If the fingerprint is already in the table, then the data is seen as a duplicate or redundant data; but if no match is found in the table, then it's a unique or non-duplicate data. To get a clear visualization, please review the Figure 4.3 captioned as "Algorithmic Flowchart of a hash based Deduplication Method".

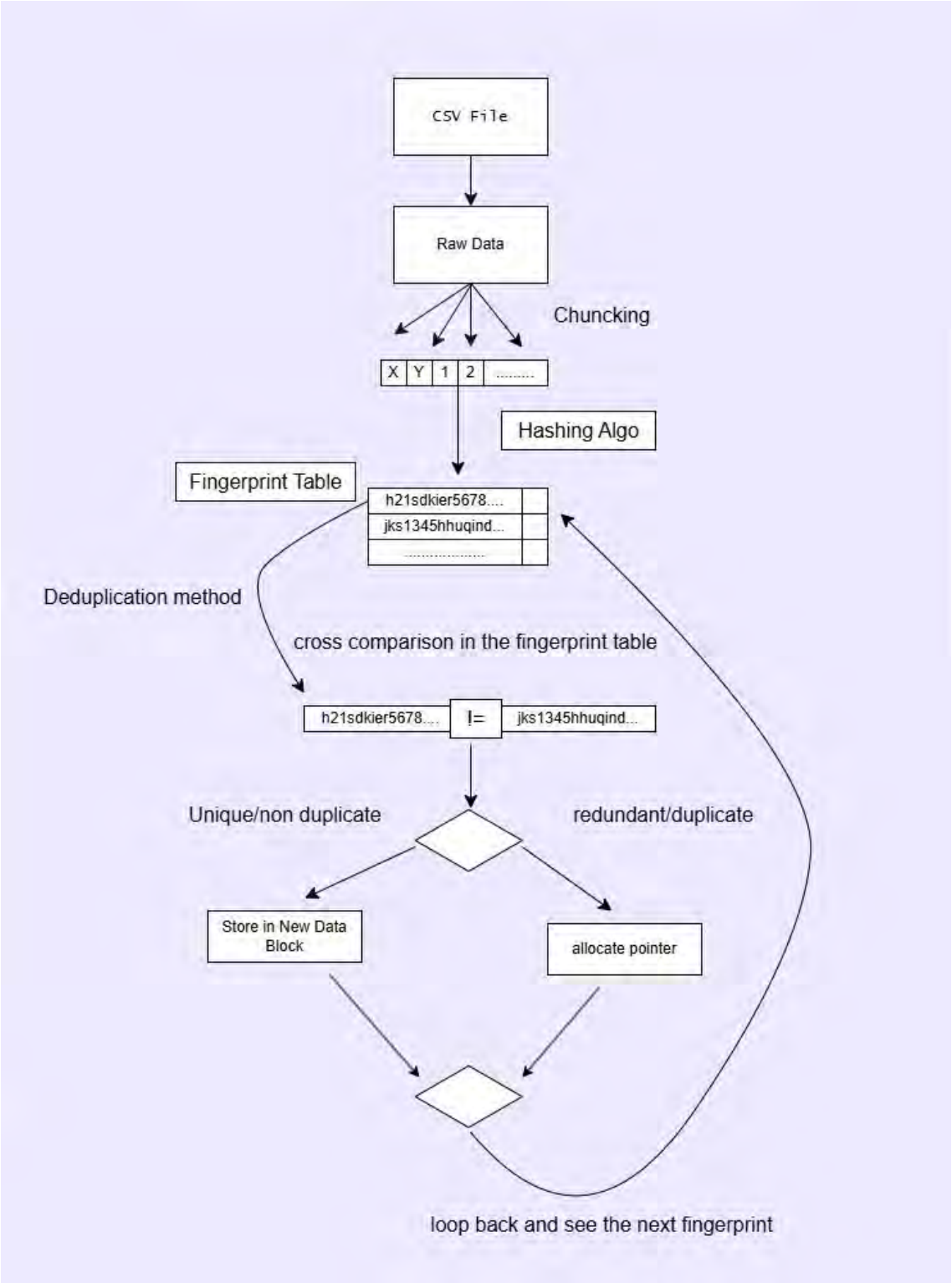


Figure 4.3: Algorithmic flowchart of a hash based deduplication method

4.3.3 Container Architecture

Figure 4.4 illustrates the internal architecture of three containerization technologies which are LXC, Docker, and Podman that has been employed in this study. Each container engine operates within a Linux-based Guest OS (virtualized) environment and follows a distinct design approach:

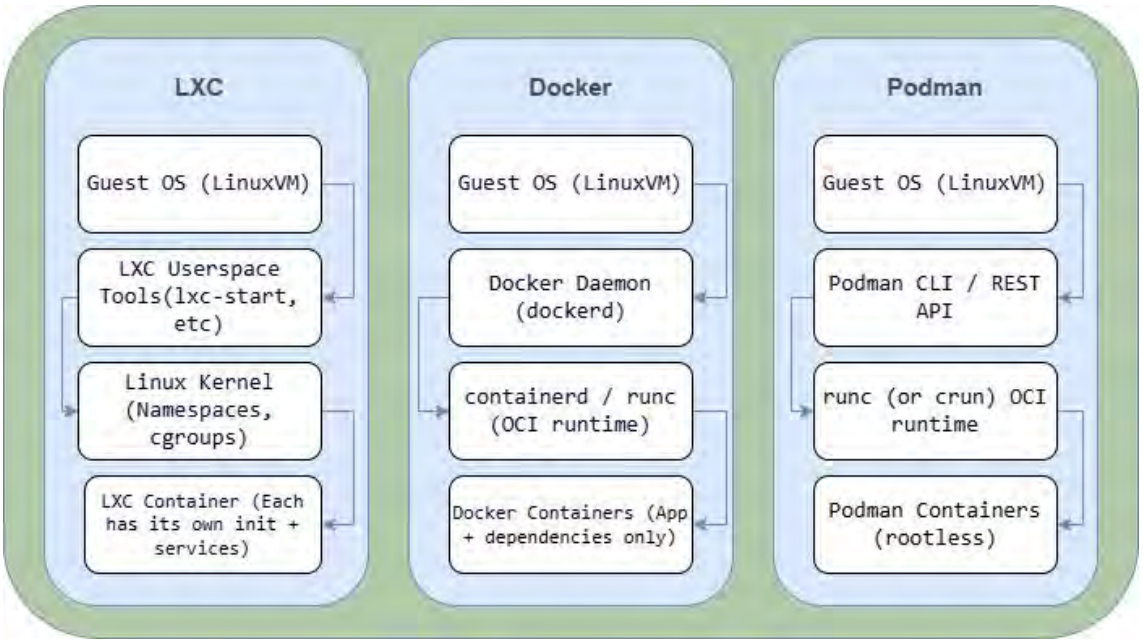


Figure 4.4: Comparative architecture of LXC, Docker, and Podman container engines used in this study and how containers interact with the guest OS and the Linux kernel.

- LXC (Linux Containers): LXC is a system container model that closely resembles a lightweight virtual machine. Each LXC container includes its own init system and services. It uses kernel features like namespaces and cgroups for isolation and resource control, and relies on LXC userspace tools (e.g., lxc-start) to manage container lifecycle.
- Docker: Docker follows a daemon-based client-server architecture. It uses the dockerd daemon to manage containers, while containerd and runc serve as low-level OCI-compliant runtimes [13]. Docker containers typically package only the application and its dependencies, excluding init systems, making them lightweight and suited for microservices.
- Podman: Podman is a daemonless and rootless container engine. It uses Podman CLI or REST API for interaction, and either runc or crun as the OCI runtime. Since it operates without a central daemon, each Podman container is managed independently under the user's control, improving security and modularity.

Each approach varies in its philosophy of containerization: LXC for full-system containers, Docker for application containers managed via a daemon, and Podman for lightweight, rootless containers with direct user-space management.

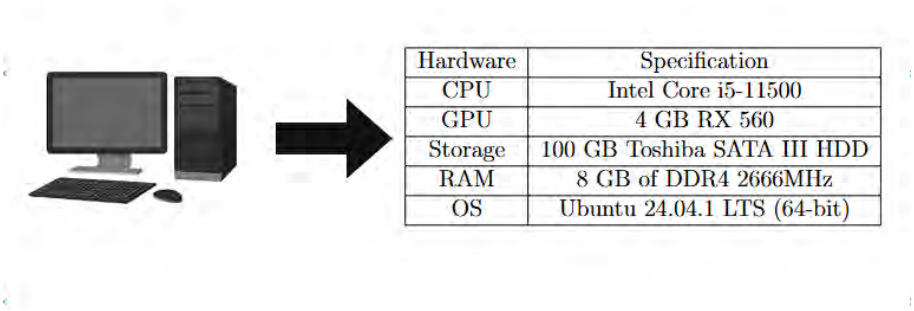
Chapter 5

Experiment and Result Analysis

This section presents the experimental environment setup and the results and analysis of our finding of different deduplication algorithms in different containerized runtime environments. This section also depicts the balance and tradeoff between multiple performance metrics like throughput, memory, collision etc.

5.1 Experimental Setup

For our experiment, all containers that we are testing are executed inside virtual machines running on Oracle VirtualBox (A Type 2 Hypervisor Software). The VM is configured with Ubuntu 24.04.1 LTS (64-bit), 8 GB of DDR4 2666MHz RAM, a 6-core Intel Core i5-11500 CPU (2.7 GHz), a 4 GB Rx560 GPU (1175 MHz) and a 100 GB Toshiba SATA III HDD. This setup replicates a mid-range computing environment, balancing between real-world constraints and performance measurement fidelity. Figure 5.1, showcases a simplified hardware and software specification of the experiment setup.



Hardware	Specification
CPU	Intel Core i5-11500
GPU	4 GB RX 560
Storage	100 GB Toshiba SATA III HDD
RAM	8 GB of DDR4 2666MHz
OS	Ubuntu 24.04.1 LTS (64-bit)

Figure 5.1: Used experimental setup specification

5.2 Workload Characteristics

For the experimental assessment three artificial datasets of varying size were produced to mimic real-world data from entries. The datasets contain 10 million (1 GB), 5 million (514 MB) and 1 million (102 MB) entries respectively. To emulate real world data, the datasets were generated using the Faker library in Python, which is commonly used for the creation of synthetic but realistic data like names, addresses, contact information, etc.

For each of the three datasets records were intentionally distributed to ensure that 70% of the entries are unique and the remaining 30% are redundant. This ratio was chosen to reflect practical scenarios where redundancy is prevalent in large-scale data storage and retrieval systems. The inclusion of redundancy is particularly important for testing the effectiveness of deduplication algorithms, as it allows for a meaningful evaluation of their performance in reducing storage overhead and improving efficiency.

5.3 Results and Analysis

5.3.1 Outcome Overview

Container	Algorithm	Memory(MB)	CPU Time(s)	Throughput(kLines/s)	Collision	Response Time(s)
Docker	CRC32	247.73	0.8515	1119.53	82	0.9359
Docker	MD5	272.99	1.0915	887.11	0	1.1718
Docker	SHA256	294.79	1.0712	905.23	0	1.1506
LXC	CRC32	249.24	0.9285	1029.09	82	1.0220
LXC	MD5	263.42	1.1718	823.99	0	1.2657
LXC	SHA256	285.32	1.1733	822.05	0	1.2714
Podman	CRC32	247.73	0.8920	1076.91	82	0.9762
Podman	MD5	273.91	1.1852	815.87	0	1.2694
Podman	SHA256	295.64	1.1402	837.09	0	1.2415

Table 5.1: Table of average performance metrics for the workload of 1 million records

Container	Algorithm	Memory(MB)	CPU Time(s)	Throughput(kLines/s)	Collision	Response Time(s)
Docker	CRC32	1138.39	4.5859	1054.08	2033	4.9521
Docker	MD5	1249.16	5.7129	852.00	0	6.0830
Docker	SHA256	1358.41	5.8212	836.58	0	6.2109
LXC	CRC32	1139.49	4.9605	971.82	2033	5.3735
LXC	MD5	1230.37	6.1008	795.39	0	6.5179
LXC	SHA256	1339.47	6.1742	785.08	0	6.6111
Podman	CRC32	1144.30	5.4122	901.09	2033	5.7898
Podman	MD5	1253.12	6.4139	761.23	0	6.7875
Podman	SHA256	1362.08	6.1890	788.14	0	6.5680

Table 5.2: Table of average performance metrics for the workload of 5 million records

Container	Algorithm	Memory(MB)	CPU Time(s)	Throughput(kLines/s)	Collision	Response Time(s)
Docker	CRC32	2249.03	10.0910	971.20	8148	10.7230
Docker	MD5	2488.29	12.2657	797.88	0	12.9757
Docker	SHA256	2705.99	13.0272	753.66	0	13.7815
LXC	CRC32	2272.89	10.9035	897.33	8148	11.6243
LXC	MD5	2369.12	12.9641	730.72	0	14.1405
LXC	SHA256	2587.27	13.1257	744.59	0	13.9144
Podman	CRC32	2252.67	11.3532	855.67	8148	12.1802
Podman	MD5	2470.57	14.6494	672.88	0	15.3456
Podman	SHA256	2686.57	14.6287	672.75	0	15.3958

Table 5.3: Table of average performance metrics for the workload of 10 million records

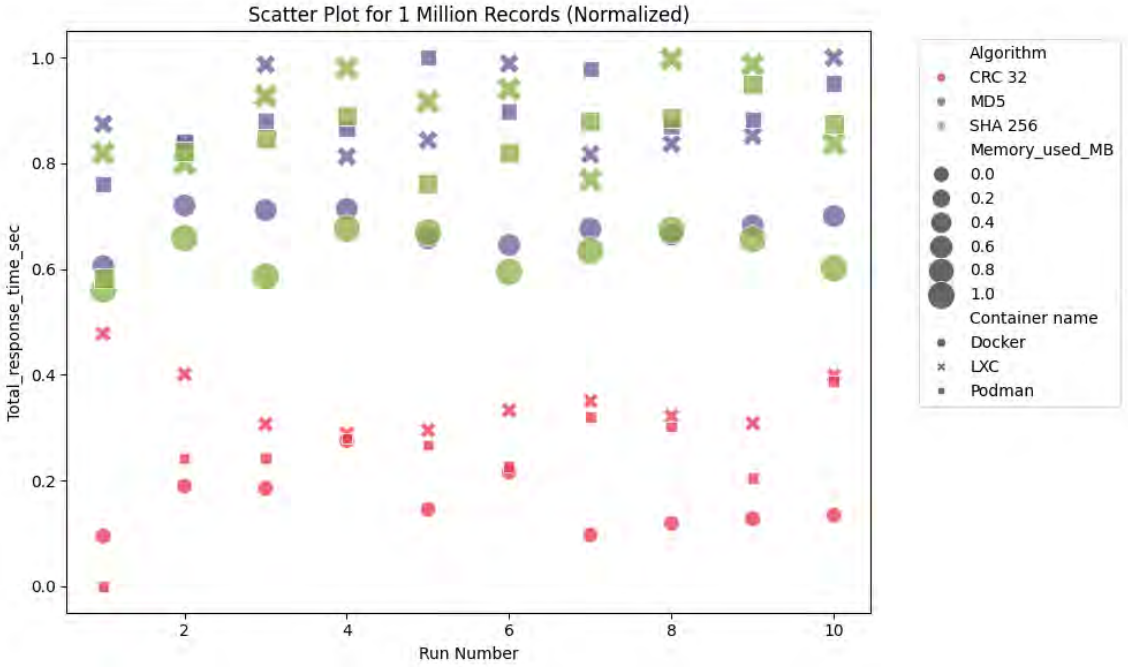
The following three tables represent the accumulated and averaged results we got from our experiment. As previously mentioned, we have taken three workloads of three different size that being one million, five million and ten million; where each of the workload had 70% unique data and 30% redundant data. We ran each workload in each container for an epoch of 10 times to see how the results came out. We have ran the whole experiment for a total of 2 times to obtain more data to work on. Thus the total number of testing considering, there are three hash-based deduplication algorithms on each of the three different containerized environment with an epoch of ten times while ruuning it for two times came out to be a total of - $(3 \times 3 \times 10 \times 2) = 180$ times. And as we used three different workloads or datasets for testing, then the total number of testing comes out to be - $(3 \times 180) = 540$ times. Now, each of the run gave us multiple types of parameters of performance metrics like the execution time, memory usage, throughput etc. So, considering only the most important of these performance metrics, the total number of data parameters that came out for each of the datasets or workload is - $(7 \times 180) = 1260$ data parameters and with three datasets or workloads, the grand total of data parameters came out as - $(3 \times 1260) = 3780$ data parameters, in which we have considering all of the 540 test cases. Showcasing these many results are very complicated, as a result we have reduced the data by taking the mean or average for each performance metrics category. This brought about, for each of the datasets or workloads, we have around 63 data parameters after the compression and as there are three of them, the total number of data parameters came out after compression is - $(3 \times 63) = 189$ data parameters. These data are shown in the above three tables (Table 5.1, Table 5.2 & Table 5.3), each consisting of those compressed 63 data parameters respectively.

5.3.2 Performance Benchmarks and Impact

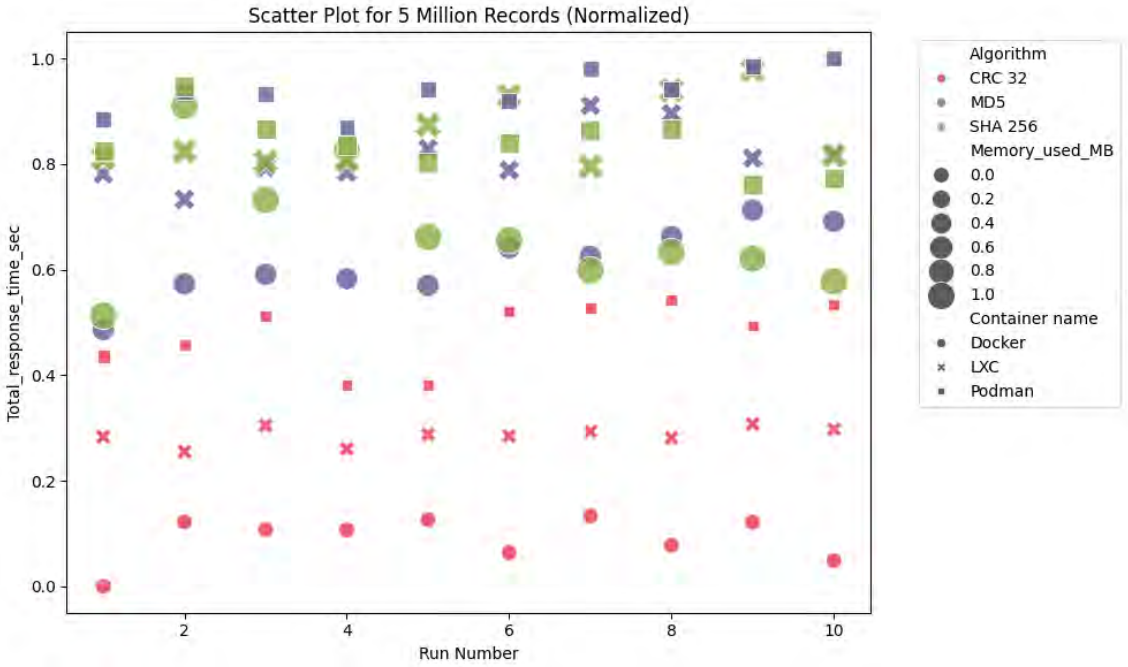
The experimental evaluation was performed on three hashing algorithms (CRC32, MD5 and SHA256) and in three container environments (Docker, LXC and Podman). The findings are summarized in the following way:

Execution Time (based on response time): Execution time is proportional to the data size. Among the algorithms, CRC32 was always the algorithm with the lowest execution time followed by MD5, whereas SHA-256 was the slowest with its computational complexity. However, on the whole, Docker had the lowest execution times, followed by Podman, with LXC sometimes showing slower execution as a result of isolation overhead. The Scatter Plot (Figure 5.2 “Scatter Plot of Execution

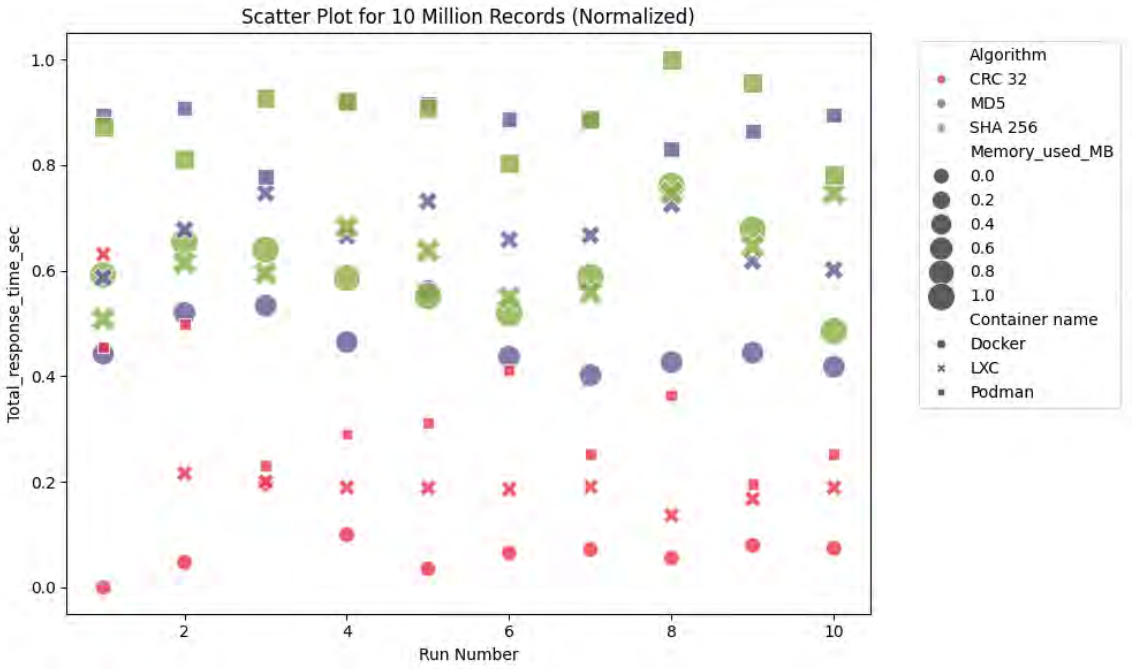
Time (Normalized)") will help in visualizing these distribution.



(a) Scatter plot for 1 million records



(b) Scatter plot for 5 million records



(c) Scatter plot for 10 million records

Figure 5.2: Scatter plot of execution time (normalized)

Memory Usage: Average memory usage ($mean \pm 95\% CI$) was $\sim 1.22 \pm 0.17$ GB for CRC-32, $\sim 1.32 \pm 0.18$ GB for MD5, and $\sim 1.44 \pm 0.20$ GB for SHA-256. CPU time followed the inverse of throughput: CRC-32 ($5.55 \pm 0.85s$) < MD5 ($6.84 \pm 1.04s$) < SHA-256 ($6.93 \pm 1.07s$). Total response times tracked similarly. Bar charts for memory are provided in Figure 5.3"Average memory usage by each algorithm".

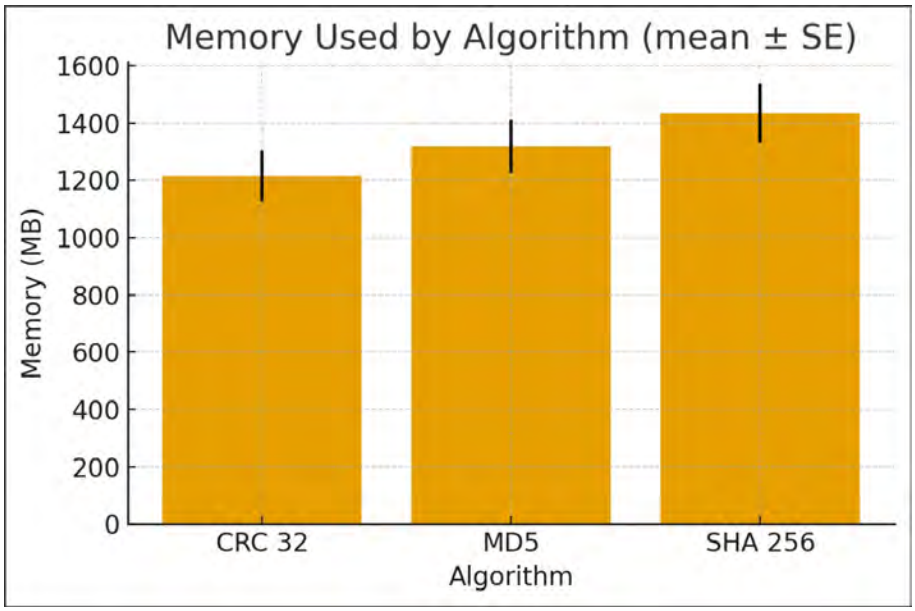


Figure 5.3: Average memory usage by each algorithm

I/O Throughput: Across all datasets and containers, CRC-32 achieved the highest throughput ($mean \approx 986,303 lines/s, 95\% CI \pm 18,978$), followed by MD5 ($\approx 793,006 \pm 13,070$) and SHA-256 ($\approx 793,908 \pm 13,924$). Welch's t-tests show CRC-32 is significantly faster than both MD5 and SHA-256 ($p < 1e^{-15}$), while MD5 and SHA-256 do not differ meaningfully in throughput ($p \approx 0.93$). Boxplots (Figure 5.4 "Throughput comparison by algorithm and container") visualize these distributions.

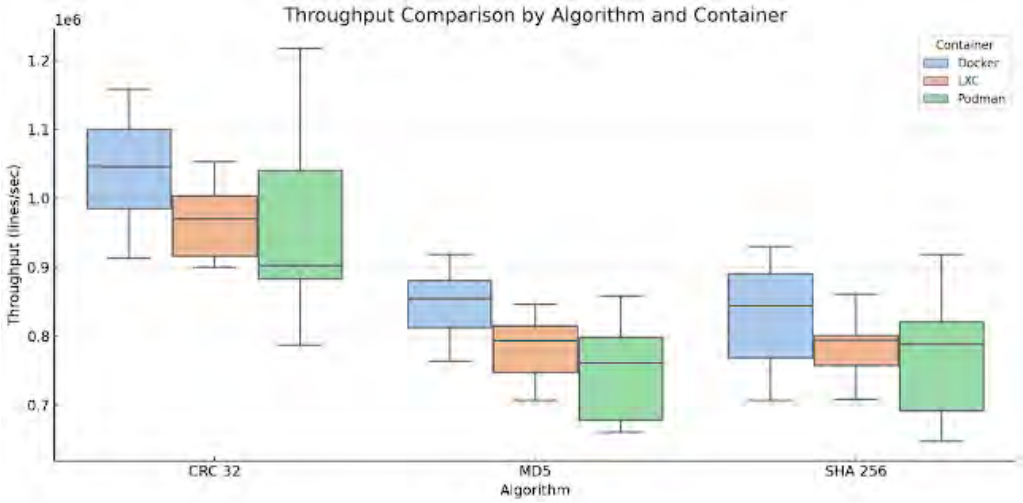
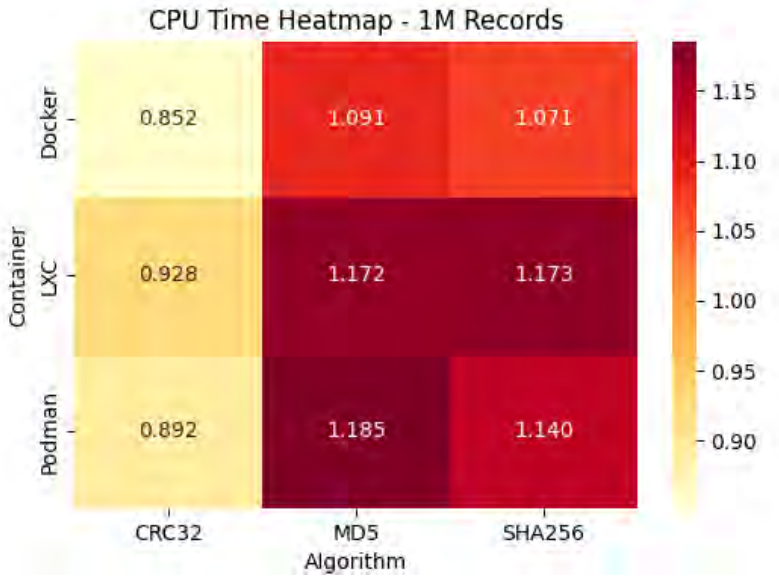
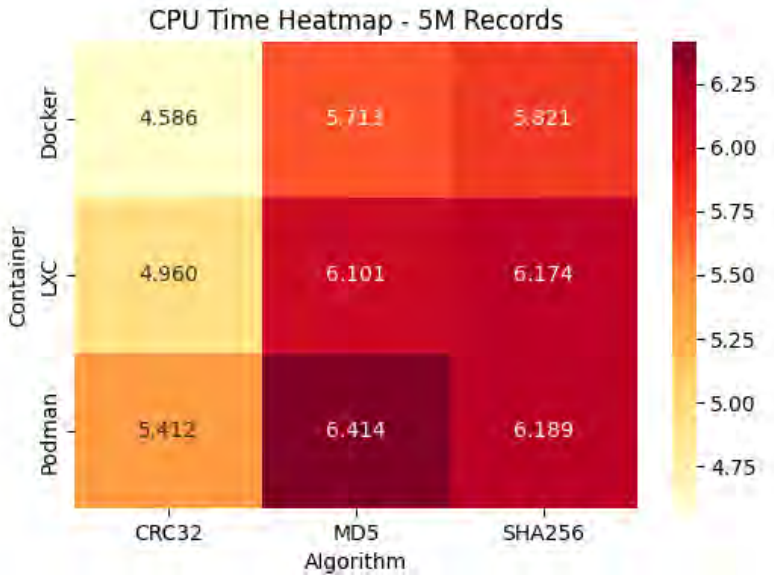


Figure 5.4: Throughput comparison by algorithm and container

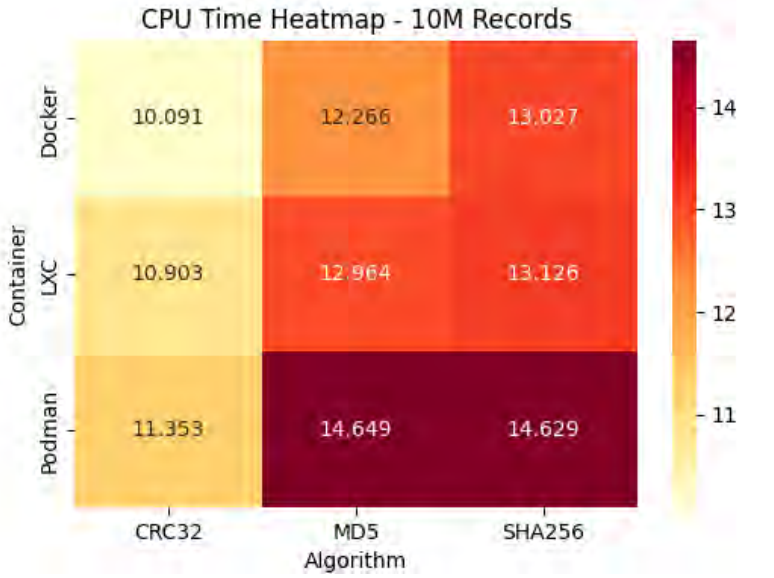
CPU Utilization (based on cpu time): CPU utilization increased as dataset size increased. CRC32 was the fastest, then MD5, and SHA256 used the most CPU resources. Docker and Podman generally exhibited better CPU utilization while LXC sometimes exhibited volatility. We can see the average Cpu Time taken in each of the containers for each algorithms from the following heatmap in Figure 5.5 (Heatmaps for cpu time (in seconds)) to judge the cpu utilization more statistically.



(a) Heatmap of cpu time for 1 million records



(b) Heatmap of cpu time for 5 million records



(c) Heatmap of cpu time for 10 million records

Figure 5.5: Heatmaps for cpu time (in seconds)

Accuracy (Collisions): Only CRC-32 exhibited non-zero collisions: mean collision rate $\approx 4.34 \times 10^{-4}$. MD5 and SHA-256 had zero collisions across all runs. This highlights the classic trade-off: CRC-32 offers markedly higher speed with a small but non-zero collision risk, whereas MD5/SHA-256 provide zero observed collisions at lower throughput (see Figure 5.6 “Collision rate by algorithm”).

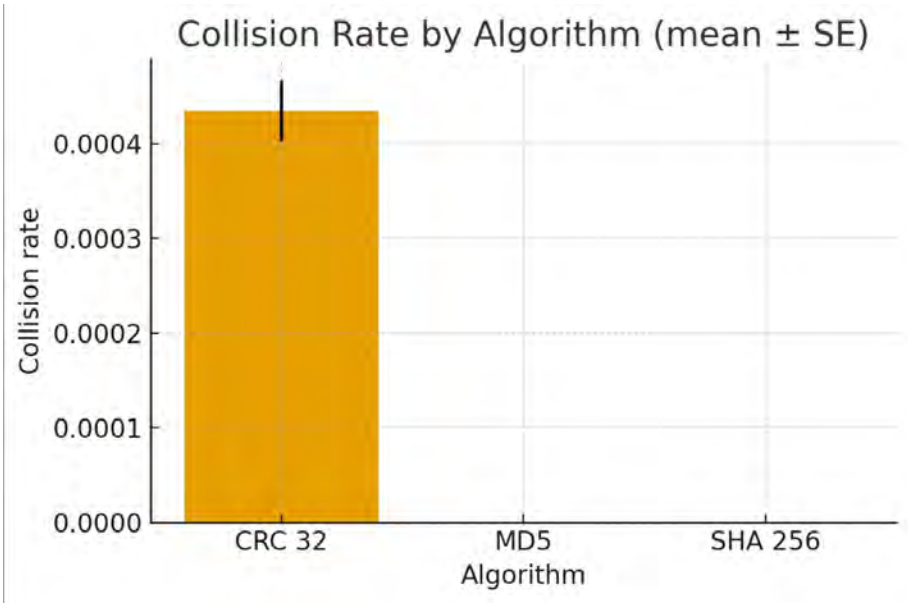


Figure 5.6: Collision rate by algorithm

Scalability: Analyzing the throughput degradation, resource scaling (memory and CPU time increases), efficiency ratios (throughput per MB and per CPU second), stability (standard deviation across runs), and collision rate growth can help to establish the capacity of scalability of a container as datasets expand from 1M to 10M records.

For Docker with CRC32, throughput degrades by 5.8% from 1.12M lines/sec (1M) to 1.05M (5M), then 7.6% to 0.97M (10M), with low variance (std 25k at 10M). Memory scales by 360% (247MB to 1138MB) then 98% (to 2249MB), CPU time by 438% then 120%, efficiency drops (throughput per MB: 4530 to 431; per CPU: 1.31M to 96k). Collision rate rises from 8e-5 to 8e-4, indicating moderate accuracy loss at scale.

LXC shows similar throughput drops (5.6% to 0.97M at 5M, 7.2% to 0.90M at 10M) with even lower variance (std 8k at 10M), memory scaling (249MB to 1139MB to 2273MB), and better CPU efficiency retention (throughput per CPU: 1.11M to 82k). For MD5, throughput falls 3.6% then 8.0% (824k to 795k to 731k), memory efficiently (263MB to 1230MB to 2369MB), zero collisions throughout.

Podman exhibits steeper degradation for CRC32 (16.4% to 0.90M at 5M, 4.9% to 0.86M at 10M) with higher variance (std 34k at 10M), similar memory but worse CPU scaling, and reduced efficiency (per MB: 4340 to 381; per CPU: 1.21M to 75k). Hashes like SHA256 show pronounced drops (837k to 788k to 673k), highlighting runtime overheads.

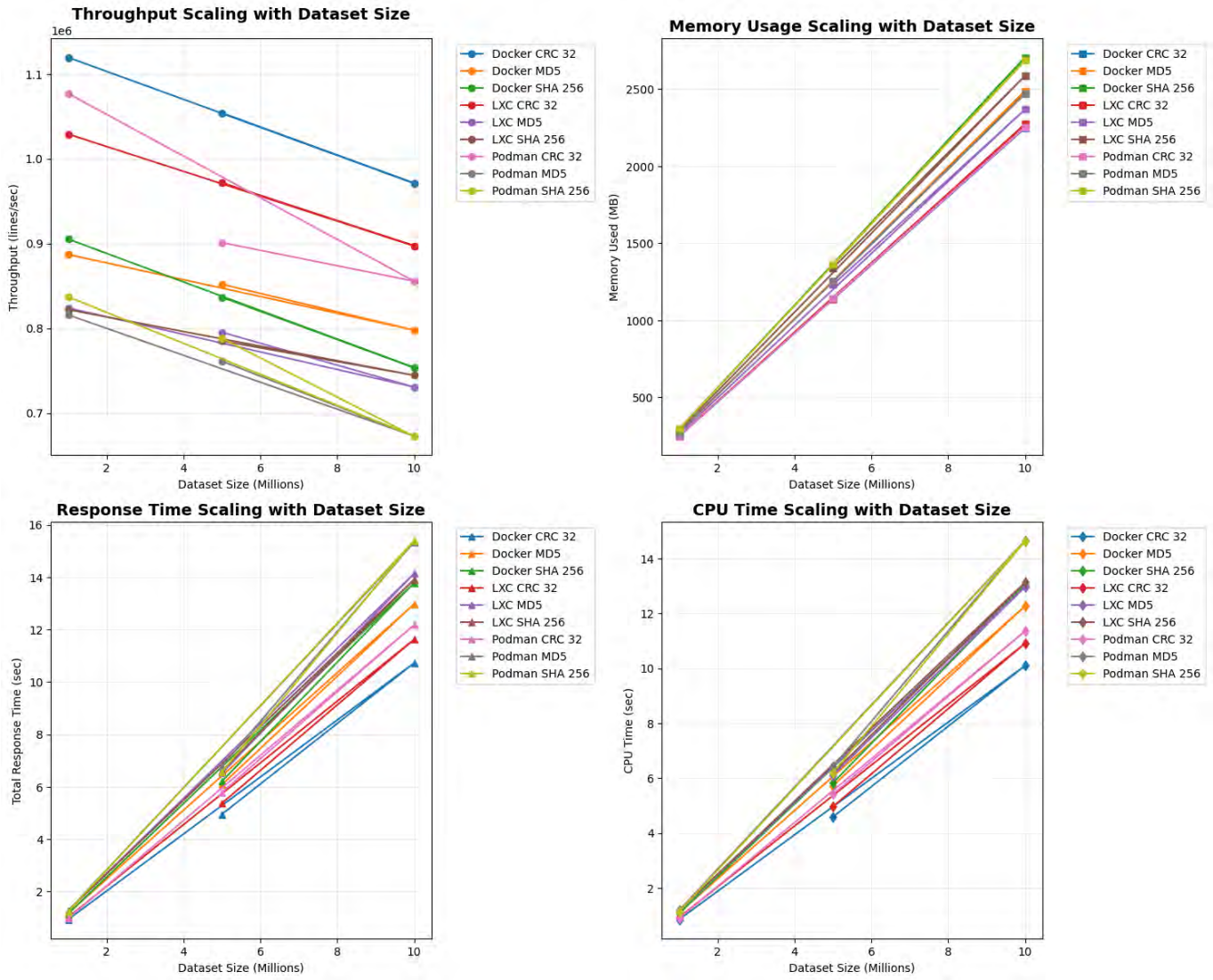


Figure 5.7: Scalability

5.4 Practical Takeaways

From the data, we achieved from our research there has been somethings that have been cleared out which are -

- If zero observed collisions are mandatory, MD5 or SHA-256 are appropriate; they perform similarly in speed and resources.
- If maximum throughput is critical and a ∼434 ppm collision rate is acceptable within downstream verification/repair workflows, CRC-32 provides ∼24% higher throughput on average than MD5/SHA-256.
- Docker showed the best average throughput across containers in this experiment; however, differences are modest relative to algorithm choice.

So, to sum it all up, algorithm and container choice depends on system requirements: CRC32 in Docker for speed-sensitive and less security sensitive applications, MD5 in Docker or LXC for performance and accuracy balance, SHA256 in Docker for correctness and collision avoidance but more resource consuming.

Chapter 6

Discussion

6.1 Key Findings and Significance

This study provides a comprehensive evaluation of three container technologies: Docker, LXC, and Podman; across varying data scales (1M, 5M, and 10M records) and deduplication algorithms (CRC32, MD5, SHA256). The experiments reveal distinct performance profiles, offering insights into scalability, stability, and suitability for time-critical or security-sensitive scenarios. By analyzing throughput, response time, memory usage, and collision rates, we elucidate trade-offs that inform container selection in diverse deployment contexts.

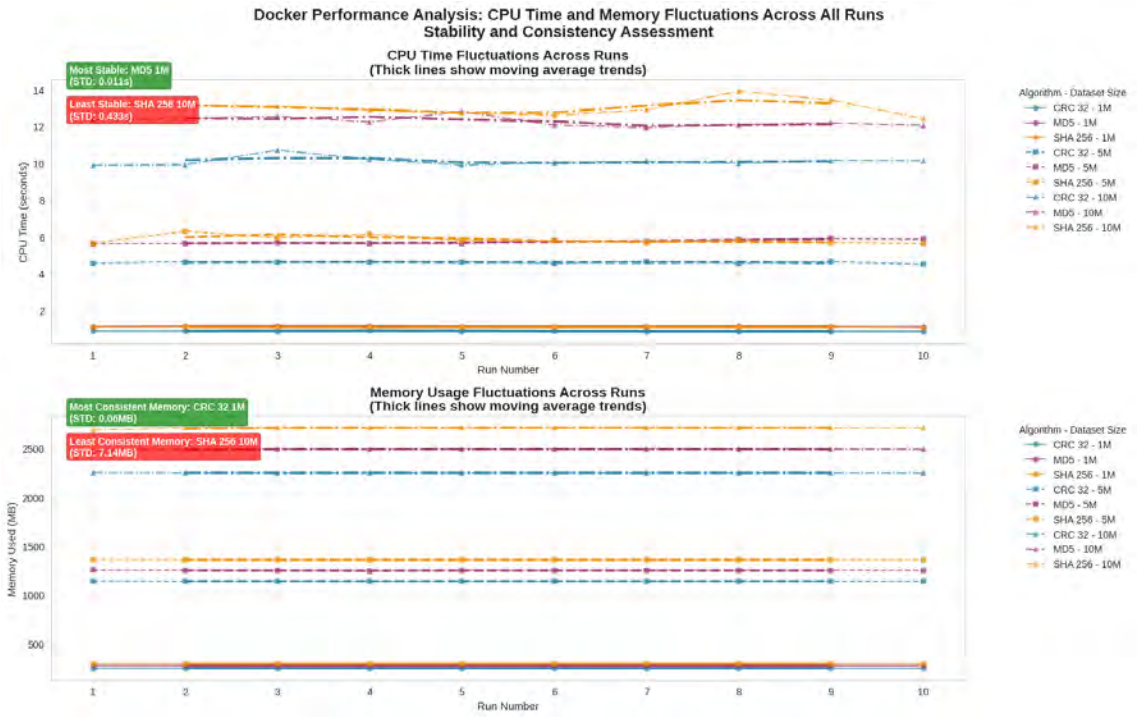
6.1.1 Stability Analysis

Measured by the consistency of throughput and response times across multiple runs, varies by container and algorithm. Docker exhibits the least variance in throughput for CRC32 (e.g., standard deviation 0.03M lines/sec for 5M records), indicating robust performance under repeated workloads. LXC shows comparable stability for hash-based algorithms, with tighter memory usage distributions (e.g., 1230MB $\pm$ 0.2MB for MD5 on 5M records). Podman, however, displays higher variability in response times at larger scales (e.g., 14.8–16.0s for SHA256 on 10M), suggesting potential instability under heavy loads. This could be attributed to Podman's daemonless design, which, while enhancing security, may introduce scheduling inefficiencies not captured in our controlled test environment.

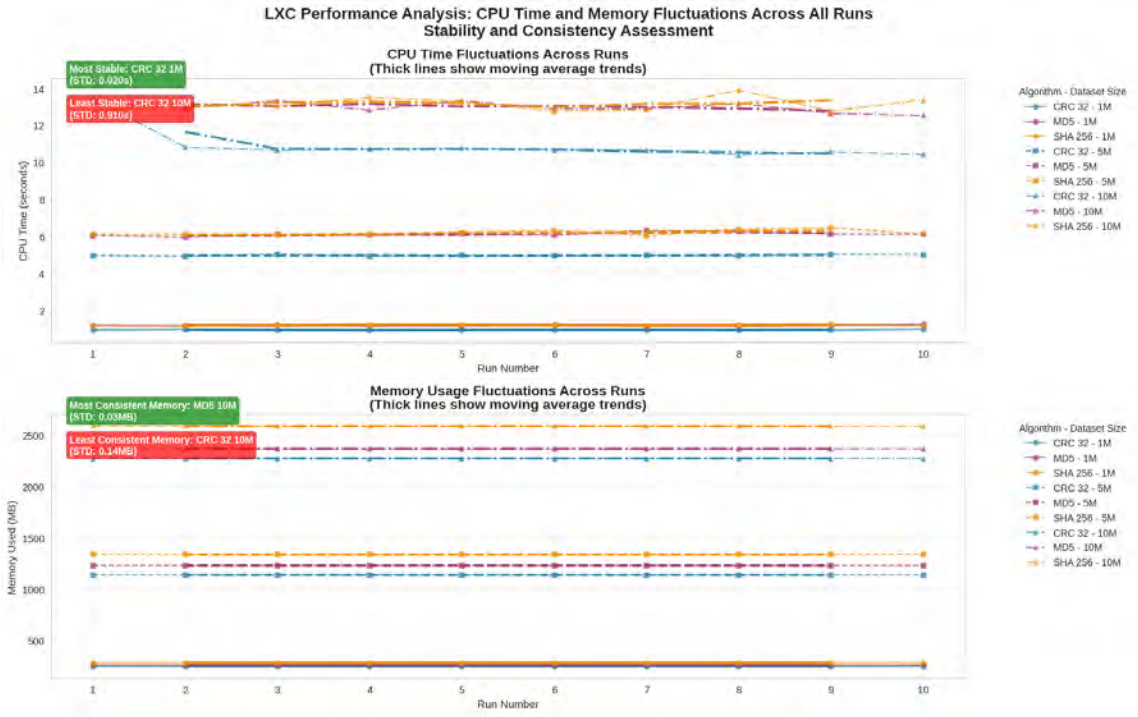
6.1.2 Implications for Deployment Scenarios

The findings have significant implications for diverse use cases. For cloud-based microservice deployments, where high throughput and low latency are paramount, Docker paired with CRC32 is optimal for non-critical deduplication tasks (e.g., log aggregation), achieving up to 1.12M lines/sec for small datasets. However, for microservices requiring guaranteed deduplication (e.g., transactional databases), Docker with MD5 offers a balance of speed (0.89M lines/sec for 1M workload) and reliability (zero collisions). In resource-constrained IoT environments, LXC's lower memory footprint for hash-based algorithms (e.g., 263 MB for MD5 on 1M workload) makes it preferable, especially for edge devices processing moderate datasets. For security-sensitive workloads, such as healthcare data processing, SHA256 is rec-

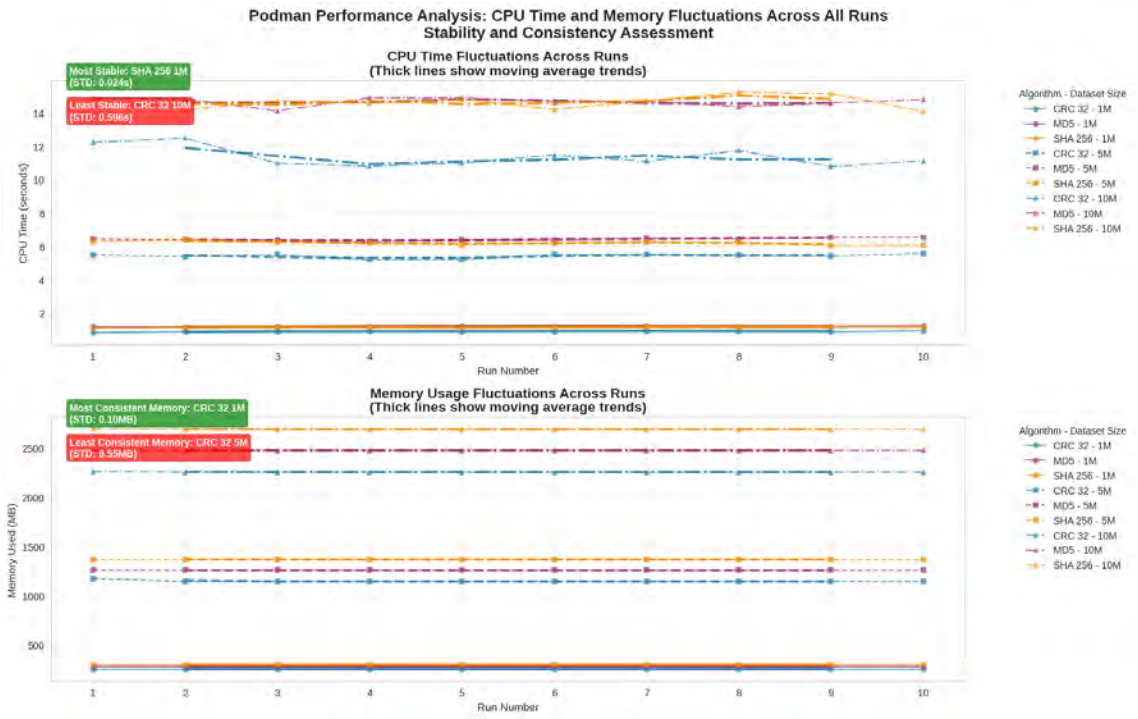
ommended due to its theoretical robustness against collisions, despite higher computational overhead (13.8s for Docker vs. 12.9s for MD5 both for 10M workload). LXC with SHA256 provides a memory-efficient alternative (2587MB vs. Docker's 2706MB), critical for secure deployments on constrained servers. Podman's rootless operation may theoretically enhance security, but its performance lag suggests it is less viable unless rootless execution is a strict requirement.



(a) Docker performance analysis



(b) Lxc performance analysis



(c) Podman performance analysis

Figure 6.1: Cpu time (sec) and memory usage (MB) across containers and algorithms for varying data scales

6.1.3 Decision Framework

The observed trade-offs between speed vs. accuracy, memory vs. throughput, and stability vs. scalability — underscore the need for a structured approach to container and algorithm selection. To address this, we developed a decision tree that encapsulates these factors, guiding users based on accuracy requirements, data scale, speed priority, and memory constraints. The tree prioritizes zero-collision needs for critical applications while recommending high-throughput options for real-time processing, ensuring practical applicability across the discussed scenarios.

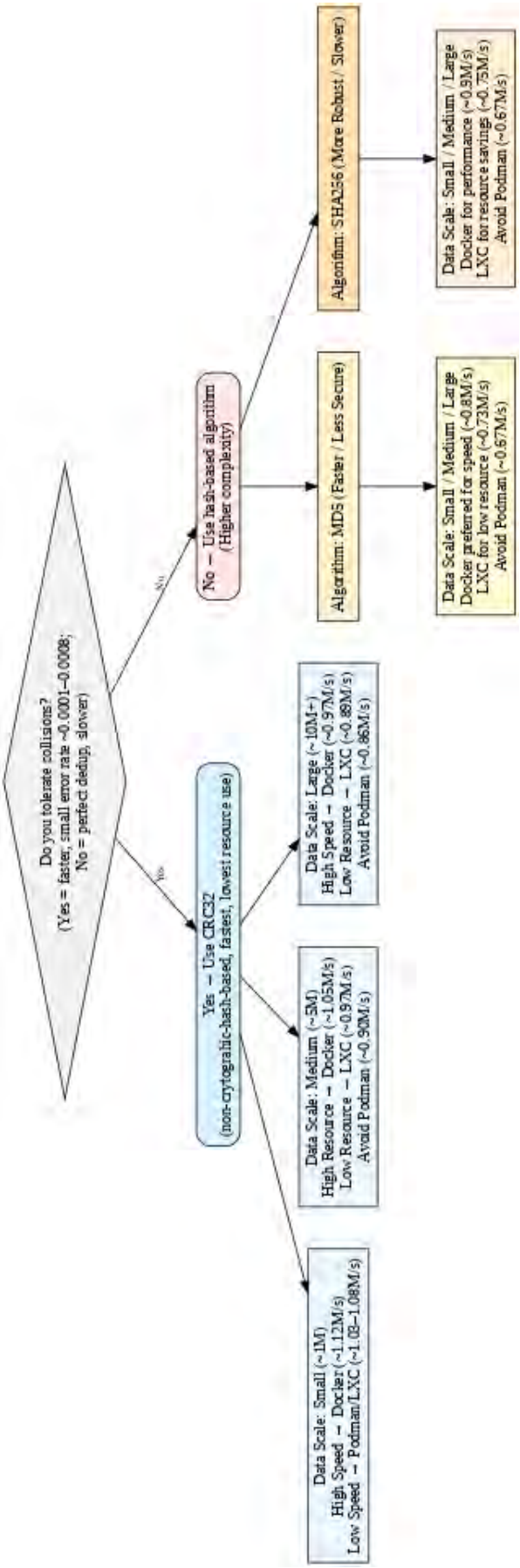


Figure 6.2: Decision tree visualization for choosing containers based on algorithm type and data scale

6.2 Limitations and Future Work

In this research work, we have worked with deduplication algorithms which are cryptographic hash function based like SHA256 (with a hash key of 256 bits) & MD5 (with a hash key of 128 bits) which are very secure with very very low chance of collision or preimages but compromises when it comes to speed; and also non-cryptographic checksum based function like CRC32 (with a bit length of 32 bits) which is typically much faster compared to cryptographic but compromises when it comes to security as it is prune to collision, so it is ineffective in large scale datasets and unreliable [44], [46]. The best is to use CRC32 as pre-filter to reduce the number of data comparisons and then use cryptographic for accuracy. Moreover, all of the algorithms that we used will effectively work on data or workloads that are repeated for multiple times or are exact similar. However, when it comes to nearly similar or partially similar data, these algorithm under performs or does not catch those cases as the value calculated by the used algorithms function changes as the data or workload changes. For these types of near similar data, there are different types of other deduplication algorithms out there that performs much better like the generalized deduplication [27], Simhash algorithm [51], Secure Locality Sensitive Hashing (SLSH) [32], FuzzyDedup [39]. It can be a novel approach for future research work, to use these near similar data deduplication methods and compare among them on performance basis or to use them for other areas. Another thing to point out is that, we are considering only text-based data for our research, however there are more types of data that are out there like images, audio etc. which needs to be addressed too.

Related work	Category	VMM(hypervisor type)	Container	Resource Considered			Service
				CPU	Mem	Disk	
Zhang et al. [21]	VM versus container	KVM (type 1)	Docker	✓	✓	–	Hadoop
Verma et al. [48]	Container on VM	Amazon EC2 (type 1)	Docker	✓	–	–	HTTP
Felter et al. [10]	VM versus container	KVM (type 1)	Docker	✓	✓	–	Redis, MySQL
Aqasizade et al. [49]	Container on VM	KVM, Xen (type 1)	Docker, LXC, Podman	✓	✓	✓	MySQL
Sharma et al. [15]	VM versus container	KVM (type 1)	LXC	✓	✓	✓	MySQL
Ramalho et al. [14]	VM versus container	KVM (type 1)	Docker	✓	–	–	–
Chae et al. [23]	VM versus container	KVM (type 1)	Docker	✓	✓	✓	–
Mavridis et al. [17]	Container on VM	KVM (type 1)	Docker	✓	✓	✓	–
Mondesir et al. [26]	Container on VM	VMware workstation (type 2)	Docker	✓	✓	✓	Arma3
Lingayat et al. [20]	Container on VM	KVM (type 1)	Docker	✓	–	✓	–
Silva et al. [47]	Container vs Container	WSL2, QEMU (type 1)	Docker, LXD	✓	–	✓	GeekBench6, CrystalDiskMark
Our work	Container vs Container	KVM, Podman-machine (type 1)	Docker, LXC, Podman	✓	✓	✓	SHA256, CRC32, MD5

Table 6.1: Our work compared to other related work

Chapter 7

Conclusion

In conclusion, the research can be derived after analysing the results and the execution is that when we need maximum speed, among the combinations the deduplication via CRC32 inside Docker consistently has the highest throughput which ensures the priority of performance over strict data integrity. However, the applicability of it is limited in use cases requiring strong reliability due to its elevated hash collisions. If we are in a need of resource-constrained runtime; among the three containers LXC demonstrated the most efficient CPU utilization, making it a suitable choice in scenarios where computational resources are limited with its performance improving over subsequent runs. However, Podman exhibited higher performance fluctuations, especially with larger workloads and more complex instructions. This variability indicates that additional evaluation may be required before it is deployed in production grade deduplication systems. MD5 containerized within Docker demonstrates a great balance between performance and data integrity. This configuration achieves robust throughput and reduces the chance of collisions compared to CRC32, which makes it suitable for general-purpose deduplication workloads. This study advances the knowledge on choosing container environments best suited for different workloads as well as deduplication algorithms.

Bibliography

[1] P.-H. Kamp and R. N. Watson, "Jails: Confining the omnipotent root," in *Proceedings of the 2nd International SANE Conference*, vol. 43, 2000, p. 116.

[2] P. Barham et al., "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[3] H. Jinpeng, L. Qin, and H. Chunming, "Civic: A hypervisor based virtual computing environment," in *Proceedings of High Performance Computing and Communications (HPCC 2007)*, 2007, pp. 809–820.

[4] M. A Vouk, "Cloud computing–issues, research and implementations," *Journal of computing and information technology*, vol. 16, no. 4, pp. 235–246, 2008.

[5] M. Vouk et al., "Powered by vcl—using virtual computing laboratory (vcl) technology to power cloud computing," in *Proceedings of the 2nd International Conference on Virtual Computing (ICVCI), NC: RTP*, 2008, pp. 15–16.

[6] B. Sharma and R. Delhi, "Security architecture of cloud computing based on elliptic curve cryptography (ecc)," *International Journal of Advances in Engineering Sciences*, vol. 3, no. 3, pp. 58–61, 2013.

[7] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE cloud computing*, vol. 1, no. 3, pp. 81–84, 2014.

[8] Y. Tian, S. M. Khan, D. A. Jiménez, and G. H. Loh, "Last-level cache deduplication," in *Proceedings of the 28th ACM international conference on Supercomputing*, 2014, pp. 53–62.

[9] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[10] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[11] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[12] M. H. Syed, E. B. Fernandez, et al., "The software container pattern," in *Proceedings of the 22nd Conference on Pattern Languages of Programs*, 2015, pp. 24–26.

[13] C. Esposito, A. Castiglione, and K.-K. R. Choo, "Challenges in delivering software in the cloud as microservices," *IEEE Cloud Computing*, vol. 3, no. 5, pp. 10–14, 2016.

[14] F. Ramalho and A. Neto, "Virtualization at the network edge: A performance comparison," in *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, IEEE, 2016, pp. 1–6.

[15] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," in *Proceedings of the 17th international middleware conference*, 2016, pp. 1–13.

[16] W. Xia et al., "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[17] I. Mavridis and H. Karatza, "Performance and overhead study of containers running on top of virtual machines," in *2017 IEEE 19th Conference on Business Informatics (CBI)*, IEEE, vol. 2, 2017, pp. 32–38.

[18] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 22–32, 2017. DOI: 10.1109/MCC.2017.4250931.

[19] R. Kaur, I. Chana, and J. Bhattacharya, "Data deduplication techniques for efficient cloud storage management: A systematic review," *The Journal of Supercomputing*, vol. 74, pp. 2035–2085, 2018.

[20] A. Lingayat, R. R. Badre, and A. K. Gupta, "Performance evaluation for deploying docker containers on baremetal and virtual machine," in *2018 3rd International Conference on Communication and Electronics Systems (ICCES)*, IEEE, 2018, pp. 1019–1023.

[21] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, "A comparative study of containers and virtual machines in big data environment," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, IEEE, 2018, pp. 178–185.

[22] M. Chae, H. Lee, and K. Lee, "A performance comparison of linux containers and virtual machines using docker and kvm," *Cluster Computing*, vol. 22, no. Suppl 1, pp. 1765–1775, 2019.

[23] M. Chae, H. Lee, and K. Lee, "A performance comparison of linux containers and virtual machines using docker and kvm," *Cluster Computing*, vol. 22, no. Suppl 1, pp. 1765–1775, 2019.

[24] Q. Hu, J. Chen, D. Feng, Q. Yang, and B. Liu, "An efficient design and implementation of deduplication on open-channel ssds," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 562–569. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00087.

[25] I. Mavridis and H. Karatza, "Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing," *Future Generation Computer Systems*, vol. 94, pp. 674–696, 2019.

[26] S. C. Mondesire, A. Angelopoulou, S. Sirigampola, and B. Goldiez, "Combining virtualization and containerization to support interactive games and simulations on the cloud," *Simulation Modelling Practice and Theory*, vol. 93, pp. 233–244, 2019.

[27] R. Vestergaard, D. E. Lucani, and Q. Zhang, "Generalized deduplication: Lossless compression for large amounts of small iot data," in *European Wireless 2019; 25th European Wireless Conference*, VDE, 2019, pp. 1–5.

[28] R. Vestergaard, Q. Zhang, and D. E. Lucani, "Lossless compression of time series data with generalized deduplication," in *2019 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2019, pp. 1–6.

[29] H. Gantikow, S. Walter, and C. Reich, "Rootless containers with podman for hpc," in *International Conference on High Performance Computing*, Springer, 2020, pp. 343–354.

[30] C. Göttel, L. Nielsen, N. Yazdani, P. Felber, D. E. Lucani, and V. Schiavoni, "Hermes: Enabling energy-efficient iot networks with generalized deduplication," in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20, Montreal, Quebec, Canada: Association for Computing Machinery, 2020, pp. 133–136, ISBN: 9781450380287. DOI: 10.1145/3401025.3404098. [Online]. Available: https://doi.org/10.1145/3401025.3404098.

[31] A. M. Potdar, N. D G, S. Kengond, and M. M. Mulla, "Performance evaluation of docker container and virtual machine," *Procedia Computer Science*, vol. 171, pp. 1419–1428, 2020, Third International Conference on Computing and Network Communications (CoCoNet'19), ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2020.04.152. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050920311315.

[32] J. Takeshita, R. Karl, and T. Jung, "Secure single-server nearly-identical image deduplication," in *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, IEEE, 2020, pp. 1–6.

[33] O. Bystrov, R. Pacevič, and A. Kačeniauskas, "Performance of communication- and computation-intensive saas on the openstack cloud," *Applied Sciences*, vol. 11, no. 16, p. 7379, 2021.

[34] A. Leekha and A. Shaikh, "Detection and localization of content deduplication using 64-bit architecture of sha-256," in *2021 IEEE 6th International Conference on Computing, Communication and Automation (ICCCA)*, 2021, pp. 483–491. DOI: 10.1109/ICCCA52192.2021.9666267.

[35] B. Đorđević, V. Timčenko, M. Lazić, and N. Davidović, "Performance comparison of docker and podman container-based virtualization," in *2022 21st International Symposium INFOTEH-JAHORINA (INFOTEH)*, IEEE, 2022, pp. 1–6.

[36] A. Malviya and R. K. Dwivedi, "A comparative analysis of container orchestration tools in cloud computing," in *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, 2022, pp. 698–703. DOI: 10.23919/INDIACom54597.2022.9763171.

[37] P. Prajapati and P. Shah, "A review on secure data deduplication: Cloud storage security issue," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 7, pp. 3996–4007, 2022.

[38] H. Deshingkar et al., "Data deduplication using python," in *2023 7th International Conference On Computing, Communication, Control And Automation (ICCUBEA)*, 2023, pp. 1–3. DOI: 10.1109/ICCUBEA58933.2023.10391968.

[39] T. Jiang et al., "Fuzzydedup: Secure fuzzy deduplication for cloud storage," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 3, pp. 2466–2483, 2023. DOI: 10.1109/TDSC.2022.3185313.

[40] M. Kathiravan, R. Logeshwari, S. Pavithra, M. Meenakshi, V. Sathya Durga, and M. Vijayakumar, "A cloud based improved file handling and duplicate removal using md5," in *2023 Third International Conference on Artificial Intelligence and Smart Energy (ICAIS)*, 2023, pp. 1532–1536. DOI: 10.1109/ICAIS56108.2023.10073786.

[41] M. Alser et al., "Packaging and containerization of computational methods," *Nature Protocols*, vol. 19, no. 9, pp. 2529–2539, 2024.

[42] S. Jamil, J. Ro, J.-Y. Hwang, and Y. Kim, "Efficient data placement in deduplication enabled zenfs via crc-based prediction," *IEEE Access*, vol. 12, pp. 197 233–197 246, 2024. DOI: 10.1109/ACCESS.2024.3520184.

[43] S. Manoli, P. Metipatil, and P. Raghavendra Nayaka, "An efficient approach for vm and database segmentation of cloud resources over cloud computing," *SN Computer Science*, vol. 5, no. 8, p. 977, 2024.

[44] M. Pandya, "Performance evaluation of hashing algorithms on commodity hardware," *arXiv preprint arXiv:2407.08284*, 2024.

[45] R. Rani, N. Kumar, and M. Khurana, "Redundancy elimination in iot oriented big data: A survey, schemes, open challenges and future applications," *Cluster Computing*, vol. 27, no. 1, pp. 1063–1087, 2024.

[46] S. Russell, "Chorba: A novel crc32 implementation," *arXiv preprint arXiv:2412.16398*, 2024.

[47] D. Silva, J. Rafael, and A. Fonte, "Toward optimal virtualization: An updated comparative analysis of docker and lxd container technologies," *Computers*, vol. 13, no. 4, p. 94, 2024.

[48] A. K. Verma, R. Patel, P. Rai, and S. Patel, "Performance comparison between docker container and virtual machine microservices," in *2024 IEEE 13th International Conference on Communication Systems and Network Technologies (CSNT)*, 2024, pp. 718–723. DOI: 10.1109/CSNT60213.2024.10546015.

[49] H. Aqasizade, E. Ataie, and M. Bastam, "Experimental assessment of containers running on top of virtual machines," *IET networks*, vol. 14, no. 1, e12138, 2025.

[50] A. W. Services. "What is virtualization?" Accessed: 2025-10-01. [Online]. Available: https://aws.amazon.com/what-is/virtualization/.

[51] Z. Xiahui, N. Zhongying, and W. Licheng, "Application of simhash algorithm based on bucket index in deduplication of privacy data," in *2025 IEEE International Conference on Electronics, Energy Systems and Power Engineering (EESPE)*, 2025, pp. 6–9. DOI: 10.1109/EESPE63401.2025.10987512.

[52] G. Cloud. "What is a virtual machine? vm uses and benefits." Accessed: 2025-10-01. [Online]. Available: https://cloud.google.com/learn/what-is-a-virtual-machine.