UNIVERSITY OF BELGRADE
FACULTY OF MATHEMATICS

Ivan Ž. Ristović

DIRECT DATA-SNAPSHOTTING AND SNAPSHOT SHARING ACROSS CLOUD-NATIVE APPLICATIONS

Doctoral Dissertation

Belgrade, 2026.

UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET

Ivan Ž. Ristović

DIREKTNO SNIMANJE PODATAKA I DELJENJE SNIMAKA IZMEĐU APLIKACIJA U OBLAKU

doktorska disertacija

Beograd, 2026.

Advisor:

dr Milena Vujošević Janičić, associate profesor
University of Belgrade, Faculty of Mathematics

Commitee members:

dr Filip Marić, full profesor
University of Belgrade, Faculty of Mathematics

dr Mirko Spasić, assistant professor
University of Belgrade, Faculty of Mathematics

dr Rodrigo Bruno, assistant professor
University of Lisbon, Instituto Superior Técnico

Date of the defense: _____

Mentor:

dr Milena Vujošević Janičić, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

dr Filip Marić, redovni profesor
Univerzitet u Beogradu, Matematički fakultet

dr Mirko Spasić, docent
Univerzitet u Beogradu, Matematički fakultet

dr Rodrigo Bruno, docent
Univerzitet u Lisabonu, Instituto Superior Técnico

Datum odbrane: _____

To you, dear reader.

Dissertation title: Direct Data-Snapshotting and Snapshot Sharing Across Cloud-Native Applications

Abstract: Cloud-computing platforms provide services to consumers through multiple service-offering models. Recent advances in these models have led to the emergence of *serverless computing*, or simply *serverless*, where infrastructure is managed by the service provider. Serverless is usually coupled with function-based programming model in which software systems are composed of reusable, lightweight units of code executed within isolated sandboxed environments. Major cloud-computing platforms, including *Amazon Web Services (AWS)*, *Microsoft Azure*, and *Google Cloud*, report that a substantial proportion of their customers employ serverless solutions.

Most cloud-computing providers employ a *pay-as-you-go* billing model. Inefficient utilization of computing resources, particularly CPU time and working memory, which constitute the most costly resources, leads to increased overall operational costs. Moreover, the requirement for resource isolation adversely affects initialization latency and results in additional CPU and working-memory overhead. Serverless sandboxes are typically deployed on top of heavyweight virtualization stacks that include*Java*, *JavaScript*, or *Python* runtime environments with accompanying frameworks, further increasing working-memory consumption.

Modern cloud-computing architectures use Checkpoint/Restore (abbr. *C/R*) techniques to freeze initialized sandboxes into a continuable form. Such techniques, in combination with cloud-native deployments, allow the virtualized environment to optimize resource consumption and share code and pre-initialized data across multiple sandboxes. However, such solutions either operate at application-build time to support data pre-initialization or sharing, or operate at execution time with limited sharing potential for data available during application execution. Such data is processed multiple times and duplicated in each sandbox.

This dissertation presents Doss, a direct object snapshotting and sharing system that performs data C/R during application execution. Doss persists data directly, without transformations, into reusable and shareable snapshots. Direct snapshotting allows Doss to achieve near-constant data deserialization time, greatly improving initialization times and reducing CPU usage. Doss architecture enables snapshot sharing across application instances, eliminating the excess memory footprint associated with data re-processing and duplication.

GraalDoss, a Doss implementation for *Java*, is integrated into the *GraalVM* ecosystem. GraalDoss is evaluated using 106 correctness and robustness tests and a novel set of cloud-native micro and macro benchmarks that exercise real-world scenarios. A comprehensive evaluation of GraalDoss shows a consistent near-constant data-deserialization overhead with serialization times comparable to state-of-the-art *Java JSON* and binary serialization libraries. GraalDoss reduces the memory footprint of web API microservice caches by sharing populated cache snapshots across microservice instances, improving the overall density by 41% for 8 microservice instances and improving first-response times by 34%. In NLP applications, GraalDoss improves the pipeline execution times by six orders of magnitude by snapshotting pipeline results and subsequently loading the snapshots.

Keywords: cloud computing, microservices, serverless, data snapshotting, compilers, *GraalVM*

Research area: computer science

Research sub-area: compilers, programming languages, programming language implementations

UDC number: 004.415.5(043.3)

Naslov disertacije: Direktno snimanje podataka i deljenje snimaka između aplikacija u oblaku

Rezime: Platforme za izvršavanje u oblaku pružaju različite servise (eng. *services*) korisnicima koristeći razne modele isporuke usluga. Savremena istraživanja ovih modela dovela su do široke primene *bezserverskog* izvršavanja (eng. *serverless*), paradigme u kojoj se softver sastoji od *funkcija* — brzih, ponovno iskoristivih jedinica koda koje se izvršavaju unutar izolovanih, virtualizovanih okruženja. Vodeće platforme za izvršavanje u oblaku, uključujući *Amazon Web Services (AWS)*, *Microsoft Azure* i *Google Cloud*, izveštavaju da značajan deo njihovih korisnika koristi bezserverska rešenja.

Većina pružalaca servisa izvršavanja u oblaku primenjuje model naplate po principu „plati koliko koristiš". Neefikasno korišćenje računarskih resursa, naročito procesorskih jezgara i radne memorije, koji predstavljaju dva najskuplja resursa, dovodi do značajnog povećanja troškova. Dodatno, potreba za virtualizacijom negativno utiče na vreme inicijalizacije i povećava potrošnju procesorskog vremena i radne memorije. Izolovana bezserverska okruženja se tipično implementiraju nad okruženjima koja zahtevaju velike količine radne memorije, kao na primer virtualne mašine za jezike *Java*, *JavaScript* ili *Python* i odgovarajuci prateći radni okviri.

Savremene arhitekture za izvršavanje u oblaku koriste tehnike snimanja stanja inicijalizovanih okruženja u nastavljivom obliku (eng. *Checkpoint/Restore*). Ove tehnike omogućavaju optimizaciju iskorišćenja resursa, kao i deljenje koda i podataka između više okruženja za izvršavanje. Međutim, postojeća rešenja funkcionišu u fazi izgradnje aplikacija i, kao takva, ne omogućavaju ranu inicijalizaciju i deljenje podataka koji postaju dostupni u toku izvršavanja aplikacija. Takvi podaci se iznova obrađuju i dupliraju u svakom pojedinačnom izolovanom okruženju za izvršavanje.

Ova disertacija predstavlja Doss, sistem za snimanje i deljenje podataka koji omogućava snimanje i deljenje podataka tokom izvršavanja aplikacije. Doss direktno snima objekte koji sačinjavaju podatke, bez njihove transformacije, u obliku ponovno iskoristivih i deljivih snimaka (eng. *snapshot*). Takav pristup omogućava Doss-u da postigne konstantno vreme deserijalizacije podataka, čime se značajno unapređuje vreme inicijalizacije aplikacija i smanjuje potrošnja računarskih resursa. Arhitektura sistema Doss omogućava deljenje snimaka između više instanci aplikacije, čime se eliminiše memorijsko zauzeće povezano sa ponovnom obradom i dupliranjem podataka i poboljšava vreme inicijalizacije aplikacije.

Implementacija sistema Doss, pod nazivom GraalDoss, realizovana je u programskom jeziku *Java* i integrisana u ekosistem *GraalVM*. GraalDoss je evaluiran pomoću 106 testova korektnosti i robustnosti, kao i korišćenjem novog skupa referentnih programa namenjenih testiranju aplikacija u oblaku. Rezultati evaluacije pokazuju konzistentno vreme deserijalizacije, uz brzinu serijalizacije uporedivu sa savremenim *Java* bibliotekama za *JSON* i binarnu serijalizaciju. U mikroservisnim veb aplikacijama, GraalDoss eliminiše memorijsko zauzeće privremenih skladišta podataka deljenjem snimaka popunjenih skladišta između instanci mikroservisa, čime se postiže povećanje odnosa brzine i memorijeskog zauzeća sistema od 41% za osam instanci mikroservisa, uz smanjenje vremena odziva aplikacije od 34%. U aplikacijama koje koriste obradu prirodnih jezika, GraalDoss unapređuje vreme izvršavanja za šest redova veličine snimanjem modela za obradu teksta i rezultata njegove primene na ulazni tekst.

Ključne reči: Izvršavanje u oblaku, mikroservisi, bezserversko izvršavanje, snimanje podataka, kompilatori, *GraalVM*

Naučna oblast: računarstvo

Uža naučna oblast: kompilatori, programski jezici, implementacije programskih jezika

UDK broj: 004.415.5(043.3)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The first decade of the twenty-first century witnessed a rapid rise and widespread adoption of the Internet. This started the migration of computing resources and software to large farms of computing and storage systems accessible via the Internet, i.e., the *cloud*. Cloud computing introduced a paradigm shift from local to network-centric computing, where cloud computing providers provision and distribute computing resources to users through their cloud-compute platforms. Today, most popular cloud-computing platforms such as *Amazon Web Services* (abbr. *AWS*) [14], *Microsoft Azure* [243], and *Google Cloud* [141] generate hundreds of billions of USD in annual revenue, with over 400 billion USD generated in 2025 alone [376, 305].

Most software in the cloud is built on top of the *service-oriented* architecture, by composing independent and reusable software components called *services* to perform complex tasks [405, 213, 114]. Services are automatically deployed and scaled by the cloud-computing platform, offering great elasticity and scalability [83]. Modern software architectures decompose services even further, into smaller, rapidly scalable *microservices* [94]. Recent advances in cloud-computing architectures decouple microservices architecture from the infrastructure, allowing the user to write applications without managing the infrastructure. This model is commonly referred to as *serverless* [299, 234, 71].

Serverless applications consist of *functions* — lightweight and fast-executing snippets of code that run in isolated virtualized sandboxes. Serverless is especially popular for applications that involve big-data analytics [46, 233, 385], linear algebra algorithms [385, 330], machine learning [69, 382, 46, 179], media processing [126, 401, 26], and microservice web applications [342, 254]. Major cloud-computing providers popularize serverless deployments by enabling users to write and deploy function code that is decoupled from infrastructure concerns. In 2025, serverless solutions were used by 65% of *Amazon Web Services (AWS)* customers, 56% of *Microsoft Azure* customers, and 70% of *Google Cloud* customers [92].

The remainder of this Chapter is organized as follows. Section §1.1 outlines the most important factors that influence the cost of executing applications in the cloud, especially when the execution environment consists of heavyweight language-runtimes of commonly used programming languages. Section §1.2 describes popular approaches to improving the efficiency of cloud-computing applications and platforms, that typically leverage some form of Checkpoint/Restore techniques, and highlights the shortcomings of such solutions. Finally, Section §1.3 presents the scientific contributions of this work, whose outline is presented in Section §1.4.

1.1 Cloud computing cost

Most cloud-computing service providers utilize a *pay-as-you-go* billing model, with compute hardware and working memory being the most expensive resources that determine efficiency of the platform [226, 327, 169]. However, the need for service and function sandboxing decreases the overall efficiency. For example, serverless platforms that employ a function-based programming model such as *AWS Lambda* [20] allow users to write functions in a large variety of programming languages, with external libraries available for a particular use-case. Using popular languages such as *JavaScript*, *Python*, and *Java* requires heavyweight language runtimes (i.e., virtual machines such as *V8* [372] or *JVM* [225]) and large amounts of external libraries. Thus, the memory overhead of each sandbox far outweighs that of individual functions [322, 168, 106].

In addition to extra working memory spent on virtualized execution environments, reaching peak performance requires significant CPU work to initialize and warm up the execution environment. For managed programming languages such as *JavaScript*, *Python*, and *Java*, the language runtime needs to be started and initialized before it can execute code. Modern language runtimes employ the *just-in-time* compilation strategy [159], where the code is compiled during application execution for greater peak performance. Achieving peak performance incurs a warmup period during which the language-runtime interprets the code and determines hot paths before producing optimal machine code. For fast-executing services, such as serverless functions, the warmup period often exceeds the function lifetime [168, 371, 333, 201, 70].

Applications that run in the cloud, especially serverless applications, are inherently distributed and dependent on cloud services. Data is read from a storage medium, distributed and local memory caches, or a communication channel, often in a serialized format. Serialization is required as language-runtime instances that execute code in different sandboxes have incompatible representations of the data during execution (e.g., objects in the language-runtime heap). This incompatibility comes from different object addresses in the process address space and incompatible language-runtime states, including the set and order of loaded and initialized classes and the accompanying class metadata. As a result, loading serialized data not only incurs I/O overhead, but also CPU work needed to deserialize the data. Previous research acknowledged and highlighted the extreme overhead of network communication and data deserialization in cloud deployments [231, 135].

Storing data in a serialized format and sharing it across multiple application instances that use it inevitably leads to data duplication, as every application instance creates a different internal representation of the same data. Figure 1.1 shows this effect, where two application instances load and pre-process the same data, resulting in different data representations for both application instances. The same effect happens when the data is communicated across application instances via common communication protocols such as message queues or *RPC*.

Repeated I/O and CPU work, in combination with extra memory needed to represent duplicated data, reduces the overall efficiency of the cloud-computing platform, manifesting in longer startup times and reduced scalability. Optimizing cloud-computing costs presents a unique two-fold problem. First, applications need to start fast and reach peak performance as soon as possible. Second, the platform needs to spawn as many application instances as possible within certain working-memory restrictions. Improving startup times often results in higher working-memory requirements due to extra pre-initialization and caches. On the other hand, lowering the working-memory requirements by data sharing often results in using serialized data representations that require additional CPU work before use.

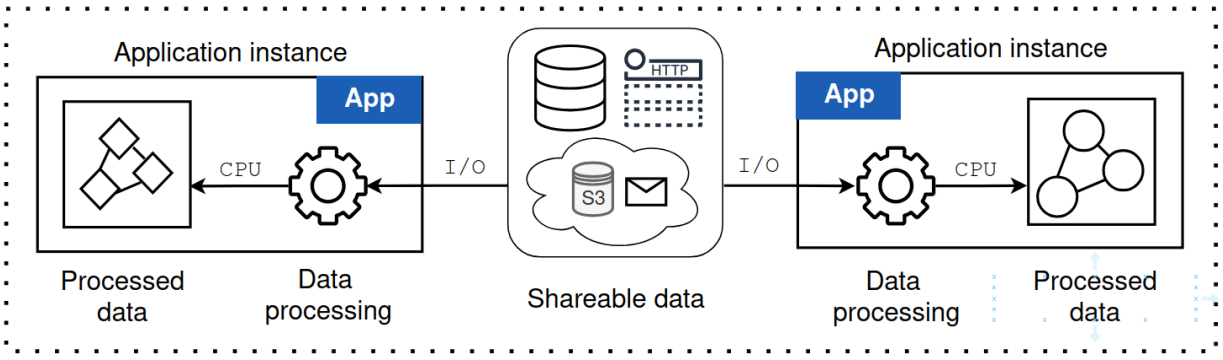


Figure 1.1: Incompatible data representations in separate application instances, as a result of repeated data pre-processing, deserialization, or communication.

1.2 Improving cloud-computing efficiency

Modern techniques for reducing application warmup times and memory footprint leverage ahead-of-time ((abbr. *AOT*)) [368] compilation strategies with cloud-native architecture [79, 214, 207, 40]. Cloud-native deployments eliminate long warm-up times associated with just-in-time ((abbr. *JIT*)) compilations during application execution and allow the compiled code to be shared across application instances. Since all the code is pre-compiled, there is no need for a heavyweight language runtime. Compiled binaries often depend on, or embed within themselves, a stripped-down version of the language runtime environment that includes essential components such as the garbage collector [271, 292]. Cloud-native applications also benefit from build-time initialization, where application components are pre-initialized during compilation [387].

Pre-initialized data in a cloud-native environment can be shared across application instances under the assumption that the data is available and can be initialized during compilation without altering the resulting application behavior. However, this assumption does not hold for data available only during application execution or data that requires real-time updates. Any changes to the pre-initialized data cannot be propagated to running application instances. In order for the platform to share updates to the data, it would need to stop all applications, rebuild them to pre-initialize updated data, and start all applications again.

Cloud-computing platforms often use Checkpoint/Restore (abbr. *C/R*) techniques to freeze the execution state in a warmed-up state snapshot. The persisted state can encompass the entire operating system state, the executing process state, along with process resources, or parts of the executing process. The warmed-up snapshot can then be continued at a later point, avoiding most of the warm-up overhead [175, 407]. Cloud-native and serverless architectures use C/R to speed up application and function startup times [178, 168]. Specialized C/R techniques can also be applied to the executing language runtime in its warmed-up state [331, 269, 265]. However, most C/R solutions operate on the entire language-runtime environment or its heap, lacking granularity and flexibility. As a result, although such solutions can replay a single application or language-runtime state, the restored state cannot be reused or shared across other language-runtime environments.

1.3 Dissertation contributions

This dissertation presents DOSS, a Direct Object Snapshotting and Sharing system for cloud-native Java applications. DOSS creates reusable and shareable checkpoints of an object graph directly from the executing language-runtime instance heap, i.e., without transformations. This allows DOSS to achieve constant-time deserialization overhead, eliminate data duplication or loss, and provide the ability to share snapshots across multiple language-runtime instances. DOSS augments the executing language runtime with a capability of hosting multiple DOSS snapshots. GraalDOSS, a DOSS reference implementation, is built on top of *GraalVM*, a closed-world solution to building cloud-native Java applications. This dissertation introduces the following contributions:

Direct Object Snapshotting and Sharing (DOSS) mechanism that performs object C/R during application execution by snapshotting objects directly from the runtime heap without transforming them. Such direct object snapshots capture a transitive closure of all objects reachable from the snapshot root object, allowing for arbitrary snapshot granularity and size.

An auxiliary shareable snapshot heap integrated into the language runtime that executes the application. The snapshot heap is divided into sections that are used as mediums for direct object snapshots during application execution. DOSS architecture allows the snapshot heap to share its sections across compatible language-runtime instances during cloud-native application execution.

Object laying-out strategies for persisted direct snapshots that are optimized for minimal memory footprint and memory fragmentation. DOSS integrates with the garbage collector of the executing language-runtime instance and supports linear and chunked object laying-out strategies.

Object hot-reloading during application execution via object snapshots, without the need for rebuilding or restarting the application. DOSS operates with multiple object snapshots simultaneously, and supports unloading and reloading snapshots during application execution, allowing for seamless updates and rollbacks of application data.

GraalDOSS, a reference DOSS implementation for *Java* cloud-native applications, integrated into *Oracle GraalVM 24.1* [271]. GraalDOSS architecture is highly scalable and extensible, allowing for a high degree of customization and specialization for external systems.

A novel C/R benchmark set for evaluating C/R and object serialization libraries that target the *Java virtual machine* (abbr. *JVM*), supporting both regular *JVM* and cloud-native deployment modes. The benchmark set consists of correctness and robustness unit tests for C/R solutions, serialization performance microbenchmarks, and macrobenchmarks involving web microservices and big-data applications.

Cold start elimination through instantaneous loading of pre-initialized application data persisted as a snapshot. GraalDOSS achieves minimal loading times by memory-mapping the snapshot into the application-process address space, eliminating costly deserialization overhead. GraalDOSS performance is evaluated on a wide-range of workload objects, against 6 state-of-the-art *Java JSON* [7, 146, 120, 257] and binary [272, 116] object

S/D libraries. In all experiments, GraalDOSS deserialization speed remained constant, several orders of magnitude faster compared to other frameworks as the workload size increases. In natural-language processing macrobenchmarks that use large-language models, GraalDOSS completely eliminated costly pipeline setup and input processing and reduced total execution times from minutes to milliseconds.

Constant data-memory overhead through sharing of application data snapshots across application instances. In microservice web API macrobenchmarks that use data caches, GraalDOSS reduced the total memory footprint of the application by sharing the warmed-up cache snapshot across application instances. Cache memory overhead remained constant when using GraalDOSS, compared to a linear increase in memory footprint in traditional deployments. For 8 application instances and 30 MB data cache, GraalDOSS improved the total memory footprint by 44% and first-response times by 34%.

The work on this dissertation and the research of relevant topics resulted in several contributions to national and international scientific journals [315, 313, 308], one United States patent [314], as well as multiple conference papers [87, 312] and abstracts [309, 311, 346, 190, 189, 347, 344].

1.4 Dissertation outline

This dissertation is organized as follows:

Chapter 2 (Cloud computing) covers the relevant background on cloud computing. It describes the need for virtualization and different virtualization techniques such as virtual machines and containers. It provides an overview of the service offering models and architectures of modern applications in the cloud, including monolithic, microservice, and cloud-native architectures. It highlights the primary factors that increase the cost of executing applications in the cloud and establishes the motivation for optimizing the cost of cloud computing.

Chapter 3 (*GraalVM* ecosystem) provides an overview of the *GraalVM* ecosystem and its components. It describes the *Graal* compiler, with an emphasis on *Native Image* ahead-of-time compiler and its compilation pipeline and components. It covers the *GraalVM* language runtime in detail, including its memory management and isolation techniques. It presents the use cases of *GraalVM* in modern cloud systems, including multi-language support and emerging serverless platforms.

Chapter 4 (Related Checkpoint/Restore (C/R) techniques) describes the state-of-the-art techniques that help reduce the cost of cloud computing by restoring the system or executing applications from a warmed-up checkpoint. It first describes system checkpointing techniques that capture the state of the entire operating system. It covers process checkpointing techniques that can be utilized to freeze entire language-runtime processes or their parts, such as their managed heap. It covers object serialization and deserialization in its various forms, from plain-text, binary, and schema-based serialization to serialization techniques accelerated by hardware and integrated in the language runtime.

Chapter 5 (Direct Object Snapshotting and Sharing system) presents DOSS, the system for managing and sharing of snapshots of objects taken directly from the runtime

heap during application execution, without transformations. It describes the core components of the system architecture and mechanisms in place that make direct snapshotting and sharing possible. It also provides an overview of the system operations for creating, storing, loading, and unloading snapshots during application execution, as well as object laying-out strategies and integration with the garbage collector. It presents several use cases for the direct object snapshotting and sharing system.

Chapter 6 (GraalDoss implementation) describes GraalDoss, the Doss prototype implementation integrated into the *GraalVM* language runtime. It contains the overview of the exposed GraalDoss API for managing direct object snapshots, as well as system options and usage examples. Finally, it provides an overview of the internal GraalDoss architecture.

Chapter 7 (GraalDoss evaluation) presents the setup and results of GraalDoss evaluation. It first presents the benchmark set used in experiments, which consists of various unit tests, microbenchmarks, and macrobenchmarks. The experiments evaluate core components of the GraalDoss system — its correctness, robustness, performance, and scaling. This chapter also presents an extensive qualitative comparison of GraalDoss with state-of-the-art C/R techniques and a quantitative comparison of GraalDoss performance against modern *Java* object S/D libraries.

Chapter 8 (Comparing Doss to other C/R techniques) presents both qualitative and quantitative comparison of Doss with modern C/R techniques. It highlights innovative features introduced by Doss and presents the results of the operational performance comparison of GraalDoss compared to state-of-the-art C/R frameworks in *Java*.

Chapter 9 (Discussion) discusses the proposed Doss system and GraalDoss prototype implementation, as well as results obtained in the evaluation of GraalDoss. It covers threats to the validity of the obtained results, including internal and external threats. Additionally, it examines potential security implications of integrating GraalDoss, including the most common vulnerabilities in modern cloud deployments, and GraalDoss-specific security concerns. Finally, the chapter discusses latency-sensitive workloads, their specific requirements, and the viability of GraalDoss when operating in such environments.

Chapter 10 (Conclusions and future work) summarizes the dissertation conclusions and presents future research directions based on the obtained evaluation results. This chapter also includes an overview of scientific contributions and publications involving the research on direct object snapshotting and sharing, and other topics related to it.

Appendix A (Detailed microbenchmark results) presents detailed results of evaluating GraalDoss against state-of-the-art *Java* serialization libraries. It contains a serialization throughput and snapshot-size comparison for every library on the entire set of workloads contained in the benchmark set used for GraalDoss evaluation.

Chapter 2

Cloud computing

Cloud computing, or simply *cloud* [83, 296, 157, 115, 38], is defined by the *International Standards Organization* (abbr. *ISO*) [165] as a paradigm for enabling network access to a scalable and elastic pool of shareable physical or virtual resources with self-service provisioning and administration on demand [166]. Cloud-computing resources can be software-based, such as virtual servers or custom software programs, or hardware-based, such as physical servers or network devices.

A *cloud provider* is an entity that provides cloud-based computing resources and services through a cloud-computing *platform* [219, 24, 282, 285, 321]. The most popular cloud-computing platforms include *Amazon Web Services* (abbr. *AWS*) [14], *Microsoft Azure* [243], *Google Cloud* [141], *Alibaba Cloud* [6], *Oracle Cloud Infrastructure* [273], *IBM Cloud* [160], *Salesforce Cloud* [318], *Tencent Cloud* [357]. In addition, numerous providers offer specialized cloud services tailored to specific domains [99, 80, 4, 77, 95, 317, 185, 295, 37, 274]. Cloud computing industry generated over 400 billion USD in 2025 [376, 305], with *AWS*, *Microsoft Azure*, and *Google Cloud* accounting for over 60% of the worldwide market shares in the second quarter of the 2025 fiscal year, illustrated in Figure 2.1.

National Institute of Standards and Technology (abbr. *NIST*) [255] identified the essential cloud system characteristics [236], which were later refined and expanded by the *ISO* [167]:

On-demand self-service — consumers request computing capabilities that are provisioned automatically as needed. This results in an *on-demand* usage environment, also known as *on-demand self-service usage*.

Broad network access — consumers access platform capabilities through standard mechanisms and interfaces. Establishing ubiquitous access for a cloud service can require support for a range of devices, transport protocols, interfaces, and security technologies.

Resource pooling — computing resources are pooled with the goal of serving multiple consumers, with physical and virtual resources dynamically assigned according to the demand.

Rapid elasticity — provisioning and releasing platform capabilities is automated and seamless, with the purpose of scaling on demand. *Elasticity* is the automated ability of a cloud to transparently scale resources, as required in response to runtime conditions or as predetermined by the provider or the consumer. To the consumer, the platform capabilities often appear unlimited and can be provisioned in any quantity at any time.

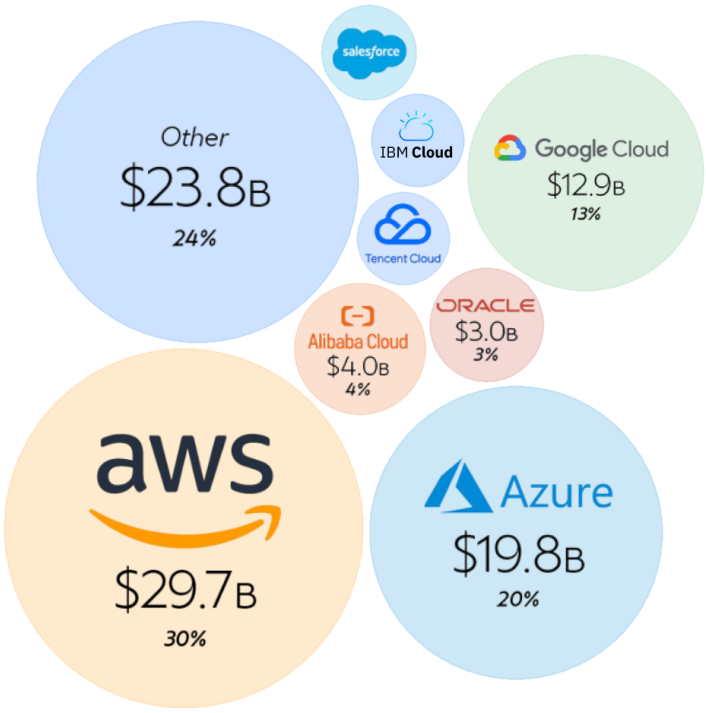


Figure 2.1: The world's largest cloud-computing providers, ranked by market share in Q2 FY2025.

Measured service control — the system automatically collect telemetry in order to control and optimize resource usage. The cloud provider can charge a consumer only for the resources actually used and/or for the time frame during which access to the resources was granted.

Resiliency to failures — the system distributes redundant implementations of computing resources across multiple physical locations. If a resource becomes deficient, processing is automatically handed over to another redundant implementation, resulting in higher availability and reliability of the platform.

Due to its reliability and scalability, cloud computing industry provides services for a wide range of use-cases, including big-data analysis [46, 233, 385], linear algebra [385, 330], machine learning [69, 382, 46, 179], media processing [126, 401, 26], and more [107]. Services are deployed and scaled automatically [209, 289, 256], resulting in high elasticity [11] and resiliency [28, 184].

Cloud architecture defines the fundamental components of a cloud computing environment — the front end, the back end, the networking, and the service offering model [84, 83, 115]. It describes the connection between components that enables the system to execute and provide services to one or more *tenants*. A cloud *tenant* is a user or a group of users that share computing resources. Cloud-computing platforms use both *single-tenant* and *multi-tenant* architectures. Single-tenant architectures emphasize greater control and higher levels of isolation [192]. Multi-tenant architectures require cost-efficient startup and the ability to scale and offload resources on demand [134].

Cloud-computing providers use virtualization techniques (§2.1) in various service offering models to deliver services (§2.2) and software (§2.3) to consumers. The pricing model used by public cloud providers is typically based on utility-centric pay-per-usage models, with several operational metrics impacting the overall performance and cost (§2.4).

2.1 Virtualization

In order to provide cloud services to consumers, cloud providers use combinations of hardware and software components. These components create a *virtualized* environment in which operating systems and applications are executed, which is known as the *virtualization stack* [287]. The operating system that runs the virtualization stack is called the *host*, whereas a virtualized environment is called a *guest*. The virtualization stack enables efficient utilization of resources, enhanced isolation and security, and improved scalability and elasticity of the cloud [171].

Depending on the virtualization technique in place, the virtualization stack can consist of different components [74]. For example, a common virtualization stack running on a *Linux* kernel [203, 91] might consist of one or more of the following components:

Hypervisor or *virtual-machine monitor* (abbr. *VMM*) or *virtualizer* such as *Xen* [396] serves as an intermediary between the virtualized guest systems and the physical hardware. It allocates physical resources and distributes them to each guest, keeping guests isolated. Hypervisors can run directly on the physical hardware (*type-1* or *bare-metal* hypervisors) [396, 354, 73, 195], or on top of a host operating system (*type-2* or *hosted* hypervisors) [60, 129, 82, 279].

Guest emulator such as *QEMU* [48] interprets the instructions from the guest system and translates them into instructions compatible with the host system.

Virtual machine manager such as *libvirt* and *virt-manager* [91] contain libraries, tools, and user interfaces that manage virtualized guests.

Container manager such as *Docker* [238] runs applications and their dependencies in lightweight virtualized environments called *containers* that share the host operating system kernel.

Language runtime such as the *Java virtual machine* (abbr. *JVM*) [225] or *.NET common language runtime* [58] executes applications that require a managed runtime environment. The language runtime can be integrated into the rest of the virtualization stack for improved efficiency [168, 5, 331, 2, 201, 70].

Figure 2.2 shows an example of a typical virtualization stack using virtual machines (left) and containers (right) to execute applications.

2.2 Service offering models

Cloud-computing providers offer specific and pre-packaged combinations of cloud-computing resources as part of the *offering model*. These resources are commonly referred to as *service offering model* or simply *service model* [115]. The most widely adopted service offering models in the cloud are *Infrastructure-as-a-Service*, *Platform-as-a-Service*, and *Software-as-a-Service*. Other service models subcategorize or extend these models, as is the case for the *Function-as-a-Service* and *serverless* models. Figure 2.3 illustrates how different resources are provided through different service offering models [300, 320].

Infrastructure-as-a-Service

The *Infrastructure-as-a-Service* (abbr. *IaaS*) offering model represents a self-contained environment comprised of infrastructure-centered computing resources, including servers, network,

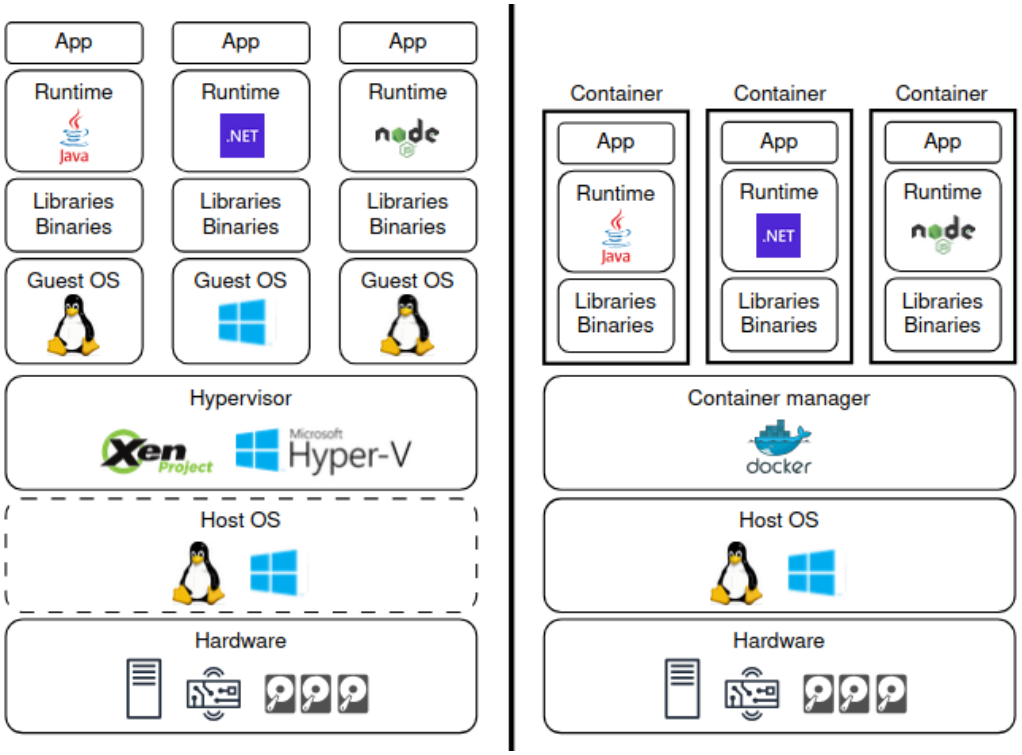


Figure 2.2: Virtualization stacks leveraging virtualized guest operating systems (left) and containers (right).

operating system, storage, and virtualization [100, 296, 163, 173]. These resources can be managed through the cloud-provider service interface, often virtualized and packaged into bundles for better scalability [115].

IaaS offering model provides a high level of control and isolation, which are traits desirable for *single-tenant* architectures. Consumers purchase and manage resources on their own, with manual resource scaling and application deployment. This enables consumers to scale resources on an *as-needed* basis, which is desirable for high-performance workloads.

IaaS resources can be grouped into several categories:

- *Compute* resources include CPUs, GPUs, and internal memory (RAM),
- *Storage* resources include block storage such as SSD or HDD, remote storage such as network attached storage (abbr. *NAS*) [261], and object storage such as *S3* [13], and
- *Networking* resources like routers, switches, and load balancers.

For example, *AWS* [14] provides various *IaaS* resources, such as *Elastic Compute Cloud* (abbr. *EC2*) [10], *Elastic Kubernetes Service* (abbr. *EKS*) [18], *Simple Storage Service* (abbr. *S3*) [13], *Virtual Private Cloud* (abbr. *VPC*) [19], *Elastic Block Storage* (abbr. *EBS*) [16], and *Relational Database Service* (abbr. *RDS*) [12].

IaaS model includes support for *containerization* [323, 238, 153] of applications into a single lightweight container that runs on any infrastructure. *Containers-as-a-Service* (abbr. *CaaS*) emerged as an extension of the *IaaS* model centered around containers [326, 158]. Compared to *IaaS*, *CaaS* offers more efficient resource utilization due to the lightweight nature of containers, in exchange for less control over the infrastructure and reduced security compared to VM-level isolation.

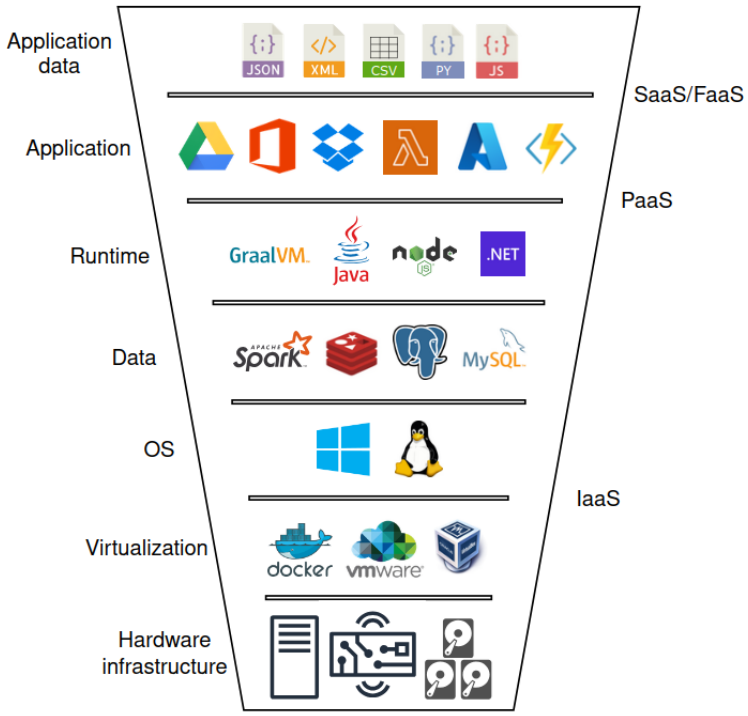


Figure 2.3: Cloud-computing resources (left) and offering models (right).

Platform-as-a-Service

The *Platform-as-a-Service* [275, 278, 337, 375, 381] offering model represents a pre-configured environment comprised of deployed and configured resources, serving as a *platform* where applications are developed, executed, and managed. *PaaS* removes the need for consumers to build and maintain the infrastructure and the platform, resulting in faster setup, development, and deployment.

Most common use-cases for *PaaS* deployments include:

- *Rapid software development* without infrastructure management,
- *Analytics or business intelligence* scenarios, including data collection, analytics, and visualization, and
- *Management services* such as database, API, integration, automation, security, and identity management.

For example, *AWS* [14] provides various *PaaS* platforms, such as *API Gateway* [15], *DynamoDB* [9], *Elastic Beanstalk* [22], and *App Runner* [17].

Software-as-a-Service

Software-as-a-Service (abbr. *SaaS*) [89, 227, 353, 211, 250, 366] offering model provides cloud applications and software to consumers. *SaaS* model is used to make a reusable cloud service or application widely available to a range of consumers, typically through a subscription-based billing model.

Most notable features that make *SaaS* a popular choice for businesses and end-users [50, 392, 393, 403, 263, 218] include:

High accessibility — applications can be accessed from any device with an internet connection, such as a computer or a mobile phone, usually through a web application interface accessible using a web browser,

Low maintenance — the provider takes care of updates, migrations, and maintenance, resulting in minimal maintenance overhead for consumers, and

Cost savings — the only cost to a business deployed on a *SaaS* model is the cost of the subscription, everything else is handled by the provider.

Examples of products running on the *SaaS* model include:

- business communication and conference applications such as *Slack* [339], *Zoom* [413], *Microsoft Teams* [245], *Cisco Webex* [78], and *Google Meet* [145],
- management and organization tools such as *Confluence* [34], *Trello* [365], *Asana* [32], and *Notion* [260],
- development and collaboration services include version control systems such as *GitHub* [132], *GitLab* [133], *BitBucket* [33], *JIRA* [35], and *Codeberg* [81], as well as continuous integration tools such as *Jenkins* [176] and *Travis* [364],
- file storage services such as *Google Drive* [143] or *Microsoft OneDrive* [244],
- communication services such as e-mail clients such as *Google Mail* (abbr. *Gmail*) [140] and *Microsoft Outlook* [247], or social-media services such as *X* [395] and *Facebook* [241], and
- document and media editing applications such as *Microsoft Office 365* [246] and *Google Docs* [142].

Function-as-a-Service

Function-as-a-Service (abbr. *FaaS*) [210, 355] is a deployment model that allows consumers to execute reusable snippets of code called *functions* or *lambdas*, in response to events and triggers provided by the application framework. *FaaS* embraces the *event-driven* model [90], where application logic is split into event-processing actions defined through *functions* that are executed by one or more *workers*.

Serverless computing, or simply *serverless* [234, 328, 71, 324, 222, 180], is a service offering model that provides *FaaS* capabilities to build and manage cloud applications, along with the instant, automatic, and on-demand scaling of computing resources without interventions from the consumer. Multiple serverless performance considerations [1, 327, 224, 70], surveys [151, 223, 112], and extensions [41] have been proposed in recent years, making serverless an attractive deployment model for common use-cases [112].

Most popular *FaaS* and serverless platforms include *AWS* [14] *Lambda* [21] and *Fargate* [20], *Microsoft Azure Functions* [243], *Google Cloud Run* [136], *Apache OpenWhisk* [29], *OpenLambda* [152], *OpenFaaS* [266], and *IronFunctions* [164]. Serverless architectures [299, 298, 212] are used for various use-cases [233, 113, 111, 208], including big-data analytics [46, 385], linear algebra [330], machine-learning workflows [69, 382, 179], and media processing [126, 401, 26].

2.3 Software in the cloud

To fully utilize the benefits of cloud computing, the software itself needs to be developed and deployed in a way that best utilizes those benefits. Different software-architecture styles can be used depending on the environment, scaling capabilities, available resources, and other requirements [373]. Most prevalent general software-architecture styles are the *monolithic architecture*, *service-oriented architecture*, and *microservices architecture*, with specialized architecture types emerging for specific use-cases and environments, one of which is the newly emerging *cloud-native* architecture.

Monolithic architecture

A *monolithic application*, or simply *monolith*, is designed and deployed as a single unit, self-contained and isolated from other applications [162, 117, 121]. All components of a monolithic application, including the user interface, business logic, and data storage, are integrated into a single codebase and deployed as a whole. Figure 2.4 shows an example of an architecture for a monolithic application that provides team collaboration tools.

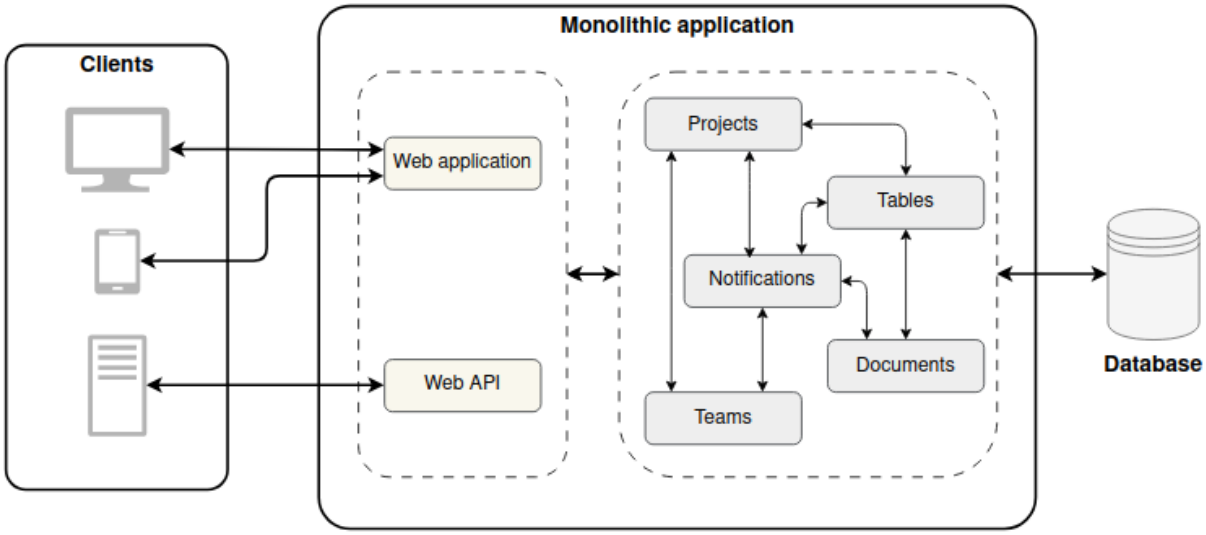


Figure 2.4: A monolithic architecture example of a team-collaboration application.

Monolithic design most often results in strong interdependence among application components, where changing one component often affects other components [127]. Similarly, changing one component results in rebuilding and redeploying the entire monolith. Furthermore, maintaining a monolithic application is difficult since every component might require its own set of dependencies [128]. Because of tight coupling between components, monoliths are scaled as a whole, on the application level, rather than at the component level.

Despite their drawbacks, monoliths are still a good choice for small and simple applications and teams, in situations where a monolithic design eases software development and deployment. The simplicity of monolithic architecture, when possible to leverage, often results in higher performance compared to other architectures, as there is no overhead of communication between components [170].

Service-oriented architecture

Service-oriented architecture (abbr. *SOA*) is an architectural style of organizing sets of capabilities into reusable and independent software components called *services* [405, 213, 114]. Services are often distributed across the network and can be controlled by different ownership domains. Services communicate with each other through well-defined interfaces and protocols to perform complex logic.

Contrary to the monolithic architecture, loosely-coupled services are easier to maintain, develop, and scale. Each service can be deployed and scaled independently, which makes *SOA* useful for many cloud service offering models. However, loosely-coupled services might require an external orchestrator or messaging queues to communicate, impacting performance [52] and adding complexity to the application as a whole.

Microservices architecture

Microservice architecture [187, 342, 254] introduces a more granular approach to building services compared to *SOA*, where each *microservice* focuses on a single business capability. Microservices are designed to be agile, highly scalable, reusable, and fault-tolerant. Similarly to *SOA*, microservices communicate via lightweight protocols, usually *HTTP*, *REST*, *RPC*, and messaging queues [259]. Figure 2.5 shows an example of a microservice application for team collaboration.

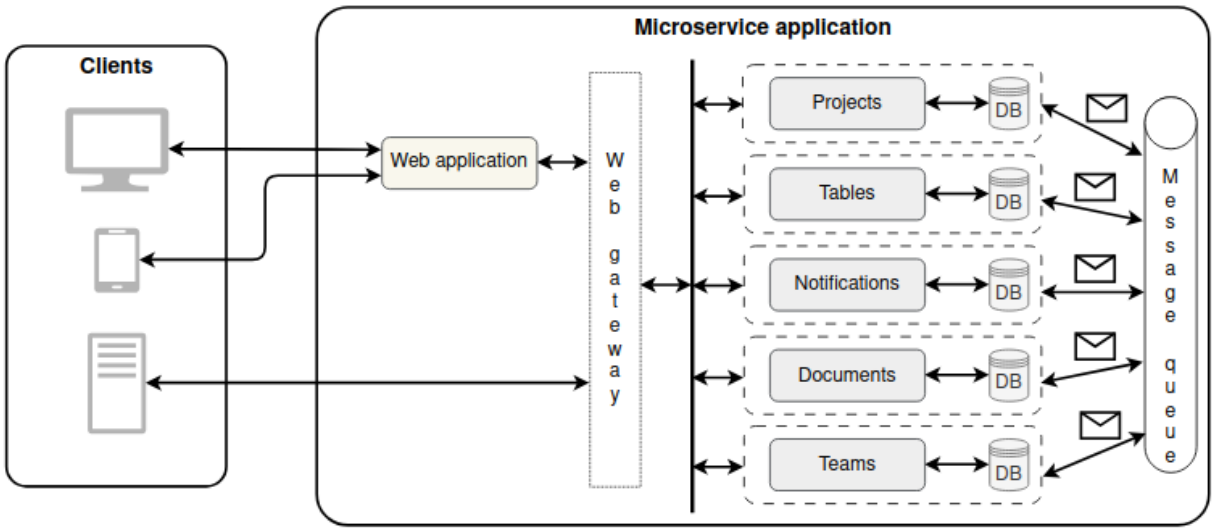


Figure 2.5: A microservice architecture example of a team-collaboration application.

Previous studies have analyzed the microservices architecture [94, 101], quality [221], performance [187, 178, 259], and challenges [383, 228], enabling efficient adoption of microservices for the cloud. In cloud computing deployment models, microservices are often containerized [228, 238, 3] and deployed, managed, and scaled automatically by an orchestrator such as *Kubernetes* [289, 199].

Cloud-native architectures

Cloud-native [79] architectures are modern architectures that fully leverage the advantages of cloud computing in the application design and utilize function-based programming model as an

evolution of microservices architecture [297, 40]. The principles of cloud-native architectural styles include *cloud-native services*, *application-centric design*, and *automation* [79, 214]. Cloud-native services are built to be resilient, stateless, rapidly scalable, and optimized for high efficiency [8].

Cloud-native architectures rely on the serverless deployment model and microservices architecture for efficient scaling and resiliency [39, 207, 199]. Cloud-native applications and services are optimized for fast startup and low memory footprint, supported by the executing language runtime [168, 5, 262, 371]. Research into cloud-native serverless runtimes includes modifications that de-bloat the virtualization stack and reduce execution costs [27, 65, 322, 216].

Figure 2.6 shows an example of a cloud-native architecture applied to serverless or *FaaS* offering models. Worker instances execute functions in multiple isolated language-runtime instances, with each instance loading code, configuration, and various forms of data. Typically, all language-runtime instances share code, whereas each language-runtime instance has its own private heap for objects created during function execution.

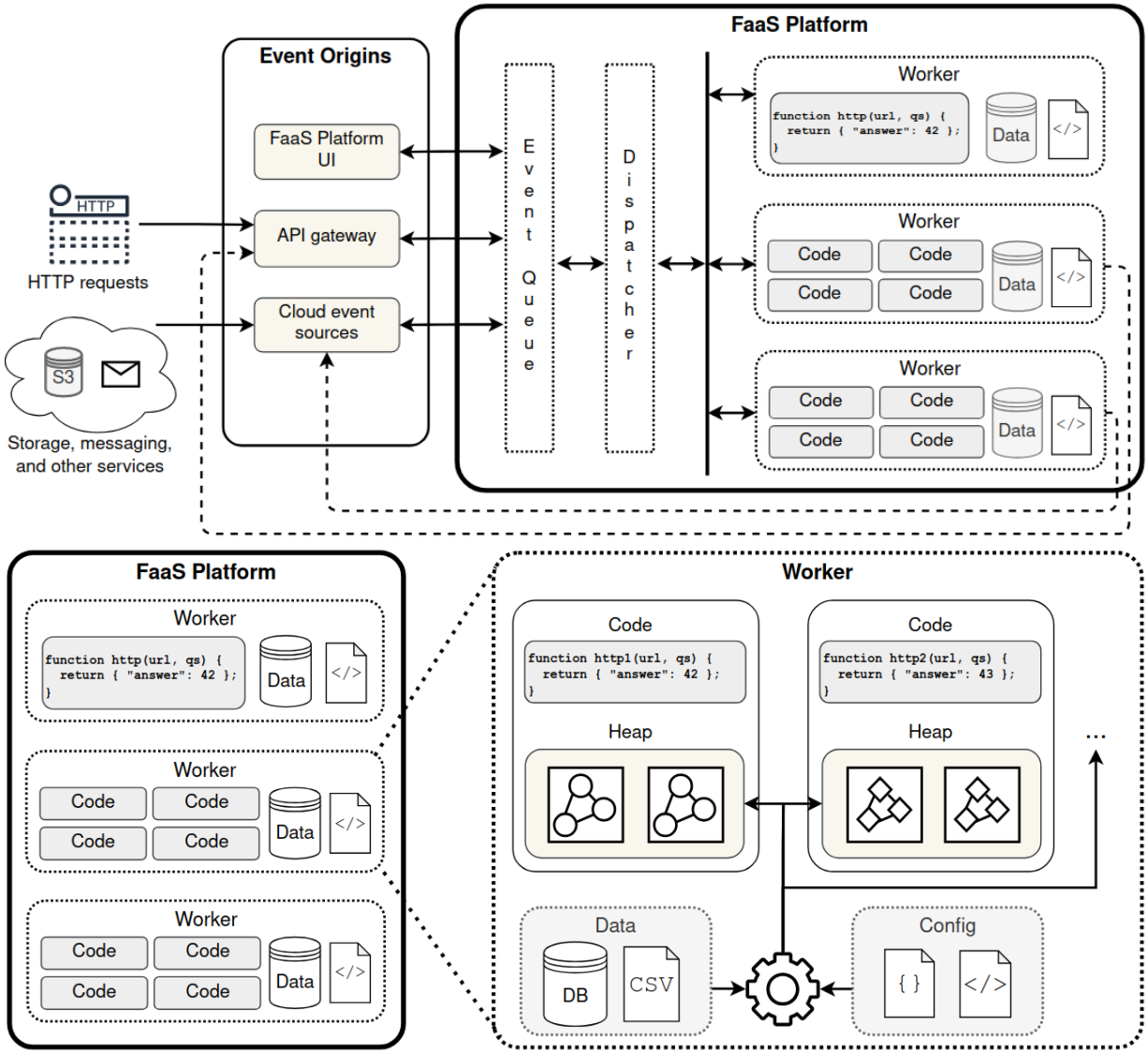


Figure 2.6: Cloud-native architecture example. Workers in the *FaaS* platform execute functions in isolated language-runtime instances, each with its own private heap.

2.4 Cost optimization challenges

The cost of computing in the cloud is fixed for most service offering models, as provisioned resources are billed in advance, e.g., the infrastructure (*IaaS*), platform (*PaaS*), or software (*SaaS*). However, most serverless and *FaaS* providers utilize a *pay-as-you-go* billing model, where the computing costs [226] directly corresponds to resource usage [100, 214]. Billed resources include:

- *compute time*, corresponding to the CPU or GPU wall-clock execution times or cycles,
- *memory footprint* reflected in the amount of allocated or consumed working memory (RAM), whose increments often range from 1 to 64 MB in practice, with fixed sizes available as well, often measured in GB, and
- *storage*, including *ephemeral*[1] [198] and *persisted* storage, measured in GB or TB.

CPU compute time and working memory are the two most expensive resources [100], and optimizations tend to either improve performance [224, 168], or reduce the overall memory footprint of the system [322, 168]. Compute time is often billed in terms of wall-clock execution time, measured in vCPU-seconds, where vCPU stands for virtualized CPU core provided to the worker. Working memory is often measured and billed in gigabyte-seconds.

The efficiency of a cloud-computing system is often expressed in terms of:

Throughput — the number of performed operations per second ($\frac{ops}{s}$), where the operation depends on the context, e.g., a request-processing operation in microservice web applications.

Memory footprint — includes the total system *resident set size* (abbr. *RSS*), used set size (abbr. *USS*), virtual set size (abbr. *VSS*), and proportional set size (abbr. *PSS*) [336], measured in megabytes (MB).

Density — throughput per provisioned memory, measured in operations per second per gigabyte, or, equivalently, the amount of operations per gigbabyte-second ($ops/(GB \cdot s)$).

Startup time — application initialization time, e.g., for a web server, the time it takes for the server to be ready to process requests. It is often measured in milliseconds (ms) or even seconds for larger systems (s).

Latency — the duration that a request is waiting to be handled, during which it is *latent*, or awaiting service [197]. Latency is often measured in common time units depending on the context, and is most often expressed in milliseconds (ms).

Response time — the time between a client sending a request and receiving a response. It is the sum of round-trip latency, request-processing delays, and queueing delays. It is often measured in milliseconds (ms) or even seconds (s) for large server-side processing pipelines (s).

First response time (abbr. *FRT*) — server-side processing time for the first request. It is often measured in milliseconds (ms) or even seconds (s) for large server-side processing pipelines (s).

[1] Ephemeral storage is a temporary storage allocated to virtual machines, containers, or serverless functions, for the duration of their lifecycle.

Time to first response — combined startup time and the first response time (FRT). It is often measured in milliseconds (ms) or seconds (s), depending on the units chosen for startup and first-response times.

The process of reaching optimal system performance is referred to as *warming up*, and consists of different steps depending on the environment. For the virtualization stack, warming up means allocating resources and creating a virtualized sandbox. For the language runtimes such as the JVM [225], warming up consists of loading all relevant classes and generating highly optimized code [169]. The service or a function may itself require data to be loaded and initialized during execution, e.g., configuration, in-memory caches, datasets, machine-learning models, etc.

A *cold start* is an application run that performs the warm-up process. Cold starts of language runtimes can be unpredictable [44] and an order of magnitude slower than subsequent executions [5]. Avoiding cold starts is difficult, especially in serverless environments, as functions are executed in isolated virtualized language-runtime environments whose initialization dominates the function execution time [229, 201, 70]. In the context of application warm-up, a *cold start* refers to initialization of data available only during application execution. For cloud-native environments, performing data pre-initialization and optimizing language-runtime memory footprint can be achieved through specialized language runtimes tailored for cloud execution, such as *GraalVM* (§3). Additionally, Checkpoint/Restore techniques (§4) can help avoid cold starts by freezing a warmed-up state of various components of the virtualization stack.

Chapter 3

GraalVM ecosystem

GraalVM [271] is an advanced *Java* development kit (abbr. *JDK*) designed for high performance and minimal memory footprint. An overview of the *GraalVM* ecosystem is shown in Figure 3.1. *GraalVM* consists of the *Graal* (§3.1) and *Native Image* (§3.2) compilers, *GraalVM* language runtime for execution of JVM-based languages (§3.3), and *Truffle* language implementation framework for execution of dynamic and *LLVM*-based languages [215] on top of *GraalVM* (§3.4). *GraalVM* is used in production as a driver for the *GraalOS* cloud-native environment (§3.5), where it improves application startup times and reduces the overall memory footprint of the system.

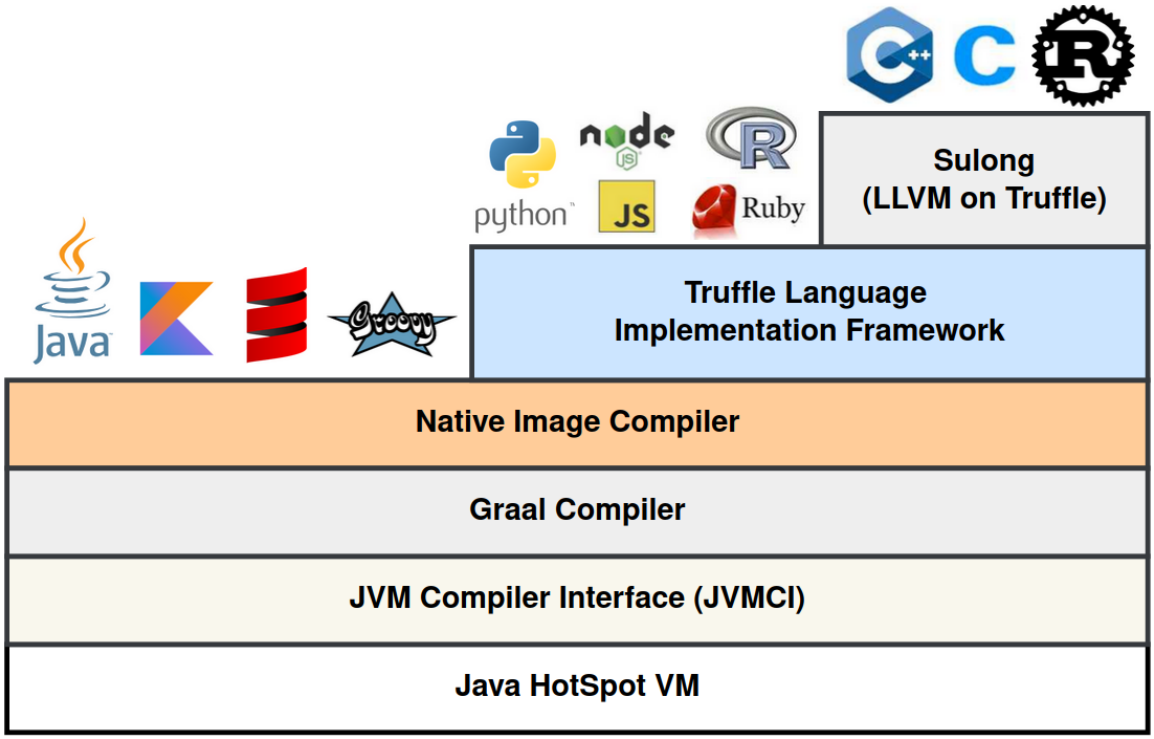


Figure 3.1: An overview of the *GraalVM* ecosystem.

3.1 *Graal* compiler

The *Graal* compiler [104, 105] is a dynamic compiler [36] written in *Java* that transforms bytecode into machine code. *Graal* supports *just-in-time* (abbr. *JIT*) compilation [159] of *Java* applications, and serves as a foundation for *ahead-of-time* (abbr. *AOT*) [368] compilation in combination with other *GraalVM* frameworks and tools.

Graal compiler can be used as a replacement for *JIT* compilers via the *JVM Compiler Interface* (abbr. *JVMCI*) [177], commonly done for programs running on the *HotSpot JVM* [276, 225]. *Graal* uses an internal representation (abbr. *IR*) called *sea of nodes* [105]. During compilation of a method, *Graal* transforms the code into its IR, and performs optimizations in *phases*, including various JIT, machine-learning, and profile-guided optimizations [88, 248, 232, 293, 380].

Graal can also be used as a backend for *AOT* compilation with *GraalVM Native Image* compiler. Using *Native Image*, *Graal* can itself be compiled into a shared library object that can link with other native executables. This means that, despite being written in *Java*, *Graal* can run without the *JVM*, retaining all the benefits of the *Java* programming language, such as improved safety, easier development and maintenance, and high portability.

3.2 *Native Image* compiler

GraalVM Native Image, or simply *Native Image* [271], is an ahead-of-time compiler that takes *Java* bytecode as an input and produces a *native executable* called *native image*, or simply *image* [387]. A native image includes all of the required code and resources needed for application execution. That includes application classes, standard-library classes, statically-linked native code from the *JDK*, and a lightweight language runtime that executes the code. The resulting native image is cheaper to run compared to the standard *JVM* execution, starts faster, requires no warm-up for peak performance, and can be packaged into a lightweight container.

The process of building native images is referred to as *image-build* time. Figure 3.2 shows an overview of the *Native Image* compilation pipeline that consists of various steps, such as *points-to analysis* for determining reachable code, class initialization, heap snapshotting, AOT compilation, method-code laying-out, and heap writing [387]. The AOT-compiled code is placed in the text section of the generated image. The heap snapshot containing pre-initialized application objects, otherwise referred to as the *image heap*, is written into the data section of the generated image.

The timespan that corresponds to the execution time of native images is referred to as *image-run* time. The lightweight *GraalVM* language runtime (§3.3) contained in a native image is first initialized, before executing the application code. The *GraalVM* language runtime manages application execution, providing a heap for object allocation and integrating with a garbage collector.

Points-to analysis

GraalVM Native Image compiler uses a *closed-world* assumption, i.e., it assumes that all *Java* classes, configuration, resources, and other pre-requisites for application execution are known at image-build time [387]. *Native Image* scans the application and all its libraries to find reachable program elements and compiles only the methods necessary for application execution. This

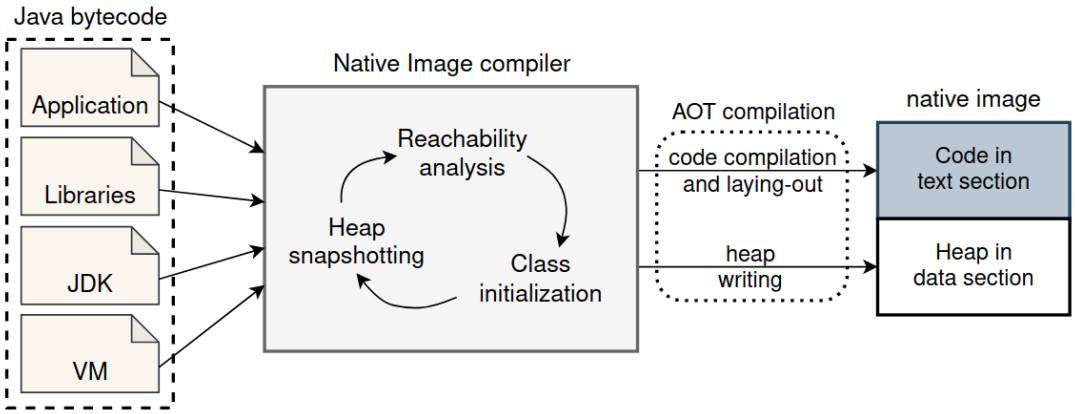


Figure 3.2: An overview of the *GraalVM Native Image* compilation pipeline.

approach reduces the memory footprint of applications that heavily rely on library code, which is often the case in *SaaS* and *FaaS* service offering models [368, 387, 212, 328, 229].

Native Image uses a technique called *points-to analysis* as an implementation of the whole-program application scanning [345], with recent research focusing on other variants and improvements to the *points-to* analysis that optimize for a large number of reachable elements, such as the *rapid-type* analysis [388, 204] and *SkipFlow* analysis [205].

During points-to analysis, application code, library code, *JDK* code, and language-runtime code are scanned iteratively, and reachable classes, methods, and fields are recorded. In each step, *Native Image* runs static initializers of newly reachable classes, and stores resulting objects in a special heap snapshot called the *image heap*. This process is repeated until no new reachable elements have been discovered. Reachable method code is compiled and laid out into the text section of the generated image. The resulting image heap is written into the data section of the generated image.

Class initialization

The semantics of *Java* require that the class is initialized the first time it is accessed during program execution [225]. Even a simple program in *Java*, such as *Hello World*, loads over one thousand classes [225, 387]. To reduce the negative impact of class initialization at image-run time, *Native Image* supports class initialization at image-build time.

Native Image initializes most *JDK* classes at image-build time, as well as *GraalVM* language-runtime implementation classes, such as the garbage collector. Classes initialized during the compilation process are referred to as *build-time initialized classes*. However, not all classes are safe to initialize at image-build time. Any class whose fields capture the state that depends on the moment the initialization occurred, such as current time, random seed, file descriptors, or thread identifiers, is unsafe for build-time initialization. The goal of *Native Image* is to initialize as many classes as possible at image-build time, while preserving application semantics.

Classes that are marked as unsafe for build-time initialization are referred to as *run-time initialized classes*. These include application classes that invoke native methods, methods that cannot be reduced to a single target (i.e., *virtual* methods), and specific *Native Image* implementation classes. Such classes are initialized at image-run time, to preserve *Java* semantics.

Image heap

During heap snapshotting, *Native Image* builds an *image heap*, comprising a transitive closure of all reachable objects starting with *root* objects, such as static field values and core objects that are required for the *GraalVM* language runtime. The image heap corresponds to an object graph of all reachable objects determined by the points-to analysis. *Native Image* writes the image heap into the data section of the generated image.

In addition to application objects, the image heap contains class *hubs* — class metadata required by the *GraalVM* language runtime, with each hub containing information about a single type. *GraalVM* class hubs correspond to the `Class<?>` type present in *Java*. Every object has a pointer to its own class hub, from which type metadata can be accessed.

The image heap is divided into several sections, including *read-only* and *writable* sections. Read-only image heap contains objects that were identified as read-only by the points-to analysis. All other objects are placed in the writable section of the image heap. This segregation of image-heap objects allows the garbage collector to only scan a contiguous portion of the image heap. Apart from dividing the heap into sections and research put into object inlining [62], *Native Image* does not perform any layout optimizations of the image heap.

At image-run time, the image heap is memory mapped into the executing image process by the *GraalVM* language runtime (§3.3). Any modifications to the image heap are private to the executing image process, i.e., the image heap contained in the image is immutable. As a result, each image execution starts from the same snapshotted image-heap state, allowing the same image to be replayed multiple times.

3.3 *GraalVM* language runtime

GraalVM language runtime, also referred to as the *Substrate Virtual Machine* (abbr. *SVM*), is a lightweight *JVM* implementation that is bundled into native images generated by the *Native Image* compiler. The *GraalVM* language runtime is designed for fast initialization and low memory footprint. It reduces much of the standard *JDK* implementation, replacing it with specialized components tailored to the ahead-of-time compilation model and the closed-world assumption.

At image-run time, the main routine initializes and invokes the *GraalVM* language runtime. The *GraalVM* language runtime first *reserves* memory in the virtual address space of the executing process with the support from the operating system. It then establishes a *copy-on-write* memory mapping[1] between the file-backed image heap contained in the compiled image and the reserved address space, before invoking the core language-runtime routines. Finally, it dynamically *commits* memory for the *runtime heap* as needed, which establishes a backing store for that memory [96]. The runtime heap holds objects allocated at image-run time.

GraalVM language runtime uses relative pointers for object references, instead of the traditional absolute pointers. All references to objects are calculated relative to the *image-heap base* pointer, which points to the beginning of the image heap. This allows the image-heap object references to remain correct at image-run time, regardless of the position of the image heap in the virtual address space of the executing image process [96]. However, relative references

[1] Copy-on-write memory mapping, also called *implicit sharing* or *shadowing*, is a resource-management technique that efficiently manages shared data across multiple programs. A private copy of the modified pages is made for each process that modified the data [57], while all non-modified pages remain shared. In contrast, *shared* mappings propagate modifications to the data across all mappings.

require additional arithmetic for all memory accesses, like loading a field from an object; the heap base must be added to the reference before the memory access takes place. To make this as fast as possible, the heap base value is always available in a fixed register (e.g., `r14` on *x64* architectures), allowing the address computation to be folded into memory access instructions in most cases.

With all object references being relative to the image-heap base, the image heap can be memory-mapped multiple times in the address space. This allows replicating the image heap without copying for creation of fast and lightweight instances of the *GraalVM* language runtime called *memory isolates*, and copy-on-write sharing of the image heap for reduced memory footprint. Figure 3.3 shows the memory layout of an executing image, with multiple language-runtime instances (i.e., memory isolates) sharing the image heap.

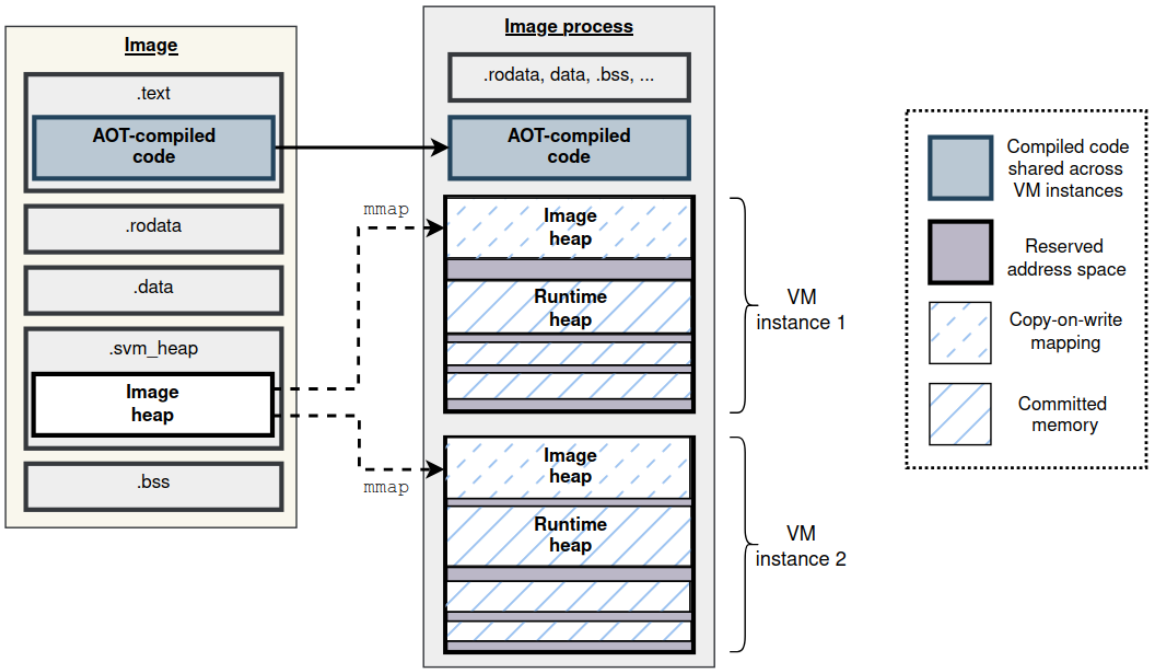


Figure 3.3: Native image sections and process-memory layout of an executing image.

Memory management

Objects allocated by the *Java* code executed by the *GraalVM* language runtime are placed in the runtime heap. The runtime heap is managed by the garbage collector. *GraalVM* provides a generational *Serial* garbage collector (*GraalVM Serial* GC) implementation, with support for other garbage collectors linked as shared libraries, e.g., the *garbage-first* (abbr. *G1*) GC [97]. *GraalVM Serial* GC performs garbage collections in *safepoints*, where the language runtime pauses all executing threads until the garbage collector exits the safepoint. A safepoint is commonly referred to as *stop-the-world*.

GraalVM Serial GC uses a *chunked* object laying-out strategy for the runtime heap, which lays out objects in *heap chunks*. *Aligned* chunks hold many smaller objects, always start at aligned memory addresses, and have a fixed size (`512KB`). These requirements make determining which chunk an object belongs to very efficient, through simple bitwise shifts. *Unaligned* chunks contain a single object, without any size restrictions.

Heap chunks are grouped into *spaces*, which make up the *young* and *old* heap generations. Each space contains a linked list of aligned and unaligned heap chunks. The young generation consists of a single *eden space* for newly created objects and multiple *survivor* spaces for objects that survive garbage collection cycles before getting promoted to the old generation. The old generation consists of a single space.

GraalVM Serial GC keeps track of references between generations through remembered sets backed by card tables, triggered by write barriers [181]. The relevant metadata, e.g., the card table, is contained in the header of each heap chunk. Similarly, *GraalVM Serial* GC tracks references from the image heap to objects in the runtime heap. The image heap is "immortal", i.e., objects in the image heap are never collected, but image-heap objects can reference objects in the runtime heap. On the other hand, every object in the runtime heap points to the image heap, at least to its hub that resides in the image heap.

Memory isolates

GraalVM memory isolates, or simply *isolates*, are disjoint heaps that allow multiple tasks in the same *GraalVM* language-runtime process to operate independently of one another. All isolates share AOT-compiled code located in the text section of the compiled image. Additionally, because of the relative pointer mechanism employed by the *GraalVM* language runtime, each isolate can also share the image heap using copy-on-write memory mappings.

GraalVM isolates provide a novel kind of optimization sometimes called the *language-level* virtualization, where multiple language-runtime instances run in the same process or thread, sharing code and pre-initialized state. Each isolate is managed separately (including garbage collections), compared to traditional scenarios where a single task can slow down other tasks that share the same heap.

GraalVM isolates allow other optimizations such as *compressed references*, where reference sizes are reduced to four bytes instead of eight, to further reduce the memory footprint. Runtime heaps in each isolate are likely small and can fit into an address range addressable by 4-byte references. Furthermore, most data structures use more space for pointers than for primitive data, so compressed references can greatly reduce the memory footprint of the application.

Due to their efficiency and sharing capabilities, *GraalVM* isolates are becoming popular in recent years as a driver for architectures that implement disposable sandboxes that start fast and require minimal amounts of resources [270, 168]. Since *GraalVM* supports executing multiple languages in a single language runtime (§3.4), *GraalVM* isolates are ideal for cloud-native architectures in *FaaS* and serverless environments (§3.5).

3.4 *Truffle* language implementation framework

Truffle framework provides libraries and tools for building programming-language implementations as interpreters [156] for the self-modifying abstract syntax trees [394]. Internally, *Truffle* uses the *Graal* compiler to *partially evaluate* the language interpreters into optimized machine code [389].

Truffle language implementation framework is bundled with the *Graal* compiler, and includes several language implementations as part of *GraalVM*, including *JavaScript* [125], *Python* [374], and *WebAssembly* [149]. *GraalVM* also provides *Sulong* [306], an *LLVM* [215] interpreter on top of *Truffle*, allowing programs written in *LLVM*-based languages to run on *GraalVM*.

Truffle provides execution *engines* for a set of *Truffle* language implementations. Code execution is done in *Truffle* language *contexts*, which can evaluate expressions and snippets of code. Language contexts can be pre-initialized by the *GraalVM Native Image* compiler and stored in the image heap, for faster startup of the first language context. *Truffle* can also execute code in sandboxes leveraging *GraalVM* memory isolates.

Truffle features full *polyglot* capabilities, where a program written in any *Truffle* language can invoke and exchange objects with programs written in any other *Truffle* language. Figure 3.4 shows an example of executing a *C* program (3.4a) compiled to *LLVM* [215] bitcode, invoked from *Java* (3.4b) using *GraalVM*, and running in sandboxed mode (3.4c). Language-interoperability and sandboxing that *Truffle* adds to *GraalVM* enable a unified language runtime for popular programming languages, with high performance and low memory footprint, making it ideal for cloud environments, especially serverless computing (§3.5).

(a) C source code (compiled to `src.bc`)

```c
#include <stdio.h>

int main(int argc, char *argv[]) {

    // BUG: %n format specifier expects a pointer
    printf("argc: %n\n", argc);

    return 0;
}
```

(b) Sandboxed *LLVM* bitcode execution on *GraalVM*

```java
// Load sources from the file
var src = Source.newBuilder("llvm", new File("src.bc")).build();

// Build a sandboxed LLVM context that allows IO
var builder = Context.newBuilder()
    .allowIO(true)
    .option("llvm.sandboxed", "true");

// Create and execute the context on the source
try (Context polyglot = builder.build()) {
    polyglot.eval(src).execute();
} catch (Exception ex) {
    System.out.println(ex);
}
```

(c) Resulting exception

```
argc: org.graalvm.polyglot.PolyglotException:
Illegal pointer access: 0x0000000000000001
    ...     // Exception details
```

Figure 3.4: Source code of a buggy *C* program (a) compiled to *LLVM* bitcode and executed on *GraalVM* (b) with full sandboxing (c).

3.5 *GraalOS* serverless platform

GraalOS [270] is a high performance serverless application deployment platform running on the *Oracle Cloud Infrastructure* [273]. Designing and deploying applications on *GraalOS* is done using the *Graal* development kit for *Micronaut* [242], a JVM-based framework for building modular microservice and serverless applications, that fully integrates with *GraalVM Native Image*.

GraalOS fully embraces the *language-level* virtualization technique proposed by *GraalVM* memory isolates. *GraalOS* uses *GraalVM Native Image* to run applications as native executables, combined with the latest advances in processor architectures for hardware-enforced application isolation without containerization. It hardens *GraalVM* memory isolates with hardware memory-protection keys (abbr. *MPK*) [237], further enforcing isolation and improving security. Additionally, it wraps system calls with utilities such as *seccomp* [66] to further reduce the attack surface.

In addition to using *GraalVM*, *GraalOS* also leverages *Truffle* language implementation framework to execute functions written in languages other than *Java*, including *Python* and *JavaScript*, and allows for polyglot interoperability between these languages. These properties make *GraalOS* ideal for designing and deploying cloud-native serverless applications that start fast, are highly responsive, and require significantly less memory to run compared to traditional solutions. *GraalOS* development is still ongoing, with potential for more optimizations such as better runtime-heap management and sharing of runtime-only data.

Chapter 4

Related Checkpoint/Restore (C/R) techniques

Checkpoint/Restore (abbr. *C/R*) techniques freeze the execution state of a program and persist it in a continuable form. Freezing and persisting the execution state is referred to as *checkpointing* or *snapshotting*, whereas resuming execution from such a state is referred to as *restoring* or *loading*. Although the term *snapshot* is most commonly used to represent a persisted execution state, the term *image* is often used to denote snapshots made by C/R techniques that persist blueprints or clones of the entire system [72, 206].

Modern cloud-computing architectures incorporate C/R to reduce the overhead of spawning virtualization stacks and warming up services [45, 235, 98, 239]. Figure 4.1 compares different C/R techniques in terms of their scope, portability, and resulting snapshot sizes. Depending on their scope, C/R techniques can be applied to the entire system, executing processes or their parts, and individual objects in the language-runtime process. System C/R (§4.1) produces system images of virtualized execution environments that are the largest in terms of size on disk. Process C/R (§4.2) freezes and restores processes, including process-specific state such as open files, started threads, and established memory mappings. Process C/R can also be customized to perform specialized C/R of language runtimes or their heaps. Object serialization and deserialization (S/D) (§4.3) serializes objects in a common and portable format, resulting in smallest snapshot sizes.

4.1 System C/R

C/R can be applied on the virtualization stack or its components to create a *system snapshot*, also called a *system image*. System snapshots include the complete state of the virtualization stack, including the operating system state and the virtualized hardware state. Depending on the virtualization technology in place, a system snapshot can represent a suspended virtual machine [130, 108, 253, 407] or a paused container [398, 131, 377], among others.

System snapshots are often split into multiple files, which improves their portability and usability. For example, virtual-machine snapshots include the state of the operating memory and the virtualized hard disk, while additional metadata files track relationships between them [60, 129, 82]. This enables large snapshots to be decomposed into reusable components, as well as saving and transferring individual components (e.g., a hard disk state) from one virtual machine to another, without having to process other snapshot components.

Modern general-purpose virtualization platforms use various file formats for different system-

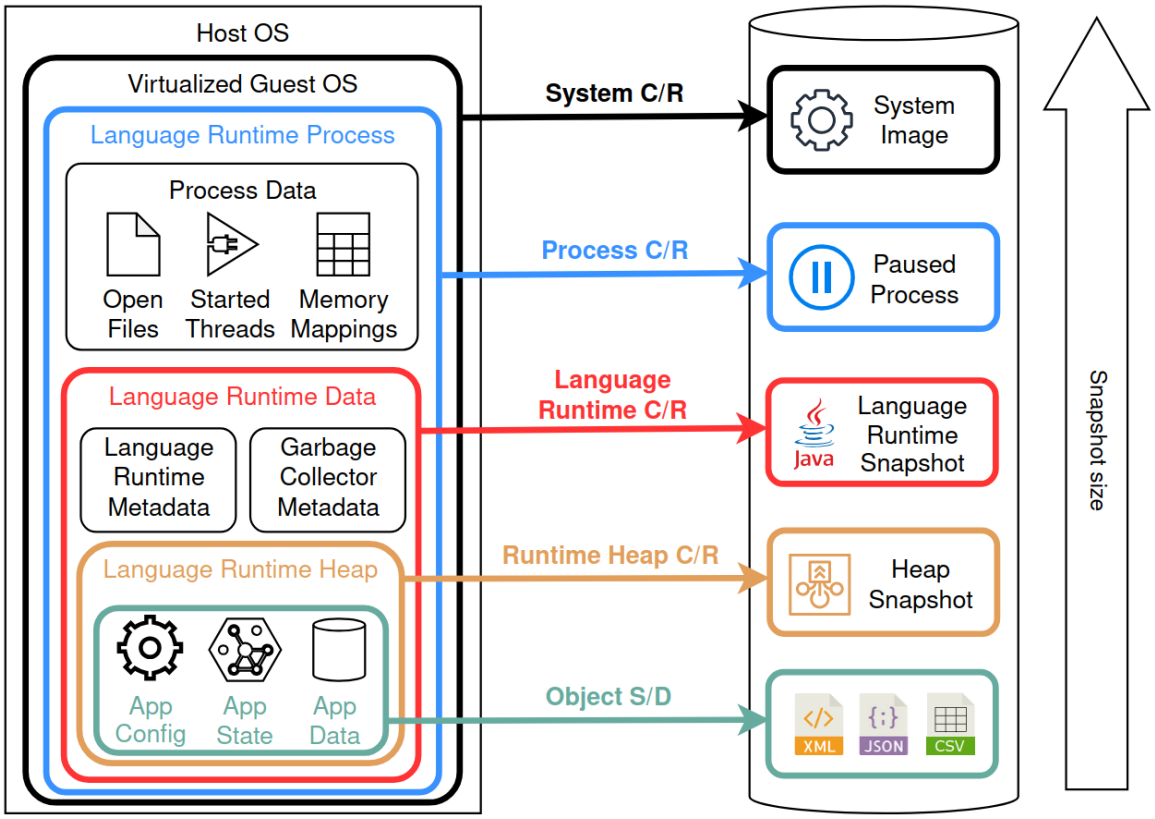


Figure 4.1: Comparison of Checkpoint/Restore techniques in terms of their scope, portability and resulting snapshot sizes.

snapshot components. Operating-memory formats range from open binary formats to proprietary formats such as the *VMEM* format used by VMware [129, 60]. Virtual machine disk state formats include the widely adopted *Virtual Machine Disk* format (abbr. *VMDK*), the *VDI* format used by *Oracle VirtualBox* [82], *Virtual Hard Disk* formats (abbr. *VHD/VHDX*) for the *Microsoft Hyper-V* hypervisor [354], the *HDD* format used by *Parallels* [279] for *Mac* and *Windows* virtualization, as well as *Qcow* and *Qed* disk formats used by *QEMU* [336].

Similar to virtual-machine snapshots, popular container-management platforms such as *Docker* [238, 288] or *Podman* [153] provide container-snapshotting capabilities. With *Docker*, it is possible to *commit* changes made to a running container as a new image, or to export an archive of the container filesystem [3]. *Docker* and *Podman* both support container C/R, leveraging process C/R techniques such as *CRIU* [23] to snapshot the state of an executing container process.

State-of-the-art research in cloud-computing environments introduces innovative modifications to the virtualization stack with the goal of better computing-model support [155, 251, 131, 27]. System C/R techniques are used on top of efficient architectures to provide fast service startup and low memory footprint, which are crucial resources for efficient serverless computing [371, 102, 322]. For this reason, serverless has become an increasingly popular target for optimizations that, to some extent, involve system C/R [106, 333, 56, 178, 42, 27, 371].

Xen [43, 396] is one of the most popular and performant free and open-source type-1 hypervisors. It provides support for creation, memory management, and CPU scheduling of virtualized *domains*. This allows multiple guest operating systems to execute simultaneously on the same hardware, in separate domains, with one special privileged domain having access

to the hardware [73]. *Xen* uses multiple approaches to running the guest operating system inside a domain, including paravirtualization, hardware virtual machines, paravirtualization with hardware virtualization, and paravirtualization in a hardware virtual machine container [379]. *Xen* provides snapshotting capabilities in the form of virtual machine snapshots. Each virtual machine snapshot represents a record of a running virtual machine at a point in time. The snapshot contains both metadata, which includes configuration, and storage information, which includes the disk state. During snapshotting, all I/O operations are temporarily halted to ensure a self-consistent disk snapshot. Virtual machine snapshots can be converted into templates for fast virtual machine creation. Templates can be orchestrated in a way that the root filesystem is shared across templates [316], and compressed to eliminate redundant memory data in the snapshot [103].

Linux *Kernel-based Virtual Machine* (abbr. *KVM*) [195] is an open-source hypervisor for Linux, included in the mainline Linux kernel. It consists of a loadable kernel module (`kvm.ko`) that provides the core virtualization infrastructure and a processor-specific module for Intel (`kvm-intel.ko`) and AMD (`kvm-amd.ko`) processors. With *KVM*, it is possible to execute multiple virtual machines running unmodified Linux or Windows images, with each virtual machine having its own virtualized hardware. *KVM* supports virtual-machine snapshotting of both executing and stopped virtual machines. Virtual machine snapshots can be *internal* or *external*. Internal snapshots are stored in the virtual disk image (often in the `Qcow2` format) of the guest virtual machine, with snapshot properties serialized into an XML file. This means that multiple snapshots of the same virtual machine will all be stored in the same image file. On the other hand, external snapshots use copy-on-write mappings to overlay a new disk image of the guest operating system.

QEMU [48], short for *Quick Emulator*, is a free and open-source virtualization software that emulates a computer's processor by translating the emulated binary code to an equivalent binary format that is executed by the machine. *QEMU* can perform full operating-system emulation for any machine for various supported architectures. It can also run *KVM* and *Xen* virtual machines with near-native performance. *QEMU* snapshots represent images that refer to the original image using *Redirect-on-Write* [397]. In this way, taking a snapshot does not change the original image. Similarly to *KVM*, *QEMU* supports creation of internal and external snapshots, as well as temporary snapshots.

Firecracker [2] is an open-source virtual machine monitor (abbr. *VMM*) that uses *KVM* to create and manage lightweight virtual machines called *microVMs*. Each microVM isolates the executing process using *cgroups* [323] and *seccomp BPF* [66], with access to a tightly controlled subset of system calls. MicroVMs can thus be launched in non-virtualized environments, taking advantage of the isolation and security of the traditional VMs, and the resource efficiency of containers. *Firecracker* supports microVM snapshotting [216], where a running microVM and its resources are serialized and stored on an external medium. Snapshots include the guest memory and the state of the hardware emulated by *KVM* and *Firecracker*. The microVM snapshot is separated into the guest memory file, the microVM state file, and zero or more disk files, depending on the disk usage. With this design, *Firecracker* snapshots support sharing of memory pages and disks between multiple microVMs. During snapshot loading, *Firecracker* establishes a copy-on-write mapping of the memory snapshot, resulting in on-demand lazy-loading of memory pages, and private modifications for every microVM that loads the snapshot.

D-CRES [253] provides a parallel and consistent live checkpoint mechanism for large-memory virtual machines [10]. *D-CRES* uses *QEMU* and *KVM*, with support for split migration [350] and remote paging. Normally, after split-migration, a large-memory virtual machine is converted into a *split-memory virtual machine*, running across multiple hosts. *D-CRES* ad-

dresses host and network failure problems associated with the migration of split-memory virtual machines, and minimizes downtime during checkpointing by deferring the write-back of a large amount of memory data to checkpoint files on disk. *D-CRES* enables efficient C/R of a split-memory virtual machine by saving its memory on every host independently, to avoid remote paging between hosts. Additionally, the checkpointing process can be performed without stopping the virtual machine. Instead of traditional solutions that restore split virtual machines on a single host, *D-CRES* allows a split-memory virtual machine to be restored across multiple hosts in parallel. *D-CRES* uses different formats for checkpoint files during C/R, converting the checkpoint files offline in order to maximize the performance.

Berkeley Lab Checkpoint/Restart (abbr. *BLCR*) [150] is a system-level checkpointer primarily designed for high-performance computing (abbr. *HPC*) environments. It consists of a loadable kernel module and a small library that has to be preloaded. *BLCR* focuses on preemptiveness and transparency, not requiring changes to application source code. In addition to *HPC* environments, *BLCR* covers a wide range of applications and systems, including MPI [352] systems.

libhashckpt [122] performs hybrid incremental checkpointing leveraging page protection and GPU-powered hashing. In addition to the operating system page protection mechanism, *libhashckpt* uses hashing and MPI hooks [352] to determine the exact address ranges that should be included in the incremental snapshot. *libhashckpt* offloads hash computations to the GPU in order to reduce the overhead of hash calculation and make checkpointing viable for high-performance computing (abbr. *HPC*) environments.

4.2 Process C/R

Process C/R techniques capture the executing process state, including open file descriptors, started threads, pipe parameters, and memory mappings. Snapshotting processes facilitates program debugging on different systems by creating and analyzing process snapshots at points of failure. Long-running jobs can be persisted periodically for improved fault tolerance and recovery in case of failures. Unexpected system failures or broken updates can be remedied by falling back to an earlier version of the program persisted as a process snapshot.

Process C/R techniques can be applied to execution environments that host applications, such as containers or language runtimes. Container snapshotting capabilities of *Docker* [238] and *Podman* [153] rely on process C/R techniques to capture the state of the executing container process. When applied to a language-runtime process, process C/R techniques generate snapshots that include language-runtime state, such as metadata or runtime heap. Snapshotting the warmed-up state of the language-runtime process avoids costly language-runtime initialization and results in faster application startup times [168, 178].

Normally, process C/R techniques capture the state of the entire process into a *whole-process snapshot*. However, process C/R techniques can be tailored to capture specific processes such as executing language runtimes, which often require special handling of language-runtime metadata and heap.

Whole-process checkpointing

Whole-process snapshots capture the entire process, including its memory state, open files, and started threads. Because they capture the entire process state, whole-process snapshots can be created without any additional support from the process itself. This makes whole-process

Listing (4.1) Contents of a *CRIU* process snapshot.

```
$ ls /tmp/criu-snapshot-pid-25341
core-25341.img  ids-25341.img      pages-1.img  timens-0.img
fdinfo-2.img    inventory.img      pstree.img   tty-info.img
files.img       mm-25341.img       seccomp.img
fs-25341.img    pagemap-25341.img  stats-dump
```

snapshotting techniques very popular for a wide range of use-cases and applications [85, 377, 398].

However, the downside of whole-process snapshotting is that the process is the process is not aware of checkpointing or restoration events. This makes coordinated checkpointing and restoring difficult, as it requires the checkpoint process to know when to trigger a checkpoint. Furthermore, the resulting snapshot can, in most cases, only be restored once at a time, 1due to the captured open files and sockets that are not easy to clone.

In recent years, *Checkpoint/Restore In Userspace* (abbr. *CRIU*) [377, 86] became a popular solution for whole-process checkpointing. It can be used to execute container processes to freeze the entire container state or individual application processes. Use-cases for *CRIU* include container or process migration [3, 183, 398, 291, 286, 363], databases [217, 410, 277], system security [131, 411, 348, 412] and updates [191], cloud-computing optimizations [75, 76, 334, 124, 202], and GPU systems [349, 402, 110, 109].

CRIU uses the `PTRACE_SEIZE` command of the *ptrace*[1] Linux kernel interface or Linux Control Group (abbr. *cgroup*) [172] to freeze the executing process. It then injects auxiliary (parasitic) code through the *ptrace* interface that collects information from inside the address space of the executing process, such as credentials and memory contexts. Finally, *CRIU* restores the original code by removing the injected parasitic code, before detaching from the process. The resulting snapshot is persisted to disk, broken down into several files, shown in Listing 4.1.

Restoring snapshots requires *CRIU* to be morphed into the tasks it restores. *CRIU* first reads snapshot files and resolves resources shared between processes, such as shared files or shared memory areas. It then calls `fork()` as many times as needed to restore the state of the process tree at checkpoint time. Afterwards, it restores basic task resources, i.e., opens files, prepares namespaces, maps private memory areas, creates sockets, and more. Finally, it restores remaining resources such as the exact memory mapping locations, timers, credentials, and threads.

Distributed MultiThreaded CheckPointing (abbr. *DMTCP*) [25] is a checkpointing package for distributed applications that implements C/R of a process on a library level. *DMTCP* intercepts library calls from the application, builds an information database about the process's internals, and forwards the library call to the recipient. The gathered information is used to build a snapshot of the executing process. Compared to *CRIU*, *DMTCP* requires its library to be dynamically linked in the executing process. Additionally, intercepting library calls adds a performance overhead, which is not present with *CRIU*.

Pin-Long Instruction Trace (abbr. *PinLIT*) [284] is a checkpointing tool built on top of Intel's proprietary *PIN* binary instrumentation tool [302]. It records the CPU register state and memory pages that contain application and library code, and optimizes snapshot sizes by storing

[1]From `ptrace(2)` *man* page: The `ptrace(2)` system call provides a means by which one process (the *tracer*) may observe and control the execution of another process (the *tracee*), and examine and change the tracee's memory and registers. It's primarily used to implement breakpoint debugging and system call tracing. `ptrace` operations are referred to as *commands*.

memory used during a specified interval. *Program Record/Replay Toolkit* (abbr. *PinPlay*) [281], a successor of *PinLIT*, logs execution of a program in a set of files called a *pinball* and then replays that run off of a *pinball*, repeating the captured execution.

Language-runtime checkpointing

Compared to whole-process checkpointing techniques that capture the entire process state, language-runtime checkpointing techniques specifically target processes of executing language runtimes. Prime candidates for such checkpointing techniques are language runtimes with high memory footprint, such as the *Java Virtual Machine* (abbr. *JVM*) [225], *V8 JavaScript engine* [372], or *Common Language Runtime* (abbr. *CLR*) environment for the *.NET* platform [58], among others.

Popular approaches to checkpointing language runtimes tailor whole-process checkpointing solutions such as CRIU [86] to better fit the language-runtime specifics. *OpenJDK's* [267] *Coordinated Restore at Checkpoint* (abbr. *CRaC*) [269] project and *OpenLiberty InstantOn* [265] provide JVM checkpoints. *CRaC* aims to develop a standard API that notifies Java programs about C/R events and introduces support for Java instance checkpointing. *InstantOn* performs checkpointing of containerized Java processes through CRIU hooks that allow the *OpenLiberty* runtime to safely persist the process in a container image.

Fireworks [331] introduces VM-level post-JIT snapshotting capabilities, where the executing language runtime coordinates with the execution environment. Resulting snapshots contain JIT-compiled code, eliminating warmup times associated with runtime compilations. *Fireworks* is implemented on top of *Firecracker* [2] *microVMs*, maintaining a high level of isolation and low memory footprint, making *Fireworks* ideal for serverless function sandboxing [331].

When applied to the *JVM*, C/R mechanisms can improve efficiency in cases where long initialization dominates the execution time [407, 175]. However, *JVM* snapshots contain the state of the runtime heap at the time when the snapshot was taken. Thus, the heap snapshot contains dead objects that increase the heap fragmentation and overall memory footprint of the resulting heap snapshot. Performing garbage collections prior to snapshotting helps avoid capturing irrelevant runtime heap sections and reducing heap-memory fragmentation [154, 193], but excessive garbage collections of the entire heap add significant overhead to snapshotting times.

Several solutions have been proposed to remedy the overhead of garbage collections prior to snapshotting and reduce snapshot sizes. *ALMA* [61] uses a migration-aware GC policy that uses heap liveness information and estimated collection costs. *Checkpoint ROllbaCk via lightweight HEap Traversal* (abbr. *Crochet*) [47] uses bytecode rewriting, debug APIs, and proxy objects to perform a lazy heap traversal that copies or restores object states during object accesses. *Ditto* [332] extracts only the necessary information to restore the application. Other solutions use dynamic pruning of unreachable data to create smaller snapshots that further reduce the overall memory footprint [335].

Runtime-heap snapshotting

State-of-the-art language runtimes use snapshotting techniques to precompute and share compiled code and data across multiple processes. The resulting snapshots are usually loaded early during language-runtime startup, avoiding costly initialization. The pre-compiled code allows the application to start hot, eliminating warmup times and achieving peak performance faster.

Oracle GraalVM [271] (explained in detail in (§3)) achieves closed-world AOT compilation of Java applications by performing points-to analysis that identifies reachable code, in combination with class pre-initialization during the build process [387]. It generates a *native image* containing the pre-initialized heap in the data section and pre-compiled code in the text section. *GraalVM* can share image sections across *GraalVM isolates*, isolated instances of the *GraalVM* execution environment.

Similarly to *GraalVM*, the *V8 JavaScript* engine [372] also supports isolated execution [123, 408]. The *V8* engine can pre-initialize execution contexts [137] by creating a heap snapshot that is loaded into isolates during context initialization. Because *V8* isolates start fast and use less memory compared to multiple *V8* processes, *Cloudflare Workers Platform* [80] uses *V8* isolates to execute *JavaScript* serverless functions.

Other language runtimes also provide snapshotting capabilities. *Dart* [59] virtual machine uses different snapshot types [138] depending on the execution strategy [325]. Similarly, *Smalltalk* [161] and *Self* [369] runtimes store both code and data in heap snapshots.

4.3 Object serialization and deserialization (S/D)

Object *serialization* is an act of converting an object into a sequence of bytes, for the purpose of saving the state of the object to persistent storage, or transmitting objects over a network; while object *deserialization* is an act of converting a sequence of bytes that encodes an object back into the actual object it represents [343]. Together, these are referred to as object S/D, as a specialized form of C/R applied to individual objects. Object S/D is widely used for configuration files, data persistence, network communications, such as those in the RESTful Web APIs [304], where data is exchanged between the client and the server or between different services, etc.

When using common serialization formats, high portability often results in data duplication or information loss. For example, JSON [182] does not store type information (data loss), nor does it have a way of referencing object fields, thus every reference to the same object results in that object being re-encoded into its JSON representation (data duplication). In general, transferring objects across incompatible language runtimes requires some sort of data abstraction and loss. Thus, interchangeable plain-text formats are often used for network communication across services that execute in unknown, often different, execution environments. Binary formats are often used to serialize and deserialize data in known execution environments. Object S/D can also be aided using object *schemas*, often defined using an interface-description language [341].

Object S/D introduces additional CPU overhead due to data transformation, and can be a performance bottleneck for applications that operate on or exchange large amounts of data, e.g., in the context of big-data analytics [174, 135, 31, 68]. Solutions for optimizing object S/D include hardware-based accelerators and language runtime-integrated mechanisms.

Plain-text object S/D

Plain-text object S/D formats include raw text files, *JavaScript Object Notation* (abbr. *JSON*) [362], *eXtensible Markup Language* (abbr. *XML*) [119], YAML [399], CSV [329], *Open Data Description Language* (abbr. *ODD*) [264], *Extensible Data Notation* (abbr. *EDN*) [118], and others [384]. Table 4.1 shows an example of an `Employee` object serialized using the most common plain-text formats.

JSON [362]	*YAML* [399]

`{`
`  "name": "John Doe",`
`  "age": 30,`
`  "coeff": 3.3`
`}`
	`employee:`
	`  name: John Doe`
	`  age: 30`
	`  coeff: 3.3`

XML [119]	*CSV* [329]

`<employee>`
`  <name>John Doe</name>`
`  <age>30</age>`
`  <coeff>3.3</coeff>`
`</employee>`
	`name,age,coeff`
	`John Doe,30,3.3`

Table 4.1: Sample `Employee` object serialized into *JSON*, *YAML*, *XML* and *CSV*.

Due to the popularity and interchangeability of the *JSON* format, *JSON* object S/D frameworks provide efficient implementations and allow users to provide custom data binding implementations and serializers in order to optimize S/D speeds [257]. For example, most popular *Java JSON* S/D libraries such as *fastjson* [7], *dsl-json* [257], *Jackson* [120], and *gson* [146] reach over one million S/D operations per second for plain Java objects [303, 340].

Binary object S/D

Many binary object S/D formats exist, prioritizing different features depending on the particular use-case [351, 378]. Most binary S/D formats aim to compact or compress data, sacrificing human-readability for snapshot size. As such, they are convenient for long-term persisted storage or high-frequency network communication.

Apache Parquet [280] is a high-performance file format designed for efficient storage and retrieval of column-oriented data. *Bencode* [49] is a binary encoding format that is part of the *BitTorrent* [54] specification. *BSON* [63] is a binary extension to *JSON* used in popular database management systems such as *MongoDB* [252]. *MessagePack* [240] and *UBJSON* [370] provide a more compact binary representation similar to *JSON* but requiring fewer bytes to encode data. *Structured Data Exchange Format* (abbr. *SDXF*) [386] is a standardized format for serialization and exchange of structured data. *Canonical S-expressions* are a binary encoding of *S-expressions* [200] popularized by the *LISP* programming language.

Kryo [116] is a binary object S/D framework for *Java* that uses a special compact binary format to enable serialization and communication across *JVM* instances. *Kryo* achieves high S/D performance and compact snapshot sizes [340], making it ideal for high-performance, network-intensive, and big-data frameworks such as *Apache Spark* [404, 30]. *Kryo* also supports automatic deep and shallow copying or cloning, with shallow copying being the default.

Schema-based object S/D

Schema-based S/D frameworks generate high-performance serializers based on a provided object schema. An object schema is defined in a separate file using a structured format. Table 4.2 shows examples of schema definitions for an `Employee` object using *Protocol buffers* [144] (left) and *Apache Avro* [359] (right). Schema-based S/D is widely used in implementing high-performance remote procedure calls (abbr. *RPC*) [53].

Protobuf [144]	*Apache Avro* [359]

`syntax = "proto3";`
 	
`message Employee {`
`  string name = 1;`
`  int32 age = 2;`
`  float coeff = 3;`
`}`
	`{`
	`  "type" : "record",=`
	`  "name" : "Employee",`
	`  "fields" : [`
	`  { "name" : "Name" , "type" : "string" },`
	`  { "name" : "Age" , "type" : "int" }`
	`  { "name" : "Coeff" , "type" : "float" }`
	`  ]`
	`}`

Table 4.2: Sample `Employee` object schema defined using *Protobuf* (left) and *Apache Avro* (right) schema formats.

Protocol Buffers (abbr. *Protobuf*) [144] are an extensible, language-neutral, and platform-neutral mechanism for serializing structured data. *Protobuf* object schema is defined in *proto* files and compiled using the *protoc* schema compiler to a high-performance serializer implementation in a variety of supported programming languages. Figure 4.3 shows *Java* serialization and *C++* deserialization code for the `Employee` *proto* schema shown in Table 4.2 (left). Protocol buffers are used in high-performance *RPC* frameworks such as *gRPC* [148] and are accelerated further by programmable data transformation and hardware accelerators [188, 290].

(a) Serializer (*Java*)

```
1 var john = Employee.newBuilder()
2     .setName("John Doe")
3     .setAge(31)
4     .setCoeff(3.3)
5     .build();
6 john.writeTo(output);
```

(b) Deserializer (*C++*)

```
1 Employee john;
2 john.ParseFromIstream(&input);
3
4 name = john.name();
5 age = john.age();
6 coeff = john.coeff();
```

Figure 4.3: Example *Protobuf* serializer written in *Java* (left) and deserializer written in *C++* (right) using the `Employee` *proto* schema shown in Table 4.2.

The Apache Software Foundation develops several schema-based S/D frameworks that provide additional features compared to existing solutions. *Apache Thrift* [361] adds support for more programming languages compared to *Protobuf*, but does not provide a versatile extension system. *Apache Avro* [359] provides dynamic typing and support for untagged data, resulting in smaller snapshot sizes. *Avro* does not require code generation to process data files or implement RPC protocols, although code generation is an optional optimization for statically typed languages. *Apache Fory* [360] leverages JIT compilation and zero-copy to implement multiple binary serialization protocols. *Apache Arrow* [358] defines a language-independent columnar memory format for flat and nested data that supports zero-copy reads for efficient data access without serialization overhead.

Solutions such as *FlatBuffers* [139] and *Cap'n Proto* [67] remove encoding and decoding steps from the S/D pipeline. *FlatBuffers* are used in performance-critical applications and game development, with emphasis on maximum memory efficiency. *Cap'n Proto* uses interface schemas similar to the *C* programming language, with support for object-oriented features such as multiple inheritance. *Cap'n Proto* also employs *time travelling*, where it folds a chain of subsequent and mutually dependent *RPC* calls into a single *RPC* call.

Hardware-accelerated object S/D

Object S/D can also be accelerated via techniques that involve specialized hardware. Popular approaches involve parallelization or splitting of S/D computations and offloading the most resource-intensive phases to hardware. Furthermore, software messaging frameworks help eliminate errors and engineering effort by generating specialized hardware that accelerates object S/D [409].

Parallelization of S/D phases is performed using memory-level parallelism, or specialized data-transformation accelerators. *Cereal* [174] processes references and object values in parallel using memory-level parallelism, in combination with a packing scheme for S/D metadata. Similarly, *Optimus Prime* [290] parallelizes S/D computations using hardware-backed data transformation accelerators to improve S/D throughput and latency.

State-of-the-art object S/D optimizations offload S/D computations to various hardware. *Naos* [356] and *RMMap* [231] use *Remote Direct Memory Access* (abbr. *RDMA*) [301] to completely eliminate steps in the S/D pipeline[2]. *Morpheus* [367] offloads S/D to specialized solid state drives. *HODS* [220] leverages the FPGA module inside *NVMe* solid state drives to optimize deserialization throughput. Hardware accelerators for *Protocol Buffers* [144] have also been proposed, leveraging super-scalar *RISC-V* cores [188].

Runtime-integrated object S/D

Language runtime-integrated S/D solutions involve specialized S/D formats tailored to the executing language runtime, or modifications to the language runtime itself. *Java* [55] and *PHP* [230] serialization formats leverage specifics of the *Java* and *PHP* language runtimes such as the *Java HotSpot VM* [276] or *PHP Zend* engine [406].

Other approaches include functional programming concepts such as *Pickler combinators* [194], that provide S/D capabilities for runtime objects. Object-oriented pickler combinators like those for the *Scala* programming language solve problems associated with traditional pickler combinators, such as open class hierarchies and type polymorphism [249]. Pickler combinators are also used in *Python* [283] for serializing objects into byte streams.

JVM-specific S/D optimizations address S/D overhead in big-data environments. *Skyway* [258] directly sends object graphs between remote JVM instances, removing most of the serialization overhead. It relativizes pointers stored in the objects during serialization, and then performs a linear scan during deserialization, where the absolute values are calculated for all the relative pointers. Depending on the number of references and the size of the object, linear scans can introduce a considerable overhead.

Zero-Change Object Transmission (abbr. *ZCOT*) [391] introduces an architecture that supports object transfer among distributed *JVM* instances without object transformation, elim-

[2] *Naos* can also work without RDMA networks, i.e., over plain TCP, although that incurs a performance penalty. *Naos* works best with *RDMA* networks in asynchronous operations modes [356].

inating the object S/D phase entirely. *ZCOT* architecture uses the *transfer space* — a distributed shared-memory (DSM) abstraction established between remote JVM instances, along with a deduplication module to avoid redundant transfers. To manage and coordinate JVM instances that use the transfer space, *ZCOT* employs a metadata server that provides distributed class-data sharing.

Table 4.3 compares the most relevant runtime-integrated C/R techniques with plain-text and binary object S/D. As network communication overhead continues to decrease [356], research increasingly focuses on eliminating S/D, either by using external hardware [356, 231] or specializing for distributed environments [258, 391].

Table 4.3: Comparison between most relevant state-of-the-art C/R solutions.

Solution	JSON S/D	java [272] Kryo [116]	Skyway [258]	Naos [356] RMMap [231]	ZCOT [391]
S/D format	JSON	Binary	Binary	Binary	Binary
Language-runtime integration	No	No	Yes	Yes	Yes
Reference tracking	No	Yes	Yes	Yes	Yes
Object transformation	Yes	Yes	Partial	No	No
External management	None	None	None	RDMA networks	Metadata Server
Data loss	Yes	No	No	No	No
Data sharing	No	No	No	No	Partial
RSS improvements	Low	Low	Low	Low	Only for big-data

Chapter 5

Direct Object Snapshotting and Sharing system (DOSS)

Direct Object Snapshotting and Sharing (DOSS) is a data-centric C/R system integrated into the executing language runtime. DOSS provides:

Direct snapshotting of live heap objects into DOSS *object-graph snapshots*, i.e., DOSS snapshots object graphs from the runtime heap, without any transformation or serialization taking place. Since DOSS snapshots do not contain serialized objects, DOSS snapshots are loaded into the executing application in constant time complexity using memory mappings, without any deserialization overhead.

Snapshot sharing across language-runtime instances via copy-on-write memory mappings. DOSS achieves near-constant data-memory overhead by sharing immutable data-memory pages, with copies of data-memory pages being made only when the data is modified. Additionally, DOSS architecture enables lazy-loading of data pages on access.

Snapshotting during execution of a program, providing a C/R solution for data available only during application execution. DOSS manages snapshots dynamically during application execution without stopping or restarting the application. Managing snapshots involves creating and persisting snapshots, loading persisted snapshots into the application, and unloading snapshots from the application.

High performance and low memory footprint of the system, as direct data snapshotting eliminates unnecessary S/D overhead and allows for fast data loading, while snapshot sharing reduces the overall memory footprint in cases where many applications share the same data. As such, DOSS is ideal for cloud-native architectures, such as *FaaS* or serverless workers, where many instances of the same language runtime share most of the runtime data.

DOSS is primarily designed to extend the capabilities of language runtimes embedded into AOT-compiled native executables, e.g., native images that embed the *GraalVM* language runtime (see §3). Therefore, the terminology and explanations in this chapter use the *GraalVM* language runtime as a reference execution environment. *Image-build* and *image-run* time denote program-compilation time and program-execution time, respectively. *Language-runtime instance* or *VM instance* refers to the embedded language runtime that is executing the code while the application is running. The *image heap* refers to the heap built at image-build time, while the *runtime heap* refers to the heap used for objects created at image-run time.

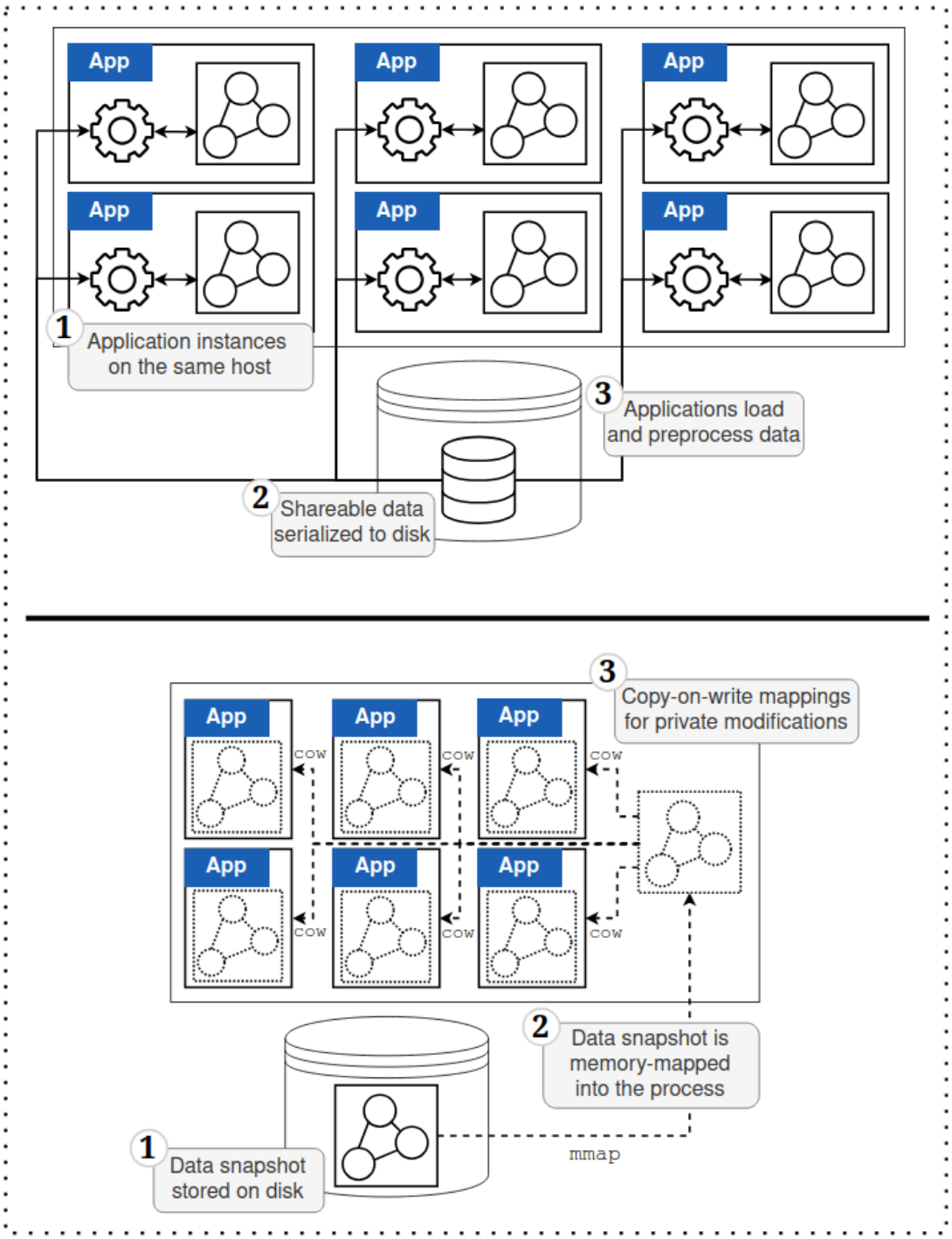


Figure 5.1: Data duplication in traditional service deployments (top), compared to direct object snapshot sharing (bottom).

Figure 5.1 shows how excessive data duplication and pre-processing in traditional deployments (top) can be eliminated using DOSS snapshots (bottom) that leverage copy-on-write memory mappings. DOSS snapshots (§5.1) reside in the *snapshot heap* (§5.2) — an auxiliary heap with shareable heap sections, serving as a medium for DOSS operations (§5.3) during application execution. DOSS supports several object laying-out strategies (§5.4), and integrates with the garbage collector used by the language runtime (§5.5).

5.1 Direct object-graph snapshots

DOSS *direct object-graph snapshots* (henceforth referred to as *snapshots*) represent a single object graph, starting from the *snapshot root object* that resides in the runtime heap. In other words, a snapshot of a root object is the transitive closure of all objects referenced from the root object. Snapshots do not contain any objects other than those present in the root object graph. DOSS places all snapshots in the snapshot heap (§5.2) during their lifetime.

Snapshots are a result of *direct* snapshotting performed by DOSS, i.e., the objects captured in a snapshot are recognizable by the language runtime and can be accessed in the same way as objects in the runtime heap. Unlike objects in the runtime heap, that might be part of the same object graph but scattered across the heap, snapshotted objects are collocated in a snapshot, with DOSS providing support for various object laying-out strategies (§5.4) inside snapshots.

DOSS direct object snapshots are often larger compared to text-based and binary S/D snapshots. This is because objects in the language-runtime heap contain accompanying metadata and need to be memory-aligned and padded, which increases the total amount of memory needed to represent the objects in the snapshot. However, there are multiple advantages to snapshotting data directly. Firstly, it enables data deduplication using object references instead of copies, which by design includes support for circular references. Secondly, it incurs no data loss during snapshotting (e.g., typing information), which is often the case for S/D. Finally, and most importantly, it allows for fast snapshotting without transforming objects and constant-time snapshot loading.

DOSS is primarily designed for creating and managing *data* snapshots. Data corresponds to a structured collection of related information stored as a single object that can be efficiently persisted and loaded without any language-runtime coordination. Examples of data objects include application configuration, application resources such as datasets or data caches, and application dependencies in the form of auxiliary data files such as ML models. Compiled code is not considered data, since loading compiled code involves code installation and language-runtime coordination during loading. Although compiled code, in principle, can be snapshotted, DOSS does not provide any routines for code installation upon loading a code snapshot.

5.2 Snapshot heap

The snapshot heap is an auxiliary heap backed by a reserved memory region that accommodates snapshots during application execution. The memory for the snapshot heap is reserved in the early stages of language runtime initialization, after the initialization of the image heap and before the initialization of the runtime heap. During application execution, DOSS dynamically commits and releases memory in the snapshot heap memory region.

Figure 5.2 shows the placement of the snapshot heap memory region in the address space of the executing language-runtime process and illustrates the internal structure of the snapshot heap. The snapshot heap resides immediately after the image heap, with a configurable offset from the heap base. This design allows arbitrary runtime heap configuration, with no restrictions during application execution, i.e., the runtime heap can dynamically grow and shrink as long as its base address lies beyond the snapshot heap.

The memory layout of the snapshot heap, shown in Figure 5.2, consists of a committed memory region for the snapshot heap *metadata section* and the reserved memory region for an arbitrary number of snapshot slots. A snapshot slot is a unit-section of the snapshot heap that

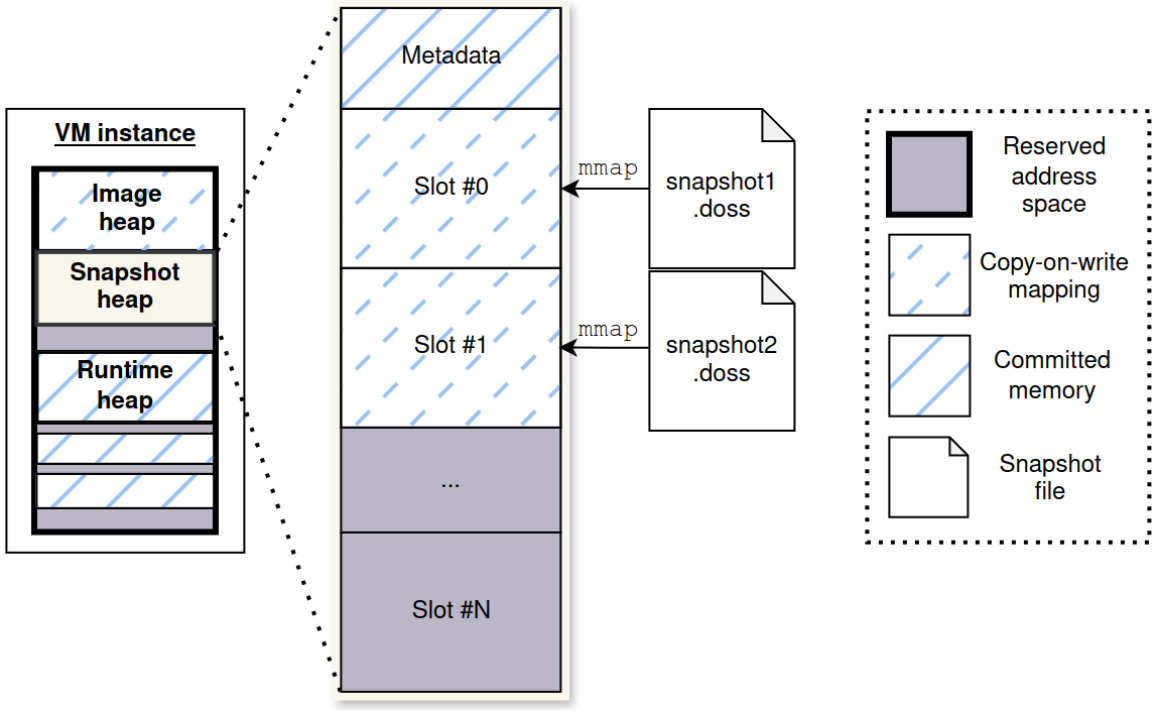


Figure 5.2: The architecture of the DOSS snapshot heap — its position snapshot heap in the executing language-runtime address space, and the breakdown of its components.

hosts a single snapshot. Each snapshot is associated with exactly one snapshot slot during the entirety of its lifetime. DOSS dynamically commits memory regions for individual snapshot slots and establishes shared memory mappings, depending on the operation that takes place (§5.3). Snapshot slot memory is committed before snapshot creation, and shared memory mappings are established when a persisted snapshot is loaded from disk into a snapshot slot.

Metadata section

The metadata section of the snapshot heap is private to individual language-runtime instances. It contains the snapshot heap configuration and a table of occupied snapshot slots in the current language-runtime instance. Snapshot heap configuration consists of the memory offset to the image-heap base, the number of snapshot slots, and their individual sizes. DOSS does not require that all snapshot slots have the same size, although, for simplicity, the illustrations in this section will use equal sizes for all snapshot slots.

DOSS initialization happens after the language runtime initializes the image heap. At that point, snapshot heap configuration is read from both the image-build time options and image-run time options. Based on the snapshot heap configuration, DOSS initialization routine calculates the size of the required metadata section, and the entire snapshot heap. The address range needed for the entire snapshot heap is then reserved, while the address range needed for the metadata section is committed. Finally, the metadata section is initialized.

DOSS stores the snapshot-slot *occupancy table* in the metadata section. The occupancy table keeps track of all snapshots currently loaded into the snapshot heap. In the example from Figure 5.2, the occupancy table consists of `N` entries corresponding to the snapshot slots. Entries corresponding to slots `0` and `1` are marked as occupied, so any attempt to operate on either of these slots will be prevented while that slot is occupied.

Snapshot slots

Snapshot slots are the building blocks of the snapshot heap. The number of slots in the snapshot heap is arbitrary, i.e., DOSS imposes no restrictions on the amount of slots. Each slot is characterized by a particular *base offset* from the snapshot-heap base pointer, and its *capacity*. The capacity of a slot is the maximum size in bytes the slot can have. It can either be set manually or inferred by DOSS based on the specified total size of the snapshot heap and the specified number of snapshot slots, in which case all slots have the same capacity.

Each snapshot slot can either be free or occupied. Free slots do not have any snapshots associated with them, and the memory region for such slots is uncommitted. If a slot is occupied, that means there is a snapshot associated with it, or that an executing snapshotting operation *targets* that slot. Slot occupancy mark is kept in the occupancy table, which is part of the snapshot heap metadata section.

If a slot has a snapshot associated with it, the *size* of the slot corresponds to the current amount of used memory within the slot. Slot size can never exceed the slot capacity.

Snapshot references

DOSS snapshots can be loaded into multiple executing language-runtime instances. Each language runtime can map the image heap to an arbitrary location in the process address space. For object references captured in a snapshot to remain valid across multiple language-runtime instances, DOSS requires the language runtime to use relative instead of absolute object references (e.g., as is the case for the *GraalVM* language runtime §3.3).

Figure 5.3 shows how DOSS relative object references stored in a snapshot remain valid after the snapshot has been memory-mapped into another language-runtime instance. The object graph contained in the snapshot consists of three objects, each represented with a distinct color. The root object is marked with an orange color.

DOSS records the image-heap offset in the snapshot, in this case `0x1000`. Upon loading the snapshot, DOSS makes sure to map the snapshot file at the same offset to the image-heap base. In this example, the image-heap base is `0xA000`, so the snapshot is memory-mapped at address `0xB000`. That way, all references to the snapshot heap from the runtime heap (e.g., looking at the reference at address `0xE008`) remain valid. The language runtime uses pointer arithmetic to add the image-heap base to the relative reference of the object being accessed, which results in a correct absolute address[1]. This way, every language-runtime instance can use the same relative pointers to access the DOSS snapshot heap, regardless of the absolute position of the snapshot heap in the address space, across all snapshot slots.

5.3 Snapshotting operations

DOSS can SNAPSHOT object graphs from the runtime heap, STORE snapshots from the snapshot heap to disk, efficiently LOAD snapshots from disk into the snapshot heap, and UNLOAD loaded snapshots from the snapshot heap back to the runtime heap. Figure 5.4 is used throughout this section to illustrate the descriptions of individual DOSS operations.

[1]Language runtimes such as *GraalVM* store the image-heap base address in a CPU register for the pointer-arithemtic to be folded (see §3.3).

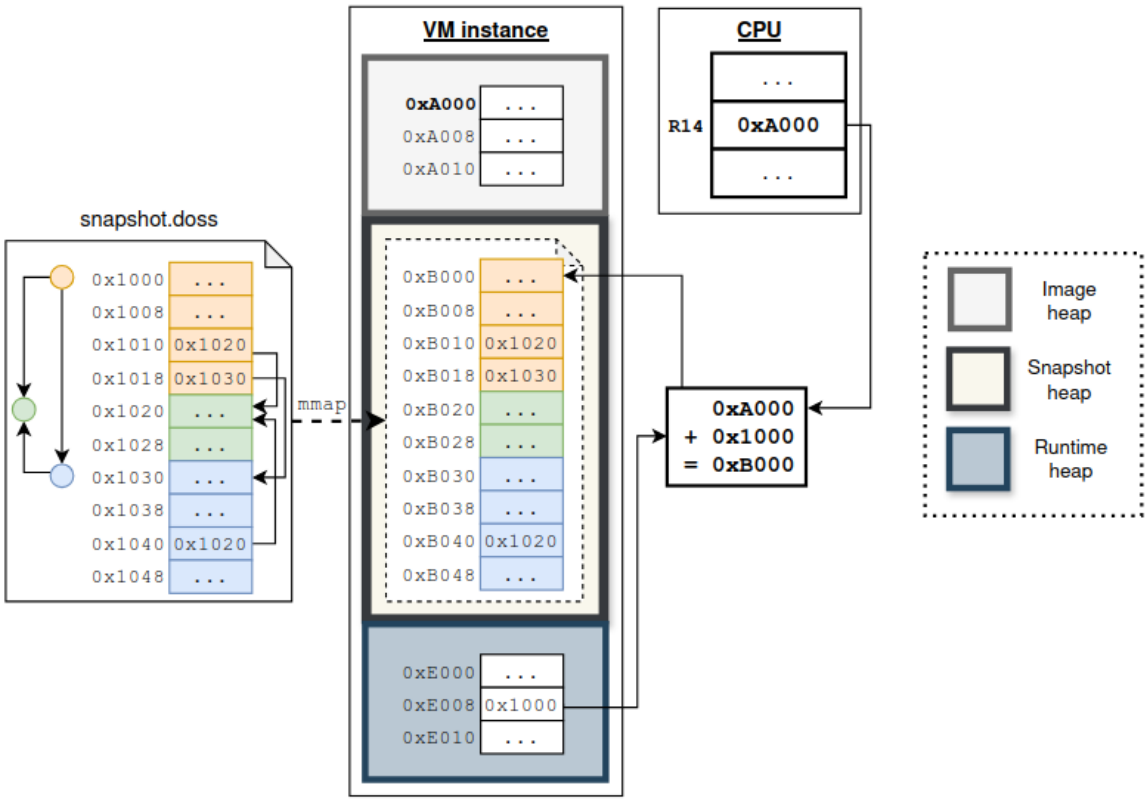


Figure 5.3: Accessing objects (marked with orange, green, and blue) loaded into the Doss snapshot heap from the runtime heap, using pointers relative to the heap base address (`0xA000`). Absolute address values are shown on the left side of each address.

Snapshot operation

The snapshot operation (Figure 5.4, operation labeled as №1) creates a snapshot of an object graph starting from the root object that resides in the runtime heap. The resulting snapshot is placed in the requested destination slot. The object graph reachable from the root object can contain references to objects outside the runtime heap. Depending on how the language runtime is organized, these references might point to the image heap (e.g., to class hubs or other pre-initialized objects in the image heap). Additionally, when using Doss, objects from the runtime heap could also reference other objects in the snapshot heap.

The snapshot operation performs a breadth-first-search[2] reference traversal starting from the snapshot root object, copying all reachable objects that do not reside in the image heap into the chosen destination snapshot slot. The snapshot operation records all copied objects and maintains a map associating references of original objects from the runtime heap to references of their copies in the destination snapshot slot. All references in the copied object graph, except those pointing to the image heap, are updated during copying so that, upon completion of the snapshot operation, the created snapshot does not reference the runtime heap, as shown in Figure 5.4 (the resulting blue object graph after operation labeled as №1).

If the snapshot operation encounters an object that contains a runtime-specific state (e.g., open file descriptors or started threads) during object-graph traversal, an error is reported, and

[2]BFS is chosen as it respects the order of fields in object memory layout. It allows Doss to perform a shallow copy of the object memory before recursively copying the fields in memory-layout order. BFS is also used by the garbage collector to move objects between spaces during garbage collections.

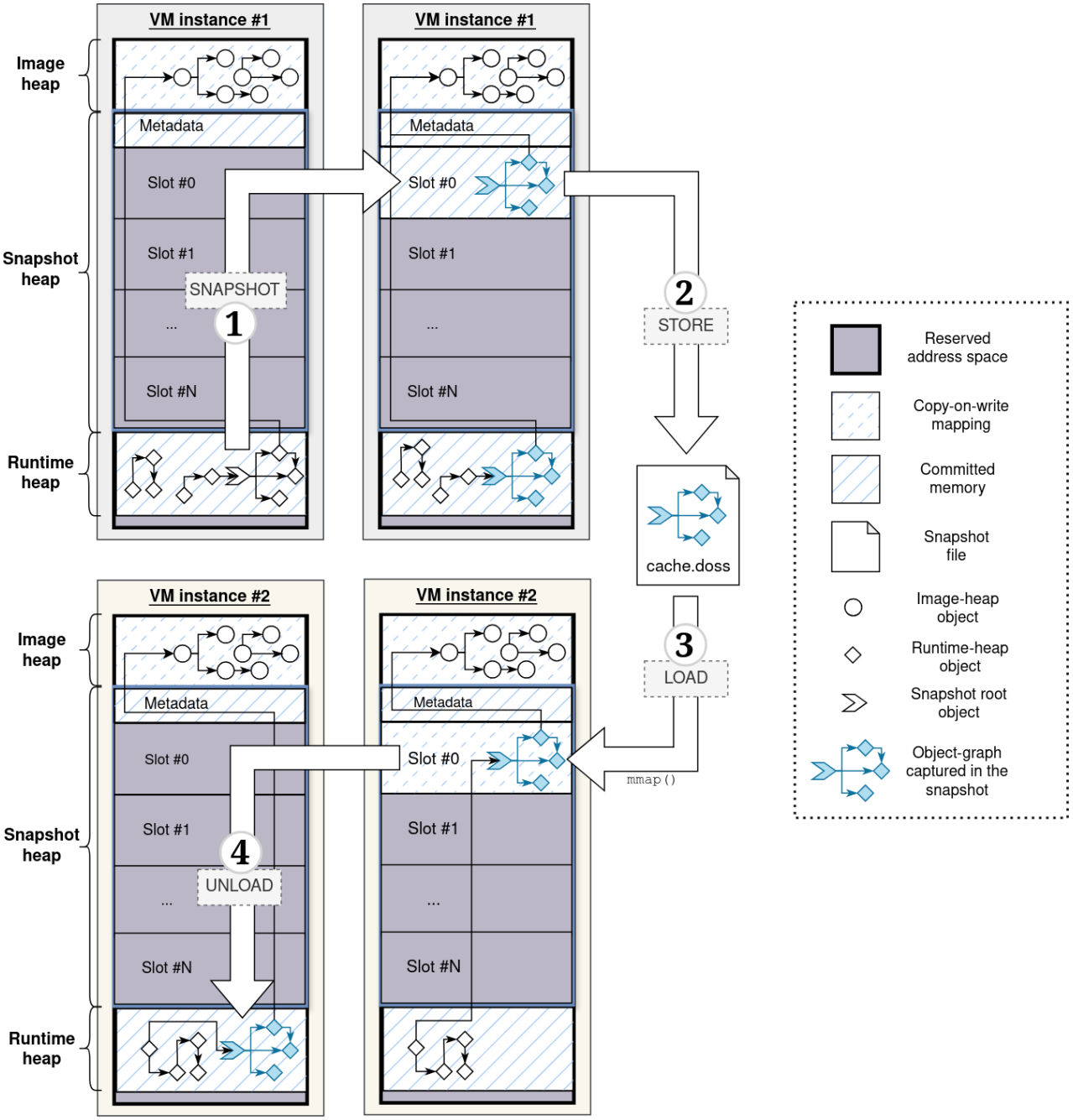


Figure 5.4: An overview of DOSS operations — SNAPSHOT (operation labeled as №1), STORE (operation labeled as №2), LOAD (operation labeled as №3), and UNLOAD (operation labeled as №4).

the operation is cancelled. This ensures that the resulting snapshot does not capture any state that is specific to the language runtime that created the snapshot.

The original object graph coexists with the snapshot. Thus, modifications to the original object do not affect the shared snapshot. This means that the garbage collector can safely manage original objects without affecting snapshotted object graphs in the snapshot heap.

STORE operation

The STORE operation (Figure 5.4, operation labeled as №2) writes a snapshot (previously created via the SNAPSHOT operation) from its associated slot in the snapshot heap to a binary *snapshot file* on disk. The snapshot file starts with a *snapshot header* of fixed size, and the snapshotted object graph that is placed after the header.

The snapshot header stores information that helps DOSS load the snapshot to the correct offset in the snapshot heap. The header stores identifiers of the language runtime that created the snapshot (see §5.3), the image-heap offset of the associated snapshot slot in the snapshot heap, and various metadata that helps determine the layout of the objects inside the snapshot. DOSS supports multiple object laying-out strategies (§5.4) in the snapshot file.

The STORE operation does not perform any modifications to the snapshotted object, i.e., it does not impose any serialization overhead. Storing a snapshot does not automatically remove it from the origin snapshot slot. Removing a snapshot from the snapshot slot can be performed through the UNLOAD operation.

LOAD operation

The LOAD operation (Figure 5.4, operation labeled as №3) establishes a copy-on-write memory mapping between the snapshot file on disk and its appropriate slot in the snapshot heap. Once the mapping is established, a reference to the snapshot root object in the loaded snapshot is returned, and the object can be referenced from the runtime heap. The language-runtime instance that created and stored the snapshot is henceforth referred to as the *origin language-runtime instance*, while the language-runtime instance attempting to load that snapshot is henceforth referred to as the *destination language-runtime instance*.

When the LOAD operation is requested, DOSS first reads the snapshot file header and determines the compatibility of the snapshot with the destination language-runtime instance. A snapshot is eligible for loading into the destination language-runtime instance if the following conditions are met:

Language-runtime-instance compatibility — Origin and destination language-runtime instances need to be mutually compatible. Language-runtime instance compatibility encompasses the set of known and initialized types and their identifiers, the size of the image heap, and the version of the executing runtime environment. Metadata required to determine language-runtime compatibility is stored in the snapshot header. DOSS computes a single identifier based on the compatibility metadata upon storing the snapshot, making the pre-loading checks in destination language-runtime instances as efficient as possible.

Garbage collector compatibility — The destination language-runtime instance uses a supported garbage collector whose heap laying-out strategy matches that of a snapshot file.

Snapshot-slot compatibility — The snapshot heap configuration of the destination language-runtime instance is compatible with the snapshot, i.e., the offset from the image-heap base of the destination language-runtime instance points to a slot in the snapshot heap (hereafter referred to as the *candidate slot*), the candidate slot is not occupied, and the snapshot can fit inside the candidate slot (the capacity of the candidate slot is greater or equal to the size of the snapshot).

DOSS proposes a safe loading mechanism by combining closed-world assumptions of an AOT compiler with a language-runtime implementation that uses relative object references. This allows the LOAD operation to efficiently solve three key problems:

Problem 1. *Ensuring the validity of object references in all language-runtime instances.*

Solution. By ensuring that the offset from the image-heap-base pointer in the origin language-runtime instance matches that of a loaded snapshot in the destination language-runtime instance, relative object references stored in the snapshot work across multiple language-runtime instances (see §5.2). This allows DOSS to LOAD snapshots into multiple compatible language-runtime processes or memory isolates (such as the *GraalVM* memory isolates, see §3.3) that have incompatible absolute object references. □

Problem 2. *Loading only objects of known types.*

Solution. The closed-world assumption of the underlying AOT compiler, combined with the language-runtime instance compatibility performed by the LOAD operation, ensures that the LOAD operation cannot load an object whose type is unknown to the destination language-runtime instance. □

Compilers that perform class initialization at image-build time defer initialization of classes deemed to be unsafe for build-time initialization to image-run time (e.g., *GraalVM Native Image*, see §3.2). Those run-time initialized classes might be initialized in the origin language-runtime instance, and not initialized at the time of the LOAD operation in the destination language-runtime instance. Therefore, DOSS has to solve an additional problem in such environments, described below.

Problem 3. *Ensuring that classes for all objects captured in the snapshot are initialized in the destination language-runtime instance.*

Solution. DOSS considers two approaches to solving this problem. The first approach is to allow objects of run-time initialized classes in snapshots, and force class initialization in the destination language-runtime instance during the LOAD operation. The second approach is to only allow snapshotting of objects whose classes are initialized at image-build time.

The first approach requires DOSS to expand the snapshot header to include a set of type identifiers for all objects present in the snapshot. To optimize for cases where there are large amounts of objects stored in the snapshot, the set can exclude classes that are always initialized at image-build time, such as common library classes. The LOAD operation would then read the set from the snapshot header and run class initializers for those classes in the destination language-runtime instance.

The overhead of this approach is two-fold. First, additional memory is needed in the snapshot header, making its size dynamic instead of fixed. For large snapshots, the amount of class identifiers stored in the header could result in a significantly larger snapshot header. Second, the overhead of class initialization at image run-time could be significant. Class initialization runs static initializers, which can run arbitrary code that can impact the performance of the LOAD operation. Even in the cases when class initialization is cheap, DOSS would still need to perform class-initialization checks for the entire set of classes in the snapshot header.

The second approach has the benefit of avoiding the class initialization checks in the destination language-runtime instance, and avoids the extra metadata needed in the object header to facilitate class initialization. The assumption of having only build-time initialized classes in

the snapshot is justified, as optimized environments such as cloud-native architectures desire to initialize most of the classes at image-build time, for maximum startup performance.

Considering these trade-offs, DOSS by default does not allow snapshotting of objects whose classes are initialized at run-time. This behavior can be overridden via an option, in cases where the performance of the LOAD operation is traded for higher snapshot compatibility across language-runtime instances. □

UNLOAD operation

The UNLOAD operation (Figure 5.4, operation labeled as №4) detaches the snapshot from its snapshot slot and offloads the object graph contained in the snapshot to the runtime heap. It checks if the memory for the snapshot slot has been committed or mapped, and releases it accordingly. The memory for the snapshot slot has been committed if the snapshot has been created in the same language-runtime process via the SNAPSHOT operation, in which case it can be safely uncommitted. Otherwise, the snapshot has been memory-mapped into the destination snapshot slot via the LOAD operation, hence the UNLOAD operation unregisters the memory mapping. The destination snapshot slot is marked as available for subsequent DOSS operations, regardless of which memory operations are performed.

Objects contained in the loaded snapshot can be modified during the program's execution, or still referenced from the runtime heap, so the UNLOAD operation triggers a *snapshot-slot scan* before unloading the snapshot. Snapshot-slot scan is a modified form of garbage collection of the snapshot heap, supported by the language-runtime garbage collector (explained in detail in §5.5). Snapshot-slot scan moves objects from the snapshot slot to the runtime heap and ensures that references to the unloaded slot remain valid after its contents have been moved to the runtime heap (see §5.5, Figure 5.7).

5.4 Object layout

Laying-out strategy for objects contained in the snapshot is determined by the garbage collector. DOSS supports *linear* and *chunked* object laying-out strategies. The snapshot slot and its associated snapshot correspond to a collection of objects referred to as a *space* or a *region*, in garbage-collector terminology.

Linear object laying-out strategy is used for garbage-collector implementations that use linear object layout for spaces. This strategy is used for laying out objects inside *G1GC* [97] regions, which have equal sizes depending on the total heap size, ranging from `1MB` to `32MB`. This strategy involves laying out objects one after the other, as shown in Figure 5.5.

Chunked object laying-out strategy is used for garbage-collector implementations that group smaller objects together in chunks and process them as a whole. This strategy is used for laying out objects inside *GraalVM* garbage collector which uses the chunked laying-out strategy inside spaces, which all have equal size of `512KB` (see §3.3). This strategy involves laying out objects in either aligned or unaligned chunks, depending on the object size, as shown in Figure 5.6.

Since aligned chunks need to be aligned to specific memory addresses that are divisible by the aligned chunk size, the chunked layout strategy can introduce severe memory fragmentation inside the snapshot slot and, as a consequence, make the snapshot file unnecessarily larger. Memory fragmentation can occur when the chunks are placed one after the other, since unaligned chunks have no alignment requirements, and aligned chunks coming after them might

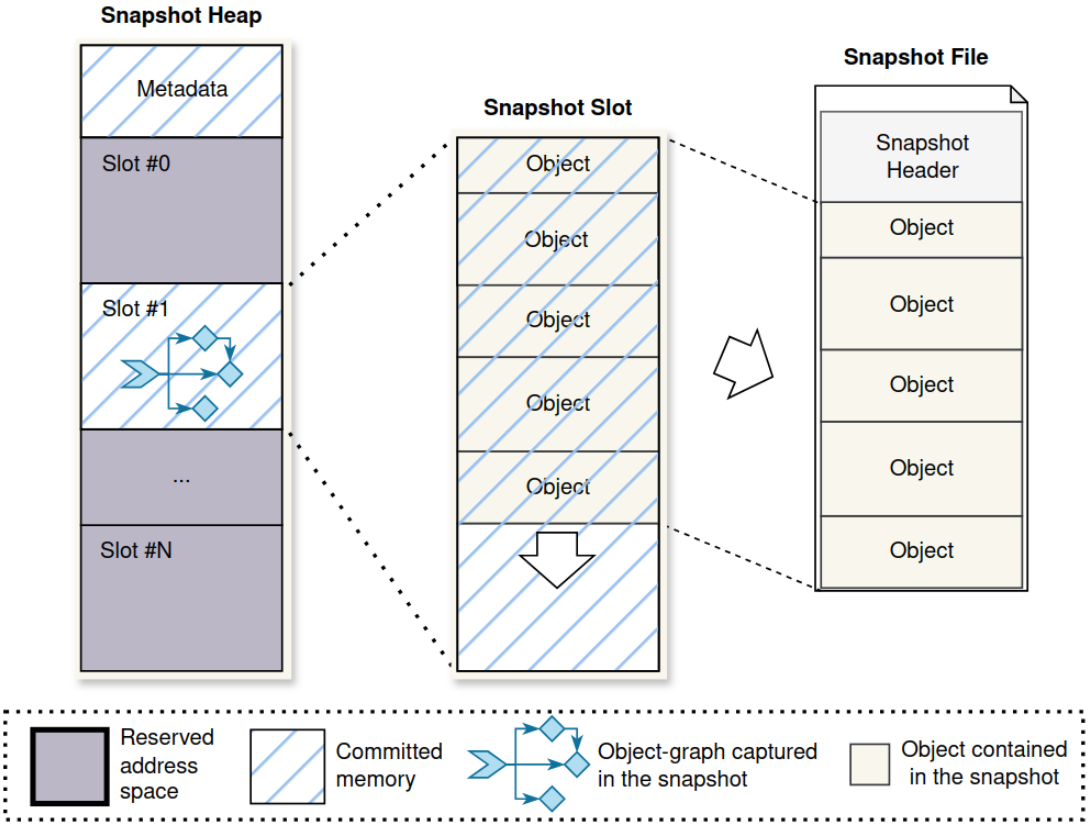


Figure 5.5: Linear object laying-out strategy inside DOSS snapshots.

need to be padded. In order to avoid padding, aligned chunks need to be laid out one after the other.

DOSS solves this issue by laying out aligned chunks together from one end of the snapshot slot, and laying out unaligned chunks together from the other end of the snapshot slot, as shown in Figure 5.6. Both chunk groups grow towards each other until the space between them is used completely. When these groups are stored in a snapshot file, the empty space is removed, resulting in no fragmentation attributed to padding between chunks. The amount of free space between the chunk groups is recorded in the snapshot header, so that the LOAD operation can recreate the free space between the chunk groups in the destination slot.

However, another way memory fragmentation may take place with the chunked layout strategy is inside the aligned chunks themselves. Aligned chunks contain many small objects that do not necessarily fill the entire chunk, introducing potentially significant memory fragmentation[3]. This is a foundational limitation of the chunked laying-out strategy, for which DOSS cannot provide a solution while retaining constant deserialization time complexity.

5.5 GC integration

For the snapshot heap to coexist with the managed runtime heap, DOSS has to integrate with the language-runtime garbage collector and modify its behavior during regular runtime-heap garbage collections and snapshot heap operations.

[3]Consider a situation where the aligned chunk size is set to `512KB` and every object has size of `257KB` (slightly larger than half of the chunk size), requiring a separate aligned chunk for itself.

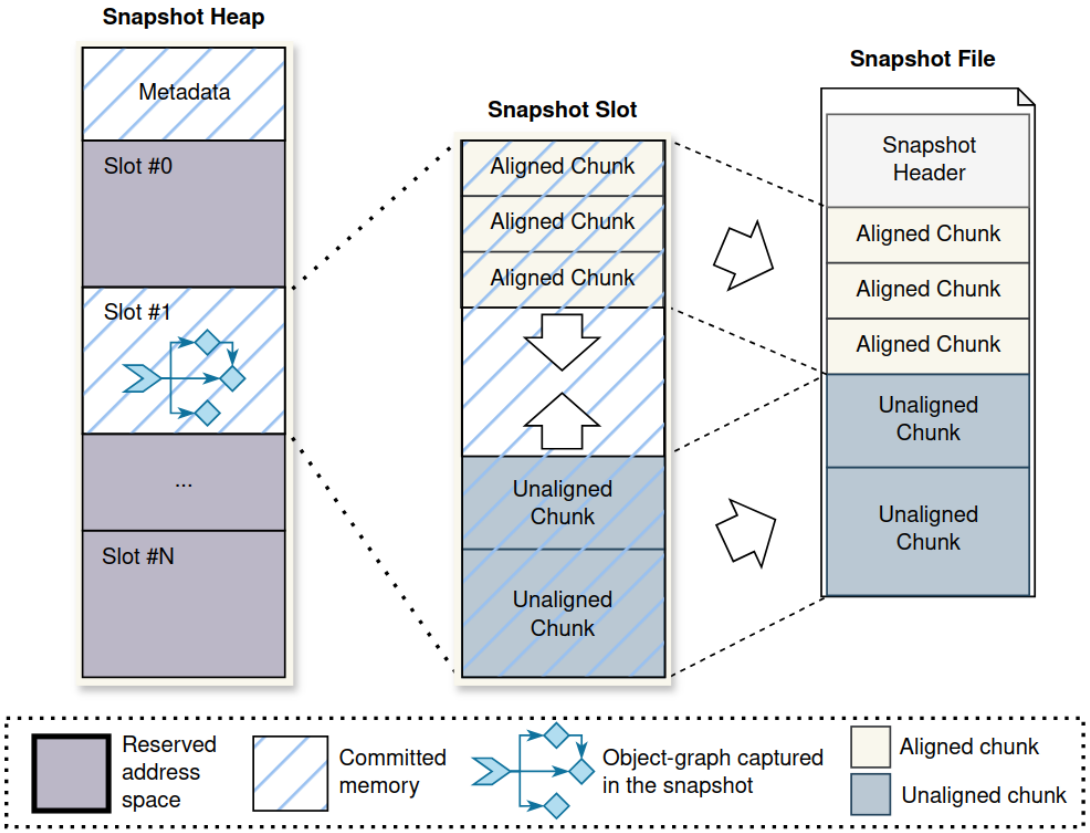


Figure 5.6: Chunked object laying-out strategy inside DOSS snapshots.

Regular garbage collections of the runtime heap treat snapshot-heap objects in the same way as image-heap objects. Garbage collector needs to scan the writable sections of the image heap and patch references to the runtime heap when runtime-heap objects are moved (e.g., when they are promoted or when the spaces are compacted). Both image-heap and snapshot-heap objects do not move during garbage collections, and remain in the image-heap and snapshot-heap memory regions, respectively.

Unlike the image heap, the snapshot heap is not immortal, i.e., the objects inside the snapshot heap can become eligible for collection. One case when this happens is during the UNLOAD operation, which may trigger a snapshot-slot scan. The snapshot-slot scan is a special type of garbage collection which moves objects from the target snapshot slot to the runtime heap (Figure 5.4, operation labeled as №4) and traverses dirty regions to discover and patch outside references to unloaded objects. Snapshot-slot scan operates in a similar way to how generational garbage collectors process references from old-generation objects to young-generation objects.

Modern garbage collectors, such as the *GraalVM Serial* GC (see §3.3), use remembered sets and/or card marking in conjunction with write barriers to mark objects or groups of objects as referents to objects of interest. DOSS modifies write barriers to mark referents to the snapshot heap, and supports both remembered sets and card marking. Snapshot-slot scan only moves objects to the runtime heap when it detects outside references to target slot contents.

Figure 5.7 shows an example of a snapshot-slot scan that occurs during the UNLOAD operation. An object in the target snapshot slot (blue) refers to objects (green) in the runtime heap (at `0xA000`), but also to other objects in the snapshot heap that reside in different snapshot slots (at `0x1000`). Objects from the runtime heap (orange) also have references to the target snapshot slot (at `0xA008`). In this situation, the snapshot-slot scan needs to move objects

from the snapshot slot to the runtime heap.

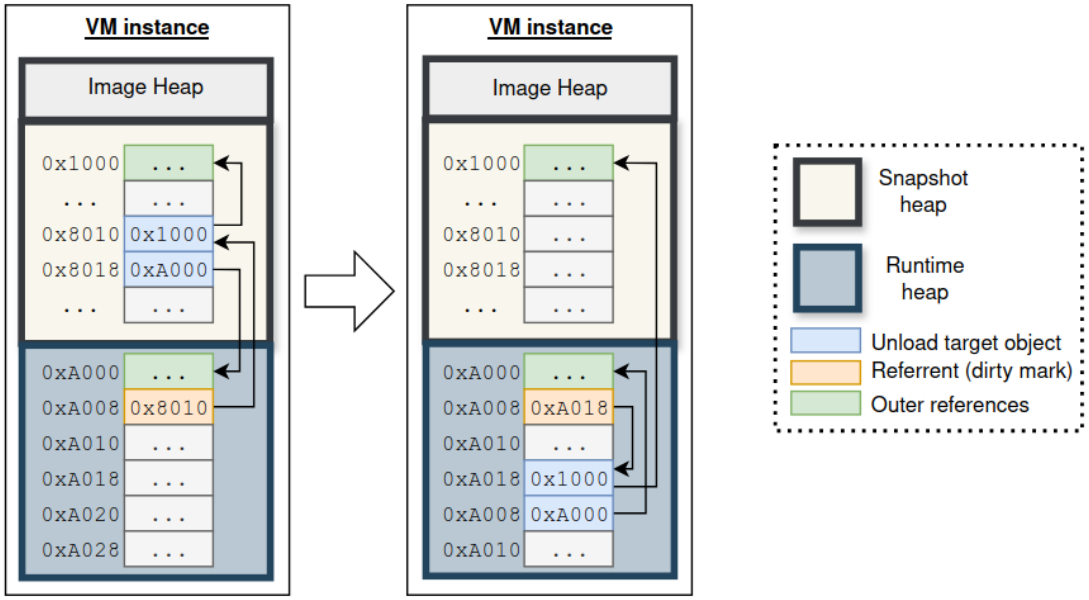


Figure 5.7: An example of a snapshot-slot scan during the UNLOAD operation — an object from the target snapshot slot (blue), referred to from the runtime heap (orange), is moved to the runtime heap. Outer references (green) from unloaded objects to runtime-heap objects or snapshot-heap objects in other snapshot slots remain valid.

Snapshot-slot scan offloads objects to the runtime heap (Figure 5.7, right). It performs no modifications to moved objects, keeping outer references (green) to other snapshot slots (e.g., `0x1000`) or runtime heap objects (e.g., `0xA000`) valid. Dirty objects (orange) that reference moved objects that now reside in the runtime heap are discovered and patched to point to the new location in the runtime heap (discovered reference at `0xA008` is patched). With the snapshot-slot contents offloaded to the runtime heap and all referents patched, the UNLOAD operation can now safely release the snapshot-slot memory region.

5.6 Envisioned use cases

Direct Object Snapshotting and Sharing (DOSS) provides fast object S/D, which is a prerequisite for efficient data pre-initialization, with the ultimate goal to eliminate cold starts. DOSS can share data snapshots across multiple language-runtime instances, reducing the overall memory footprint of the system. Snapshots can be dynamically created, loaded, and unloaded during application execution, allowing for data hot-reloading without rebuilding or restarting the application. DOSS also supports snapshot versioning with arbitrary version conventions, which can be achieved by creating multiple snapshots of the same object graph at different times.

Using DOSS requires manual management of DOSS snapshots. In other words, DOSS provides maximum flexibility by allowing the end user to decide which objects to capture in a snapshot, as well as when to create, load, and unload snapshots during the execution of an application. The management of snapshots can be implemented by a framework or by an external service, for improved usability and developer experience. An example of DOSS-backed framework support can be found in §6.2, Figure 6.5. By leveraging DOSS, microservice frameworks can provide DOSS data caches to enable efficient cache loading and sharing across multiple application instances (see §7.5).

Fast object S/D

DOSS performs *direct* and *fine-grained* object snapshotting. Direct snapshotting means that objects are captured directly from the language-runtime heap and snapshotted in that same format, without any additional object S/D operations. Fine-grained snapshotting means that DOSS can snapshot a single object, or an arbitrary-sized collection of objects, depending on the use case. Figure 5.8 shows how regular plain-text object S/D (such as *JSON* S/D) compares to snapshotting performed by DOSS.

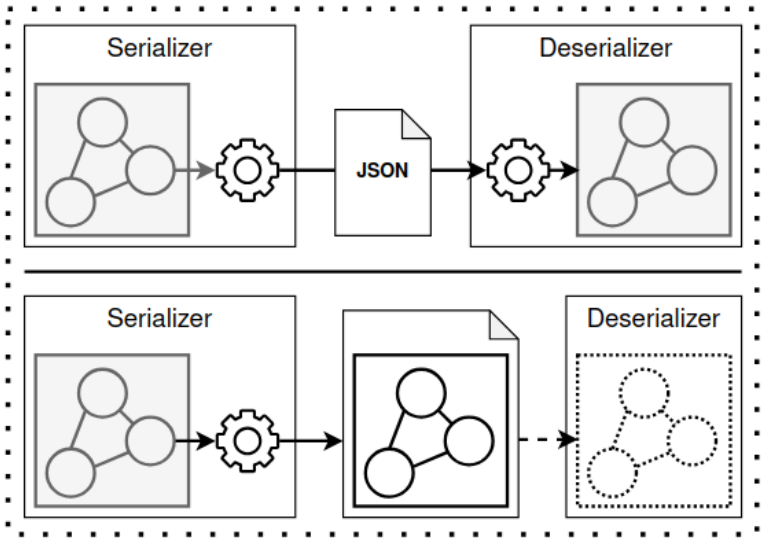


Figure 5.8: Traditional object S/D (top) compared to direct object snapshotting (bottom).

When performing S/D using *JSON* (Figure 5.8, top), the serializer needs to traverse the object graph and transform objects into a *JSON* string, before persisting the string into a file on disk. During this S/D process, information can be duplicated or lost. Data duplication is caused by the lack of references in the *JSON* format, meaning that every time the same object is referred to, it will be duplicated. The deserializer cannot assume that two identical *JSON* object strings are the same object, so the deserializer must duplicate data as well. Data loss comes from the fact that *JSON* does not store typing information. Therefore, e.g., a list and a set can both be encoded as a *JSON* array. Binary S/D can resolve data duplication and loss, but still requires additional work to transform objects during S/D operations.

For environments where fast startup is required, such as serverless computing, serialization overhead is not as important as deserialization overhead. Faster deserialization means faster data loading, which speeds up application initialization. Storing objects in a serialized format means that extra work needs to be done during deserialization.

In traditional S/D solutions, deserialization requires the entire snapshot to be read before transforming the data from a serialized form into objects in the heap. In extreme cases, reading a single data point from a serialized data object results in I/O, CPU, and memory overhead associated with reading the entire snapshot file and deserializing objects contained in the snapshot. Because DOSS snapshots objects directly from the runtime heap, those objects can be efficiently memory-mapped into the application, entirely eliminating the overhead of deserialization (Figure 5.8, bottom). This allows DOSS to load object snapshots in constant time complexity, regardless of the snapshot size. Furthermore, memory mappings established by DOSS allow for *lazy-loading* of application data, as the operating system will page-in memory-mapped pages only when a page fault occurs, as shown in Figure 5.8 (bottom, Deserializer).

Fast S/D capabilities make DOSS useful as a regular object S/D framework, that can be used independently of other DOSS capabilities. Elimination of deserialization from the S/D pipeline makes DOSS a cornerstone of optimized cloud-native architectures where fast communication between services is a requirement. For example, big-data applications that utilize frameworks such as *Apache Spark* [30, 404] frequently exchange large amounts of static data, which can be snapshotted and exchanged using DOSS to reduce communication overhead.

Data pre-initialization

Modern big-data and machine-learning applications need to initialize heavyweight frameworks that load and pre-process large amounts of data before the application is ready to perform its main operations. The data needed by the application is often serialized in memory-efficient and portable formats, e.g., JSON, CSV, or custom binary files. For example, natural language processing (abbr. *NLP*) models can take hundreds of megabytes of disk space for complex natural language processing pipelines (e.g., *StanfordNLP* [294] models). Initialization of such pipelines can easily dominate the application startup time, and can even dominate the total application execution time, e.g., in serverless functions.

DOSS avoids long warmup times by introducing a capability for the application to restore a snapshotted warmed-up state, lazy-loading it in constant time complexity. When applied to NLP models and pipelines, DOSS can snapshot the loaded model, the entire pipeline object in its initialized state, or the result of the pipeline in a warm-up run, and then restore the snapshot to remove most of the application-initialization overhead in the optimized run. Figure 5.9 illustrates this principle, where data preparation is performed in a warm-up run (left), and shows how the resulting warmed-up data snapshot can be loaded efficiently in an optimized run (right).

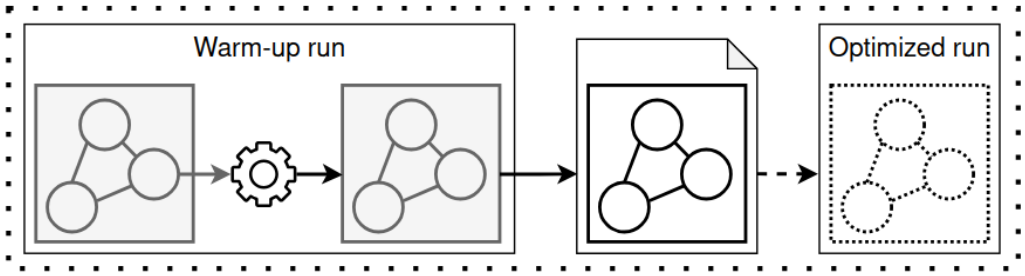


Figure 5.9: Data pre-initialization via direct object snapshots.

However, a warmed-up application state is not always immutable and could evolve depending on the workloads exhibited during execution. One example is application in-memory caches whose contents depend on the workload patterns that aren't known in advance. One reason why this is the case is that workloads encountered during warm-up runs differ from those in production. Warm-up runs are usually done on non-production infrastructure due to application rebuilding that takes place after the warm-up run to bake the warmed-up cache into an optimized application executable. Outdated cache snapshots can cause excess memory consumption and lead to degraded performance. Rebuilding the application in production environments to pre-compile the warmed-up cache for faster subsequent executions requires significant pauses and application restarts.

The same principle of using DOSS for data pre-initialization shown in Figure 5.9 applies to evolving data available only during application execution. In-memory caches can be populated in a warm-up run in the production environment, during which proper workloads are

encountered to create a warmed-up cache snapshot that can be loaded in subsequent application invocations. Furthermore, such caches can be categorized and versioned based on the encountered workload patterns. The application framework can specialize separate application instances to handle specific patterns in data, loading a specific cache snapshot to optimize for the expected workload patterns. For example, DOSS can be applied in linear-algebra and NLP applications to pre-initialize and version-control data and models in order to significantly improve service-initialization times in subsequent executions.

Data sharing

DOSS uses copy-on-write memory mappings to share data across multiple application instances, similarly to image-heap sharing performed by language runtimes such as *GraalVM* (see §3). Copy-on-write memory mappings allow for lazy memory paging-in as data is accessed and efficient sharing of unchanged memory pages. Any modification to objects contained in a loaded snapshot results in a private copying of the modified pages.

Figure 5.10 illustrates DOSS sharing capabilities. The *snapshotter* language-runtime instance (up) pre-initializes and creates a snapshot of the data meant to be shared across application instances. The snapshot of the data is then memory-mapped, i.e., loaded, in subsequent application language-runtime instances (*loaders*, down), leveraging copy-on-write. For example, *Java* microservice frameworks such as *Micronaut* [242] or *Spring* utilize data-cache implementations that can be backed by DOSS for improved cache loading and sharing.

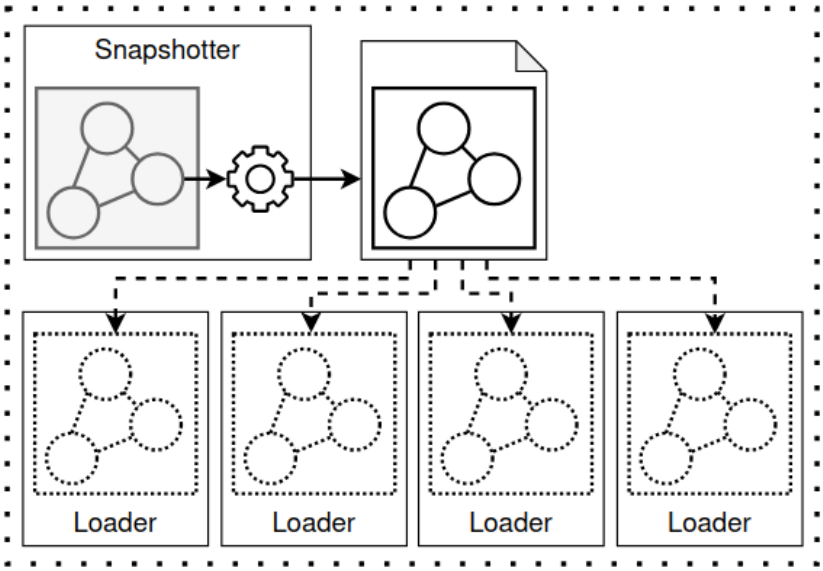


Figure 5.10: Sharing direct object snapshots across multiple language-runtime instances.

DOSS data sharing significantly reduces the memory footprint of the entire system. With DOSS, there are no copies of data objects in memory needed for N application instances, instead of N copies of the same data in traditional deployments. Lazy memory paging-in is desirable in cases where application instances use only a portion of the data. DOSS object laying-out strategies (see §5.4) support lazy loading, with the linear object laying-out strategy being the best fit, as objects are laid out in the breadth-first traversal order.

Hot reloading

In addition to performance-improving features, DOSS architecture provides a number of usability-related features. DOSS operates with a configurable number of snapshots during application execution, allowing the possibility to load multiple snapshots at once, providing the foundation for snapshot versioning, seamless updates, hot-reloading, and version rollbacks. DOSS operations work during application execution to dynamically SNAPSHOT, STORE, LOAD, and UNLOAD data.

Changes to the configuration of AOT-compiled applications usually require application restarts or, in the worst case, the application needing to be rebuilt. Using DOSS to make configuration snapshots allows for dynamic configuration replacement during application execution, without the need for restarts or rebuilds. Figure 5.11 illustrates how DOSS can be used to create versioned data snapshots (data provider, left) and then dynamically replace a loaded snapshot by unloading it (`v1`) from the application (data consumer, right) and loading another version of the same snapshot (`v2`) instead.

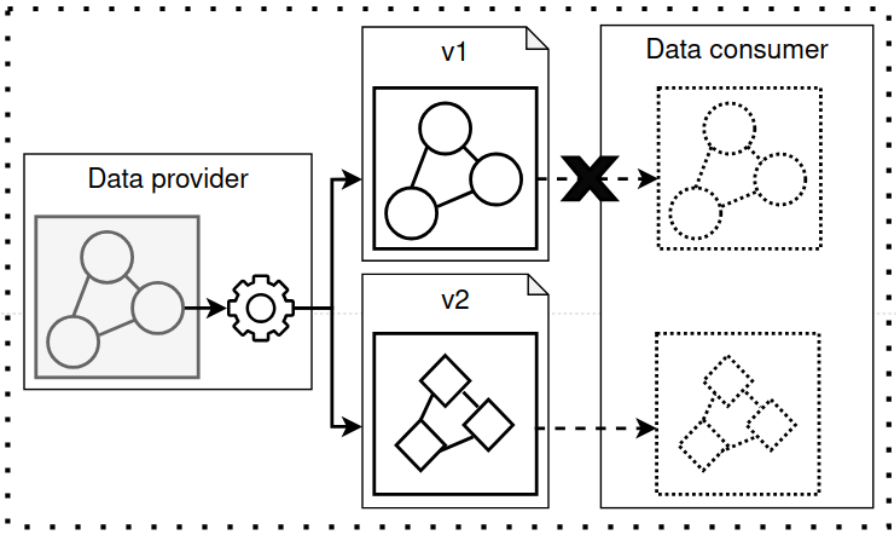


Figure 5.11: Data hot-reloading backed by direct object snapshots.

Examples of using versioned DOSS snapshots in practice include dynamic serverless function configuration and cache or big-data categorizing and versioning. For example, the aforementioned NLP models can be categorized depending on the natural language and versioned so that the snapshot version matches the model version. Propagating updates to all applications that use a particular model of a specific version and category can be done by hot-reloading DOSS model snapshots. Similarly, rollbacks are also propagated in the same way, by unloading the snapshot of the current model and replacing it with a snapshot of a previous version.

Chapter 6

GraalDoss implementation

This chapter presents GraalDoss — a Doss implementation in *Java* integrated into *Oracle GraalVM 24.1*. GraalDoss design and internal architecture is built to provide:

Fast initialization during language-runtime startup, which is crucial for applications that need to start fast (e.g., serverless workers that execute short-lived functions). To achieve this, GraalDoss willingly trades a portion of usability to completely eliminate initialization overhead at image-run time. GraalDoss snapshot-heap configuration is determined at image-build time, and cannot change at image-run time. This means that the entirety of the snapshot heap infrastructure, including its metadata objects, can be pre-initialized at image-build time and placed in the image heap with other pre-initialized objects. Therefore, at image-run time, GraalDoss only needs to reserve the pre-computed amount of memory for the snapshot heap, an operation whose overhead is negligible compared to the overhead of other steps in the language-runtime initialization process.

Extensibility with a modular architecture that makes it easy to extend GraalDoss with support for interchangeable language-runtime components. Even though GraalDoss integrates with the *GraalVM Serial* GC and its object laying-out strategies, GraalDoss architecture allows it to seamlessly support other garbage collector implementations and object laying-out strategies. GraalDoss API provides low-level access to snapshot heap and individual snapshot slots for maximum flexibility and repurposing. Users can implement arbitrary processing prior to GraalDoss operations to implement snapshot versioning, compression, and more.

Reusability of the snapshot heap with the goal of allowing other language-runtime components to use it in their pipeline. GraalDoss snapshot heap is made to be derivable, so that other implementations can extend the snapshot heap and use it as a medium for arbitrary mechanisms, such as message queues, specialized in-memory data and code caches, large-object storages, and S/D-free communication between applications.

GraalDoss provides a user-friendly API (§6.1) for managing direct object snapshots during program execution. GraalDoss is highly configurable, with multiple options that adapt GraalDoss for a multitude of use cases (§6.2). The architecture of GraalDoss is highly scalable and extensible, allowing GraalDoss to serve as a medium for other systems that leverage or use GraalDoss in their pipeline.

6.1 API overview

GraalDOSS implements the DOSS snapshot heap (see §5.2) as an auxiliary heap extension to the runtime heap that supports DOSS snapshotting operations (see §5.3). GraalDOSS integrates with the *GraalVM Serial* GC (see §3.3), supporting its chunked object laying-out strategy (see §5.4). Figure 6.1 shows an overview of the GraalDOSS API, which provides interfaces for creation, storing, loading, unloading, and sharing of direct object snapshots. The API consists of the following interfaces:

Heap interface exposes the implementation of the DOSS snapshot heap. It consists of an arbitrary (`1..*`) number of snapshot slots (`Slot`) and exposes methods that provide selective access to snapshot heap metadata, e.g., configuration and the number of slots. Accessing a slot in the `Heap` based on the slot identifier returns a `Slot` object that can be passed to GraalDOSS snapshot-operation methods.

Slot interface exposes the implementation of the snapshot slot in the snapshot heap. Each `Slot` object represents a single slot in the snapshot heap (`Heap`), with a unique identifier, accompanying metadata, capacity (in bytes), and current size (in bytes) in case a snapshot is attached to the slot. All GraalDOSS slots have the same capacity. Each `Slot` can have at most one `Snapshot` implementation attached to it at a time. GraalDOSS uses slot indices in the snapshot heap as slot identifiers.

Snapshot interface represents a snapshotted or loaded object-graph in the snapshot heap. Each `Snapshot` object contains basic snapshot metadata, such as the size and capacity of the snapshot, as well as a reference to the root of the object-graph captured in the snapshot.

Provider interface exposes the internal implementation of DOSS operations: SNAPSHOT, STORE, LOAD, and UNLOAD, through an implementation of the GraalDOSS operation provider. The provider implementation depends on the currently executing garbage collector and its object laying-out strategy, and removes implementation details of the GraalDOSS operations in the current executing environment.

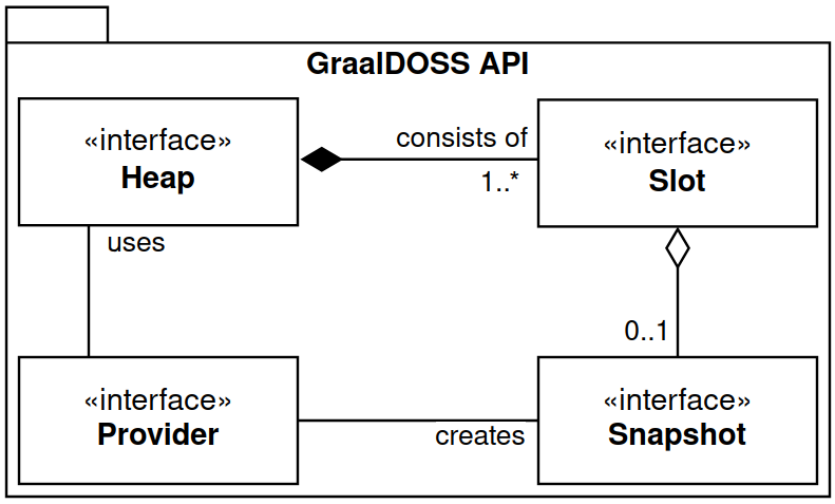


Figure 6.1: GraalDOSS API overview.

Figure 6.2 lists all GraalDoss API methods. The entrypoint to the GraalDoss API is the `DOSS` class that provides static access methods that retrieve internal implementations of public interfaces (`heap()` and `provider()`). The `Heap` interface acts as an accessor to `Slot` and `Snapshot` objects that are passed to GraalDoss operations. The `Provider` interface provides access to GraalDoss operations. GraalDoss operations can throw a special type of exception (`DOSSException`) in case of an error. `DOSSException` consists of a message explaining the error or another exception object that caused the exception.

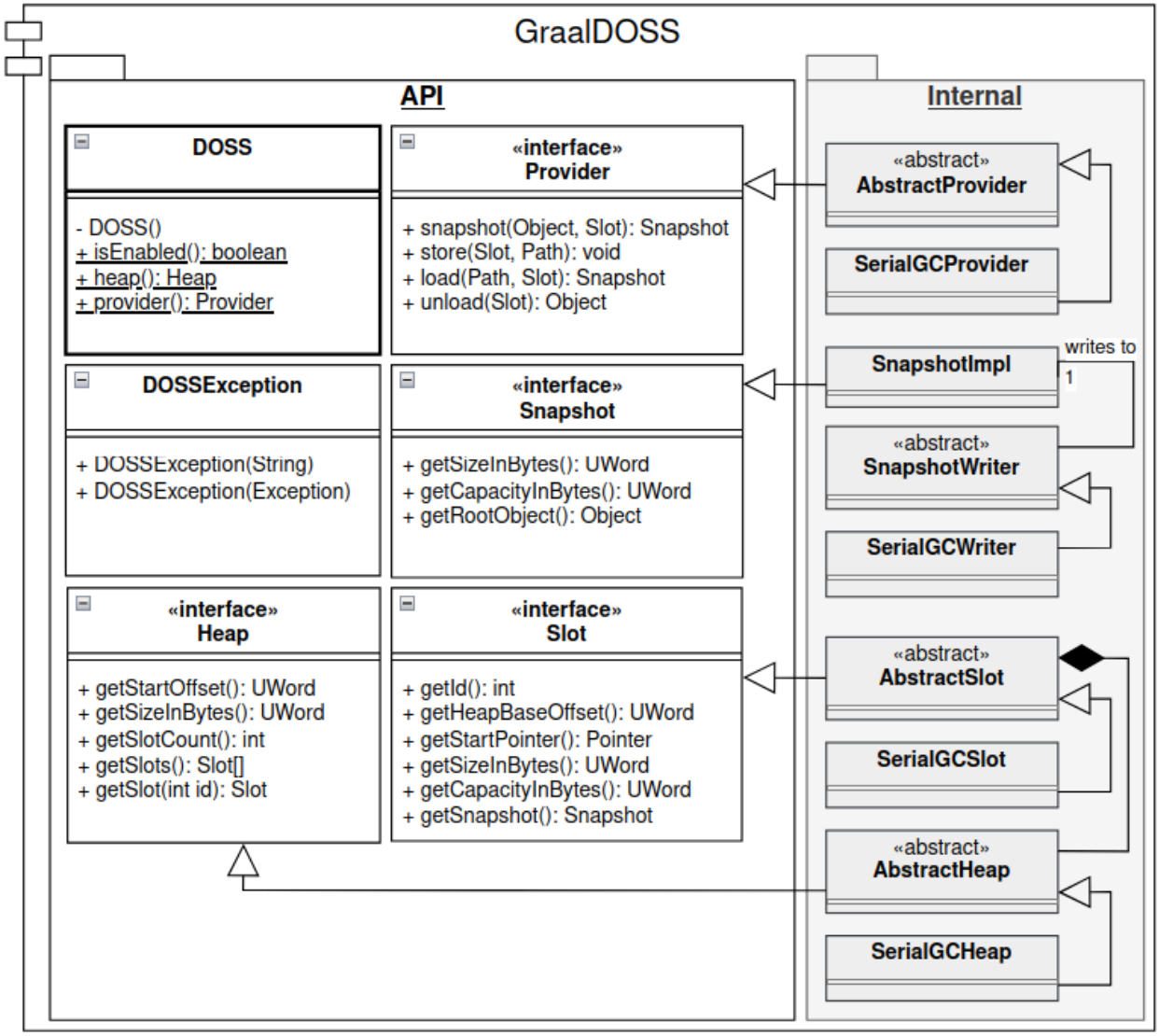


Figure 6.2: Complete GraalDoss API provided in the `org.graalvm.nativeimage` package, and the internal architecture overview.

Figure 6.2 also shows the overview of the internal GraalDoss architecture. The API layer is extended internally by abstract types that extend the `Provider`, `Snapshot`, `Heap`, and `Slot` interfaces. Specialized implementations are then derived from the abstract extensions, such as implementations that provide support for the *GraalVM Serial* GC.

Snapshots are created by the `SnapshotWriter` implementation, which can be tailored to support different object laying-out strategies and optimized for faster serialization performance. `SnapshotWriter` traverses the object graph during the SNAPSHOT operation, copying objects to the target snapshot slot, and keeps track of copied objects. The `SerialGCWriter`

implements the chunk-based traversal and packing objects contained in aligned and unaligned chunks, which the garbage collector would normally treat as a single unit when moving objects in the heap.

6.2 Usage

GraalDoss can be used as a general object S/D library, with additional sharing capabilities. Figure 6.3 shows an example of passing and sharing an object between two language-runtime instances. Both language-runtime instances prepare for GraalDoss operations by accessing the GraalDoss API entrypoint — the `DOSS` class. The `DOSS` class is used to retrieve the provider for GraalDoss operations and the GraalDoss snapshot heap, and determines which snapshot slot to use (Figure 6.3a, line 3, and Figure 6.3b, line 3). After the provider and the heap are retrieved, both language-runtime instances can perform GraalDoss operations.

(a) Origin language-runtime instance

```
1  void storeObject(Object root, Path path, int slotId) {
2      // Prepare for operations
3      DOSS.Provider doss = DOSS.provider();
4      DOSS.Slot slot = DOSS.heap().getSlot(slotId);
5
6      // Perform SNAPSHOT operation
7      doss.snapshot(root, slot);
8
9      // Optional pre-store code
10
11     // Perform STORE operation
12     doss.store(slot, path);
13 }
```

(b) Destination language-runtime instance

```
1  Object loadAndUseObject(Path snapshotFile, int slotId) {
2      // Prepare for operations
3      DOSS.Provider doss = DOSS.provider();
4      DOSS.Slot slot = DOSS.heap().getSlot(slotId);
5
6      // Perform LOAD operation
7      DOSS.Snapshot loadedSnapshot = doss.load(slot, snapshotFile);
8
9      // Retrieve reference to root object in the loaded snapshot
10     Object loadedObjectRoot = loadedSnapshot.getRootObject();
11
12     // Use loaded object
13
14     // (Optional) Perform UNLOAD operation
15     // Returns reference to unloaded root object in the runtime heap
16     return doss.unload(slot);
17 }
```

Figure 6.3: GraalDoss usage example — snapshotting and storing an object in the origin language-runtime instance (6.3a) before loading and using the snapshotted object in the destination language-runtime instance (6.3b).

The first (origin) language-runtime instance (Figure 6.3a) performs SNAPSHOT (line 7) operation on an object graph starting at the root object. The SNAPSHOT operation traverses the object graph, copies it to the provided snapshot slot, and returns a `Snapshot` object. In this example, the `Snapshot` object is ignored, as the snapshot is meant to be persisted to disk. However, the object-graph contained in the snapshot could still be referenced by the returned `Snapshot` object if needed, as snapshots are not automatically unloaded upon storing. The language runtime can perform arbitrary operations before invoking the STORE operation, e.g., it can prepare IO devices where the snapshot would be stored (line 9). Finally, the snapshot is persisted to disk using the STORE operation (line 12) with a destination file path.

After the object is persisted, the second (destination) language-runtime instance (Figure 6.3b) performs the LOAD operation (line 7) on the snapshot file, which returns a `Snapshot` object. `Snapshot` object contains snapshot metadata and a reference to the root of the object-graph contained in the snapshot. Depending on the use-case, the snapshot can be kept in the snapshot heap if the goal is to share the snapshot, but can also be offloaded to the runtime heap through the use of the UNLOAD operation (line 16), in which case a reference to the unloaded root object in the runtime heap is returned. If the reference to the return value of the UNLOAD operation is dropped, the unloaded root object will be collected during the next garbage collection of the runtime heap.

Configuration. GraalDOSS provides a set of options for additional configuration that are passed to the *GraalVM Native Image* compiler at image-build time (options prefixed with `-H:` or `-R:`), or to the application itself at image-run time (options prefixed with `-XX:`). Options passed at image-build time to the *Native Image* compiler prefixed with `-R:` can be overridden at image-run time using the `-XX:` prefix. GraalDOSS provides the following set of options:

`-H:±ObjectSnapshots` — Enables (+) or disables (−) GraalDOSS. When enabled, GraalDOSS is automatically initialized during language-runtime initialization. GraalDOSS is disabled by default.

`-H:ObjectSnapshotSlotCount=<value>` — Sets the number of GraalDOSS slots available in the snapshot heap to the provided value. Default value is set to `8`.

`-H:ObjectSnapshotSlotSizeKiB=<value>` — Sets the capacity of all GraalDOSS slots in the snapshot heap to the provided value, given in `KiB`. This value effectively controls the amount of memory needed for the snapshot heap (not accounting for the size of the metadata section, which is determined based on the number of slots). Default value is set to `10240` (`10 MiB`).

`-H:ObjectSnapshotHeapOffset=<value>` — Manually sets the offset of GraalDOSS snapshot heap to the image-heap base to the provided value. This option is useful for manually aligning the snapshot heaps of DOSS-compatible language-runtime instances that have different image-heap sizes. By default, this option is unset, i.e., snapshot heap is laid-out immediately after the image heap (with possible padding due to page alignment). Values that are smaller than the image-heap size will result in an error during language-runtime initialization.

`-R:±ObjectSnapshotLogging` — Enables (+) or disables (−) verbose logging of GraalDOSS initialization and operations. Verbose logs are disabled by default.

Invocation. Figure 6.4 shows a complete example of enabling and using GraalDoss. The program consists of a simple routine (Figure 6.4a) that creates a list of records (line 7), creates a snapshot of the list (line 8), stores it to file (line 11), unloads the snapshot (line 13), and loads the stored snapshot (line 15). Figure 6.4b shows the program compilation and invocation commands, in which the `GRAALVM_HOME` environment variable points to the GraalDoss-enhanced *Oracle GraalVM* directory. Figure 6.4c shows the resulting output with detailed GraalDoss logs.

Integration. Figure 6.5 presents examples that show how GraalDoss can be integrated into serverless runtimes to provide data sharing across functions. Figure 6.5a uses an infrastructure-level function definition using *AWS* [20] Cloud Development Kit (abbr. *CDK*). In this example, a repository of products is loaded as a previously stored snapshot into processes that execute the `getProductFunction`. Figure 6.5a shows a function definition for the *Hydra* [168] platform, which uses *GraalVM* as part of its virtualization stack. The function performs video processing using the `ffmpeg` program suite. In this example, a function directly invokes the GraalDoss API to share the `ffmpeg` suite between `GraalVM` sandboxes that execute functions.

Error handling. GraalDoss handles errors in two ways, based on the time when the errors occur. Errors that occur during GraalDoss operations, such as user errors (e.g., invalid paths or disallowed objects captured in the snapshot), are recoverable and are handled by throwing an exception of type `DOSSException`. Unrecoverable errors that happen during language-runtime initialization or memory management of GraalDoss snapshot heap cause the language-runtime process to exit with a particular error code (`ERRNO`):

`ERRNO 34` — *Insufficient memory for the snapshot heap.*

`ERRNO 35` — *Insufficient snapshot heap offset; the image heap is too large.*

`ERRNO 36` — *Could not commit memory for the snapshot heap.*

(a) Sample *Java* source code (`Main.java`)

```java
record Point(double X, double Y) {};

public static void main (String[] args) {
    var provider = DOSS.provider();
    var slot = DOSS.heap().getSlot(0);

    var root = List.of(new Point(1.0, 1.0));
    var s = provider.snapshot(root, slot);
    System.out.println("Snapshotted: " + s.get());

    provider.store(slot, Path.of(args[1]));

    provider.unload(slot);

    var loaded = DOSS.provider().load(Path.of(args[1]), slot);
    System.out.println("Loaded     : " + loaded.getRootObject());
}
```

(b) Building and invoking the GraalDoss-enabled image

```
$ $GRAALVM_HOME/bin/javac Main.java
$ $GRAALVM_HOME/bin/native-image -H:+ObjectSnapshots -o main Main
$ ./main -XX:+ObjectSnapshotLogging /tmp/sample.doss
```

(c) Resulting output

```
 [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: SNAPSHOT to slot 0, ptr=0x00007fdaf6980000
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: walking: [chunk=0x00007fdaf6980000 space=null]:
        java.util.ImmutableCollections$List12@0x00007fdaf6980830
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: visit: ptr = 0x00007fdafb9810b0; off=0;
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]:       | obj = [chunk=0x00007fdafb980000 space=
       null]: Point@0x00007fdafb9810b0
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]:       | par = [chunk=0x00007fdaf6980000 space=
       null]: java.util.ImmutableCollections$List12@0x00007fdaf6980830
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: walking: [chunk=0x00007fdaf6980000 space=null]:
        java.util.ImmutableCollections$List12@0x00007fdaf6980830
Snapshotted: [Point[X=1.0, Y=1.0]]
 [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: STORE request for slot 0: /tmp/sample.doss
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: Opening file: /tmp/sample.doss
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: STORE: Successfully wrote 2136 bytes
 [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: UNLOAD request for slot 0
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: Detaching snapshot
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: Freeing slot memory
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: UNLOAD for slot 0 successful
 [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: LOAD request for slot 0: /tmp/sample.doss
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: Performing pre-load checks; size: 2136
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: Snapshot successfully mapped to slot 0
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: Attempting to read snapshotted object
   [I:0x00007fdaf5f80000|T:0x000055b520d8c9c0]: LOAD to slot 0 successful: /tmp/sample.doss
Loaded     : [Point[X=1.0, Y=1.0]]
```

Figure 6.4: A complete example of GraalDoss usage in a simple program (6.4a), building and invoking the image (6.4b), and the resulting output (6.4c).

(a) Infrastructure definition — Amazon AWS [20]

```
Function getProductFunction = Function.Builder
    .create(this, "GetProductFunction")
    .runtime(Runtime.JAVA_21)
    .loadSnapshot(0, "/product-repository.doss")
    .code(Code.fromAsset("../software/",
          AssetOptions.builder()
              .bundling(builderOptions)
              .build()))
    .handler("org.example.aws.ApiGatewayGetProductRequestHandler")
    .memorySize(256)
    .environment(environmentVariables)
    .build();
```

(b) Function definition — Hydra [168]

```
public static HashMap<String, Object> main(Map<String, Object> args) {
    Path p = Path.of(args.get("ffmpegPath"));

    DOSS.Slot slot = DOSS.heap().getSlot(0);
    DOSS.Snapshot ffmpeg = slot.getSnapshot();
    if (ffmpeg == null)
        ffmpeg = DOSS.provider().load(slot, p);
    Object ffmpegObj = ffmpeg.getRootObject();

    byte[] video = getBytes(args.get("videoUrl"));
    String out = ffmpegRun(ffmpegObj, video);
    return formJsonResponse("output", out);
}
```

Figure 6.5: Conceptual integrations of GraalDoss into serverless platforms. Listing 6.5a shows a section of a serverless shopping application infrastructure definition using AWS [20] Cloud Development Kit (CDK). Listing 6.5b shows a serverless video-processing function for Hydra [168] that uses GraalVM memory isolates.

Chapter 7

GraalDoss evaluation

GraalDoss is thoroughly evaluated on an open-source benchmark set (§7.1), in an isolated and controlled environment (§7.2). Evaluation consists of correctness and robustness tests combined with operational performance measurements (§7.3), and generalization of GraalDoss to cloud-native applications that leverage GraalDoss data pre-initialization (§7.4) and sharing (§7.5) capabilities. GraalDoss evaluation is concluded with a summary of obtained results (§7.6).

7.1 Benchmark set

A novel open-source S/D-*benchmarks* set was developed for the purpose of evaluating object S/D and language runtime-integrated C/R solutions. The S/D-*benchmarks* set is tailored towards cloud-native applications, measuring S/D throughput, average and maximum application process memory footprint (RSS), application startup time (i.e., the time until the application becomes ready to process requests), and request-response time (i.e., the time it takes for the application to process the request and return a response to the caller).

S/D-*benchmarks* set consists of a subset of S/D correctness and robustness tests, core microbenchmarks that evaluate S/D operation performance, and macrobenchmarks involving microservice and big-data applications. Combined, the benchmarks evaluate the correctness of an S/D library, and its impact on the efficiency and scaling of applications in a cloud-native environment. All benchmarks in S/D-*benchmarks* are bundled together in a single repository, published on *Zenodo* [307] and *GitHub* [310].

Individual benchmarks in the S/D-*benchmarks* support two execution modes:

JVM mode — In this mode, the benchmark is executed on a stock *JVM* as a regular *Java* application leveraging JIT compilation.

Native mode — In this mode, the benchmark is built as a standalone native image compiled ahead of time with *GraalVM*, leveraging profile-guided optimizations (abbr. *PGO*) to ensure the compilation produces optimized code for hot paths. PGO mode consists of several steps. In the first step, an instrumented benchmark image is built using *GraalVM Native Image*. This image is executed, and profiles are collected. Collected profiles are then fed back to *Native Image* in order to build an optimized benchmark image. In native mode, each benchmark build produces two images: one image with the default set of compiler optimizations and the other one optimized with PGO.

Correctness and robustness tests

S/D-*benchmarks* set provides core unit tests that validate the correctness and robustness of an S/D library. All core unit tests are invoked as a whole in both *JVM* and native modes, with configurable *JVM* options in *JVM* mode and compiler/runtime options in native mode. Unit test harness uses a `Serializer` interface, which specific S/D library serializers need to implement. The `Serializer` interface consists of the following methods:

`void register(Class<?>[])` — Registers given types as templates for S/D operations.

`Object preSerialize(Object)` — Prepares for serialization of a given object.

`Object serialize(Object)` — Serializes an object and returns serialized data.

`Object preDeserialize(Class<?>, Object)` — Prepares for deserialization of a given data object into an object of a given type.

`Object deserialize(Class<?>, Object)` — Deserializes a given data object using the given type as a template. Returns the deserialized object.

`void flush()` — Flushes streams used by the serializer (if any).

`long sizeOf(Object)` — Returns serialized object size.

Correctness tests perform S/D operations on objects that are expected to be handled correctly by the library. These tests evaluate the compatibility of an S/D library on a wide range of commonly used Java classes. Each test prints the string representation of original and snapshotted objects, testing field accesses and virtual `toString` method dispatches. Correctness tests cover:

Simple types — *Java* primitive (value) and boxed types, *Java* records, and strings. Tests involving simple types evaluate basic S/D library capabilities.

Complex types — 25 types of which 15 are custom POJO classes containing fields of simple types and getter/setter methods, and 10 are wrapper types encoding various object graphs. POJO types correspond to typical application model components, such as addresses, customers, departments, documents, employees, invoices, meetings, products, and users. Tests involving complex types evaluate whether an S/D library correctly handles POJO S/D, non-trivial object-graphs, and reference cycles.

Java collection types — 34 implementations of Java `Collection` interface, including `List`, `Map`, and `Set` implementations and their unmodifiable, navigable, and synchronized wrappers. Tests involving *Java* collection types evaluate the capability of an S/D library to correctly perform S/D operations on generic data structures without the loss of information and data duplication commonly present with plain-text S/D libraries.

Common JDK types — 24 Java library classes implementing support for big numbers, time and calendar utils, regular expressions, unique identifiers, string-building utils, IO support, and more. Tests involving common JDK types evaluate whether an S/D library can be used in a broader context, where objects might contain state that needs to be preserved (e.g., regular expressions or unique identifiers).

Reflection types — types that provide support for Java reflection, such as `Class<?>`, `Field`, `Method`, and `Constructor` classes. Tests involving reflection types evaluate whether an S/D library integrates with the dynamic nature and introspection capabilities of *Java*.

Robustness tests exercise how an S/D library handles erroneous or disallowed input in the form of objects that contain execution-specific state. Those can include open file descriptors, running threads and continuations, cancellables, and memory segments. A robust S/D library should recognize such objects early and not attempt to serialize or deserialize them. Instead, the library should throw an exception with a user-friendly error message and shouldn't disrupt the rest of the application.

Core microbenchmarks

S/D-*benchmarks* provides *sd-jmh-native* — a set of microbenchmarks that evaluate the S/D performance of *JVM*-based serializers, inspired by widely-adopted benchmarks for serializers targeting the *JVM* [340, 303]. *sd-jmh-native* uses the *Java Microbenchmark Harness* (abbr. *JMH*) [268], a widely adopted harness for building, running, and analysing microbenchmarks targeting the *JVM*. *sd-jmh-native* is extensible both in terms of the workload set and S/D framework integration. S/D library support are provided by the same `Serializer` interface used for the unit tests.

When executing benchmarks multiple times in the same language-runtime environment, profiles gathered from the previous runs may influence the current execution. This phenomenon is commonly referred to as *profile pollution*. *sd-jmh-native* design is built to minimize profile pollution in all execution modes. In *JVM* execution mode, *JMH* manages profile pollution across multiple benchmark runs through JVM forks. In native execution mode, PGO mode ensures that compilation produces optimized code for hot paths in all S/D libraries. Since in native execution mode all code is compiled and optimized at image-build time, profiles collected during instrumented runs need to match the optimized runs. *sd-jmh-native* generates reproducible workloads to reduce the risk of profile pollution between instrumented and optimized benchmark runs.

sd-jmh-native improves on the state-of-the-art microbenchmarks by providing an extensible workload set and a sophisticated workload generator. The workload set covers a wide range of workloads relevant to cloud-native applications, giving insight into the performance for more complex objects that can be broken down into objects contained in the workload set. Each entry in the workload set is categorized by *workload type* and, optionally, the *size* of the particular workload object. The workload set consists of:

Primitives such as integers or floats.

Strings with alphanumeric characters, where the *size* of the workload corresponds to the length of the string.

Records with primitive and string fields, where the *size* of the workload corresponds to the number of fields in the record.

POJOs commonly used in microservice applications, such as the `Client` POJO that contains 19 fields, including integers, longs, doubles, strings, long arrays, enums, big integers, dates, and lists of companion `Partner` records. `Partner` record holds three fields: id (`long`),

name (`String`) and registration date (`OffsetDateTime`). The detailed breakdown and description of `Client` and `Partner` POJOs can be found in Appendix A.

Java arrays with primitive elements, where the *size* of the workload corresponds to the size of the array.

Matrices backed by *Java* arrays with primitive elements, where the *size* of the workload corresponds to the number of rows and columns of the matrix.

Lists with primitive, record, string, and POJO elements, where the *size* of the workload corresponds to the size of the list.

Maps with integer and string keys, and primitive, record, string, and POJO values, where the *size* of the workload corresponds to the number of map entries.

A *microbenchmark configuration* consists of the target S/D library, microbenchmark operation (serialization or deserialization), number of iterations, workload kind, and optional size of the generated workload objects, and a seed for the randomized workload generator. The seed can be set to an arbitrary value, allowing for reproducible workloads. By default, the workload generator generates randomized workloads with every benchmark run. Combining all possible benchmark options, *sd-jmh-native* provides 2424 possible benchmark configurations across 176 different workload objects.

Cloud-native macrobenchmarks

S/D-*benchmarks* includes several cloud-native macrobenchmarks build on top of state-of-the-art *Java* microservice and machine-learning libraries:

`micronaut-cache` — a Java microservice web API server powered by *Micronaut* [242], a full-stack framework for building modular microservice and serverless applications. `micronaut-cache` serves news headlines, providing API endpoints that retrieve headlines for a given month through *HTTP* requests. It deploys an in-memory cache provider, leveraging *Micronaut*'s built-in caching mechanism and its support for custom cache providers. `micronaut-cache` provides an option of using GraalVM memory isolates to spawn multiple instances of the service efficiently when running in a native deployment mode. That way, all instances share the same starting configuration and pre-compiled code. The benchmark also includes external tooling that sends requests to a news headlines web server and measures response times and memory footprint of the process, including all of its memory isolates.

`stanfordnlp-preload` — a Java microservice that performs natural language processing (NLP) on the input using *StanfordNLP* [294] large language models (abbr. *LLM*) for the English language, version 4.4.0, that run on the CPU. The benchmark uses the *StanfordNLP CoreNLP* Java API to build the NLP pipeline and annotate an input *document*. Input documents are constructed from a set of raw-string inputs, categorized by the size of the input (i.e., the number of sentences, words, and characters). `stanfordnlp-preload` provides several input categories, ranging from small that have 5 sentences, to large that have over 50 sentences. The benchmark measures pipeline creation times (i.e., service warm-up time), document creation times, document annotation times, and total execution time of the application.

All cloud-native macrobenchmarks support both *JVM* and native deployment modes. In native deployment mode, all application dependencies (e.g., framework libraries and resources) are baked into the resulting image. Native deployment mode also generates images optimized with PGO, which require a separate instrumentation run. All benchmarks ensure that the instrumentation run and the actual timed benchmark run use the same workloads, as to minimize profile pollution.

7.2 Evaluation setup

Evaluation is performed in an isolated environment, on hardware specialized for benchmarking purposes. The rigorous evaluation setup minimizes the risk of external factors influencing the obtained results. This includes special hardware, software, and environment configuration.

Hardware configuration. All experiments were performed on a single `X5` compute node with two 18-core *Intel* E5-2699 processors with the frequency of 2.30 GHz, that is part of the internal benchmarking cluster running on the Oracle Cloud Infrastructure. Each node has 64 KB of L1 cache, 256 KB of L2 cache, and 46080 KB of L3 cache. The working memory of the system consists of 512 GB of DDR4 RAM. Additional hardware includes an *LSI MegaRAID SAS-3* 3108 storage controller and *Mellanox Connect-X Infiniband* network interface cards. All performed experiments are also reproducible on machines with lower specifications, such as desktop or laptop computers.

Environment configuration. In all experiments, processor *turbo boost* was disabled and the benchmark process was bound to a single CPU core to reduce the impact of varying CPU frequency and non-uniform memory access on the obtained results [196, 51]. All experiments were performed with *hyperthreading* enabled and CPU *C-states* were disabled (i.e., set to C-state 0) to avoid power-saving modes. All experiments were executed in a 32GB RAM disk (`tmpfs` mount) to avoid IO noise.

Software configuration. All experiments were executed in the same software environment. The operating system used was *Oracle Linux Server* version 7.4 with Linux kernel version 4.1.12. *Java* environment consisted of the *Oracle GraalVM 24.1* for *Java 21*, with default options in all experiments, unless specified otherwise. Garbage collection was performed using the default *GraalVM Serial* GC garbage collector of the *GraalVM* language runtime.

Measurements. In all experiments, multiple measurements were taken to ensure reproducibility and validity of the obtained results. Additionally, multiple warm-up executions were executed prior to measured benchmark executions, in order to improve benchmark stability. Each microbenchmark consisted of 5s of warmup iterations and 10s of measured benchmark executions, obtaining a large amount of benchmark results. Each macrobenchmark consisted of a number of warmup iterations and a number of measured benchmark executions. The number of warmup and measured iterations is determined so that the standard deviation between measurements becomes less than 5%.

Result aggregation. Depending on the measurement, different result-aggregation strategies were used in the experiments. Microbenchmark results were aggregated by *JMH* [268], which

Listing (7.1) S/D-*benchmarks* `Serializer` implementation for GraalDoss.

```
1  @Override
2  Object serialize(Object data) throws Exception {
3      DOSS.Snapshot s = provider.snapshot(data, slot);
4      return s.getRootObject();
5  }
6
7  @Override
8  Object preDeserialize(Class<?> clazz, Object data) throws Exception {
9      provider.store(slot, snapshotPath);
10     provider.unload(slot);
11     return data;
12 }
13
14 @Override
15 Object deserialize(Class<?> clazz, Object data) throws Exception {
16     DOSS.Snapshot s = provider.load(snapshotPath, slot);
17     return s.getRootObject();
18 }
```

computes statistical summaries like mean, median, and confidence intervals. Macrobenchmark results were aggregated manually using *maximum* and *average* aggregation strategies. Reported maximum macrobenchmark measurements (e.g., maximum RSS or maximum latency) are the 99.99-th percentiles of the results obtained in measured benchmark executions. If not otherwise specified, all reported results are averaged from results obtained from measured benchmark executions.

7.3 Operation correctness and performance

GraalDoss core evaluation consists of exercising the basic S/D features of GraalDoss using the S/D-*benchmarks* set. The correctness and robustness of GraalDoss is evaluated through a set of unit tests that exercise GraalDoss operations on a wide range of objects. Core microbenchmarks contained in the *sd-jmh-native* set evaluate the performance of GraalDoss operations. GraalDoss SNAPSHOT operation is compared to system calls that perform memory copying. GraalDoss LOAD operation is compared to system calls that establish memory mappings. GraalDoss UNLOAD operation is compared to garbage collections of the same amount of memory.

Unit testing

Correctness and robustness of GraalDoss is evaluated by adding GraalDoss support for running unit tests contained in the S/D-*benchmarks* set (see §7.1). Listing 7.1 shows the implementation of the `Serializer` interface provided by the S/D-*benchmarks* unit-testing framework. GraalDoss serialization consists of only the SNAPSHOT operation. Preparation for deserialization consists of invoking the STORE operation to persist the snapshot at a predefined `snapshotPath`, and UNLOAD-ing the snapshot in order to free up the target slot for the LOAD operation. The LOAD operation is invoked as part of the deserialization routine, returning the loaded root object.

Table 7.1 summarizes the results of unit tests. GraalDoss passes all 106 correctness and robustness tests, successfully snapshotting and loading objects of commonly used types. GraalDoss successfully operates with complex object graphs, reference cycles, immutable and synchronized *Java* collections, and *Java* reflection classes. In robustness tests, GraalDoss successfully prevents S/D and returns a proper error message.

Table 7.1: GraalDoss correctness and robustness tests results.

Test category	# of tests	Passing	Failing	%
Simple	7	7	0	
Complex	25	25	0	
Java collections	34	34	0	
JDK common	24	24	0	
Reflection	4	4	0	
Robustness	12	12	0	
Total	**106**	**106**	**0**	**100%**

Serialization (SNAPSHOT and STORE) performance

SNAPSHOT operation is evaluated against the theoretical maximum — a single `memcpy` system call. This is because the SNAPSHOT operation performs object-graph traversal and copying from the runtime heap to the snapshot heap. Snapshotting an object cannot be faster than a single `memcpy` system call, even if a single object is being snapshotted without any references to other objects, as GraalDoss needs to coordinate with the language runtime and maintain metadata during object-graph traversal and snapshotting.

Figure 7.2 compares the theoretical maximum throughput, i.e., `memcpy` throughput (blue), with the measured SNAPSHOT operation throughput (orange) for different memory sizes depending on the type of workload objects being processed. The difference between GraalDoss serialization throughput and the theoretical maximum throughput comes from the language-runtime coordination and the reference tracking that GraalDoss needs to perform during object-graph traversal. For smaller object graphs, the cost of language-runtime coordination required prior to storing a GraalDoss snapshot dominates the total SNAPSHOT operation times. For larger snapshot sizes, the cost of reference tracking dominates the SNAPSHOT times.

Since GraalDoss performs direct snapshotting without transforming objects, object sizes may vary depending on the underlying object layout strategy. As *GraalVM* uses chunked layouts by default, every snapshot file contains at least one heap chunk. Every chunk starts with the chunk header, which contains a card table and a first-object table, which also takes up space in the resulting snapshot. The size of the chunk header depends on the chosen card-table implementation and may even be zero. *GraalVM Serial* GC card tables add at most 2 KB to the size of the chunk header. In all experiments, reported snapshot sizes do not include the memory overhead of card tables.

The performance of the SNAPSHOT operation depends on the properties of the object-graph, such as the number of objects captured in a snapshot and the shape of the object graph itself, i.e., the number of references, presence of cycles, etc. Another factor that can impact SNAPSHOT operation performance is the runtime-heap locality of objects captured in a

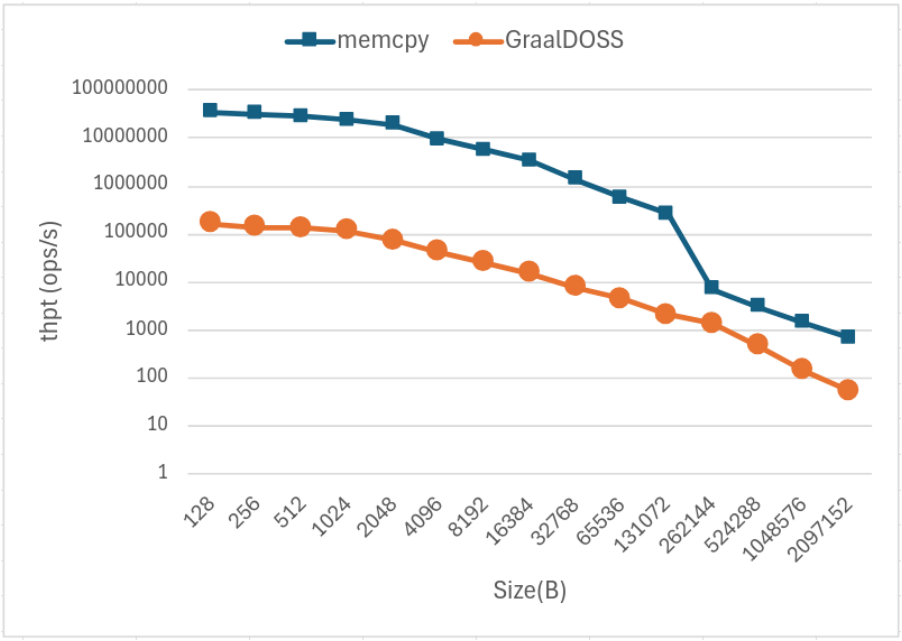


Figure 7.2: GraalDoss serialization throughput (orange, circle) compared to a single `memcpy` system call (blue, square). Throughput is presented on a log-log scale.

snapshot, i.e., whether the objects that are part of the object graph are collocated or scattered across the runtime heap.

Object-graphs used in SNAPSHOT performance experiments are byte arrays of fixed length. Using byte arrays is convenient, as the size of the object can be easily controlled through the size of the underlying byte array. The sizes of reference objects measured in Figure 7.2 (x axis) correspond to total object sizes of byte arrays of appropriate length, including the object headers. Additionally, every microbenchmark iteration performs garbage collections prior to measuring the performance of the SNAPSHOT operation, in order to further improve runtime-heap locality of the objects and reduce its effect on the obtained results.

Deserialization (LOAD) performance

GraalDoss deserialization performance is compared against the theoretical maximum — a single `mmap` system call. This is because the LOAD operation establishes memory mappings between the snapshot file persisted on disk and the target snapshot slot inside the snapshot heap. Loading an object cannot be faster than a single `mmap` system call, since LOAD operation needs to coordinate with the language runtime and read the snapshot header to determine language-runtime compatibility prior to establishing memory mappings.

Figure 7.3 shows the measured deserialization throughput of GraalDoss for different object snapshot sizes (orange), compared to a single `mmap` system call (blue). The results show that, for smaller objects, the cost of language-runtime coordination required prior to loading a GraalDoss snapshot is dominating the total deserialization time. However, as the snapshot size increases, the deserialization time remains nearly constant, decreasing slightly due to the higher cost of pre-loading checks that GraalDoss needs to perform.

Normally, for most S/D libraries, deserialization performs a linear pass over the mapped memory in order to parse, transform, or patch references of objects contained in the serialized snapshot. To simulate the performance of regular deserialization, Figure 7.3 also includes

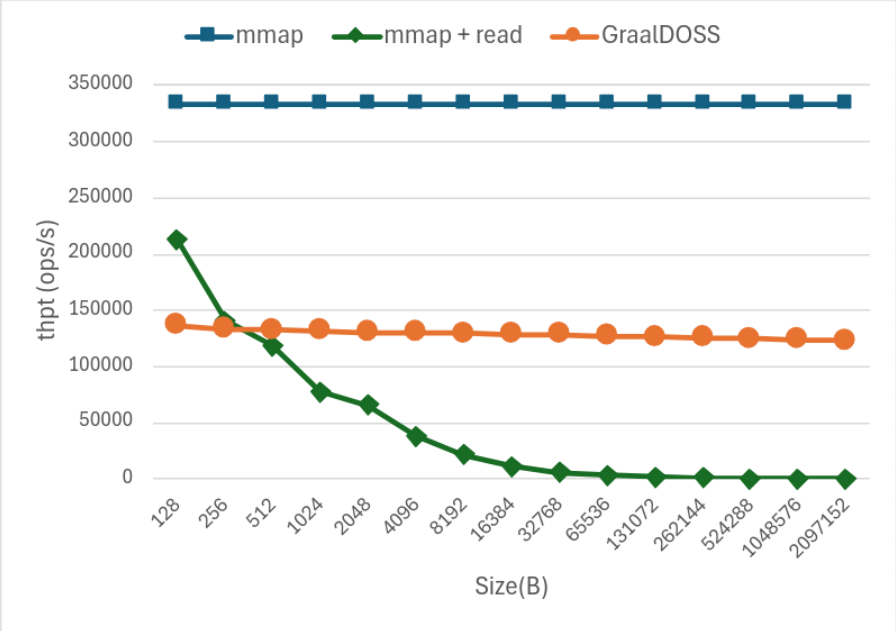


Figure 7.3: GraalDoss deserialization throughput (orange, circle) compared to a single `mmap` system call with (green, diamond) and without (blue, square) a linear pass over the mapped memory region. The throughput is presented on a linear scale.

results for a `mmap` system call with a linear pass over the mapped memory during which the memory is just read, not modified (green). Note that, in this case, the memory is not modified as it normally would be with regular deserialization. This sets a best-case benchmark time for regular deserialization. The results show that, the performance of a linear pass decreases with the size of the snapshot, contrary to GraalDoss snapshot loading. This proves that GraalDoss snapshot loading is lazy, i.e., it does not perform a linear pass over the mapped memory.

Snapshots used in the LOAD performance experiments were made of the same byte arrays used for the SNAPSHOT performance experiments. Their sizes correspond to sizes reported on Figure 7.3 (x axis), without the object header. For smaller snapshot sizes, object header dominates the total snapshot size, whereas for larger snapshot sizes the object header size is quickly overshadowed by the size of the snapshot content.

To maximize the performance of both system-call microbenchmarks, all the code for them was written in the *C* programming language, using the *POSIX* system-call interface for `memcpy`, `mmap`, and `read` system calls, as well as additional system-call utilities. The performance of the GraalDoss LOAD operation can be influenced by the language-runtime coordination that LOAD operation performs prior to loading snapshots. In order to minimize language-runtime noise, no threads other than the standard *Java* threads (such as cleaner and reference handler threads) were running, in addition to the thread that executes the LOAD operation.

Snapshot unloading (UNLOAD) performance

GraalDoss UNLOAD operation operates in a safepoint due to its integration with the *GraalVM Serial* GC. The UNLOAD operation performs a snapshot-slot scan, which can, depending on references to slot contents, cause overhead comparable to the overhead of regular garbage collections. Therefore, the overhead of the UNLOAD operation is broken down into two components — entering a safepoint and performing a snapshot-slot scan.

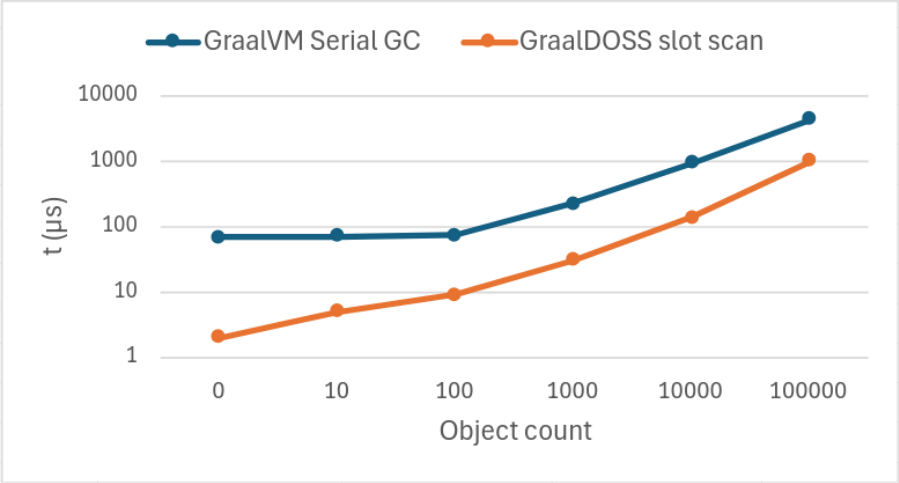


Figure 7.4: *GraalVM Serial* GC pauses (blue) compared to GraalDoss snapshot-slot scan overhead (orange) in microseconds (y-axis) for increasing amount of objects (x-axis), presented on a log-log scale.

The overhead of entering a safepoint is compared to *GraalVM Serial* GC pauses where nothing is collected. This reflects the minimal UNLOAD operation overhead, represented by the scenario where there are no objects to scan. The overhead of a snapshot-slot scan is measured in the worst-case scenario — when all objects inside the snapshot slot are referenced from the runtime heap, so the entire slot contents need to be moved to the runtime heap. The overhead of the worst-case scenario for the snapshot-slot scan is compared to the measured *GraalVM Serial* GC pauses incurred for scanning the snapshot-slot contents.

Figure 7.4 shows the results of comparing the snapshot-slot scan (orange) with the *GraalVM Serial* GC (blue), presented in a log-log scale. The results show that the snapshot-slot scan imposes less overhead compared to regular garbage collections. The minimal UNLOAD operation overhead corresponds to the scenario where there are 0 objects to collect (first data point). *GraalVM Serial* GC needs to perform special processing of writable sections of the image heap during garbage collections. Because of that, GC pauses always add a constant overhead, even when there are no objects to collect in the runtime heap. This constant overhead depends on the size of the image heap, which varies between applications[1]. GraalDoss does not need to perform any such processing, thus achieving better performance for smaller amounts of objects. As the number of objects increases, so does the total overhead of copying objects from the snapshot slot to the runtime heap. Thus, for large amounts of objects, snapshot-slot scan performance approaches *GraalVM Serial* GC performance.

7.4 Data pre-initialization evaluation

GraalDoss data pre-initialization capabilities are evaluated using the `stanfordnlp-preload` macrobenchmark contained in the S/D-*benchmarks* set. The benchmark loads a natural language-processing *StanfordNLP* [294] model for the English language, sets up the NLP pipeline, and annotates the input document. The input document types used in GraalDoss data pre-initialization experiments are presented in Table 7.2. The resulting annotated document

[1]This experiment involves a minimal *Java* benchmark without any dependencies in order to minimize the image-heap size. The size of the benchmark image is 16.7 MB, out of which the image-heap section takes 8 MB.

contains pipeline processing results, including tokenized input, part-of-speech tag lists, constituency and dependency parse data, knowledge base population relations, entity mentions and quotes, and original and canonical speakers of quotes.

Table 7.2: Annotated input document types used for GraalDoss data pre-initialization experiments, taken from the *Harry Potter and the Sorcerer's Stone* book.

Document type	Characters	Words	Sentences
tiny	329	77	5
small	983	212	15
medium	1674	368	25
large	3049	687	53

Figure 7.5 shows the `stanfordnlp-preload` execution times for different document types, broken down into *Setup pipeline* and *Annotate document* phases. The *Setup pipeline* phase consists of loading pipeline configuration and setting up the pipeline, which includes loading the *StanfordNLP* model. The *Annotate document* phase consists of input document preparation and annotation. *Other* phase represents other operations not covered by the *Setup pipeline* and *Annotate document* phases, such as final result reporting.

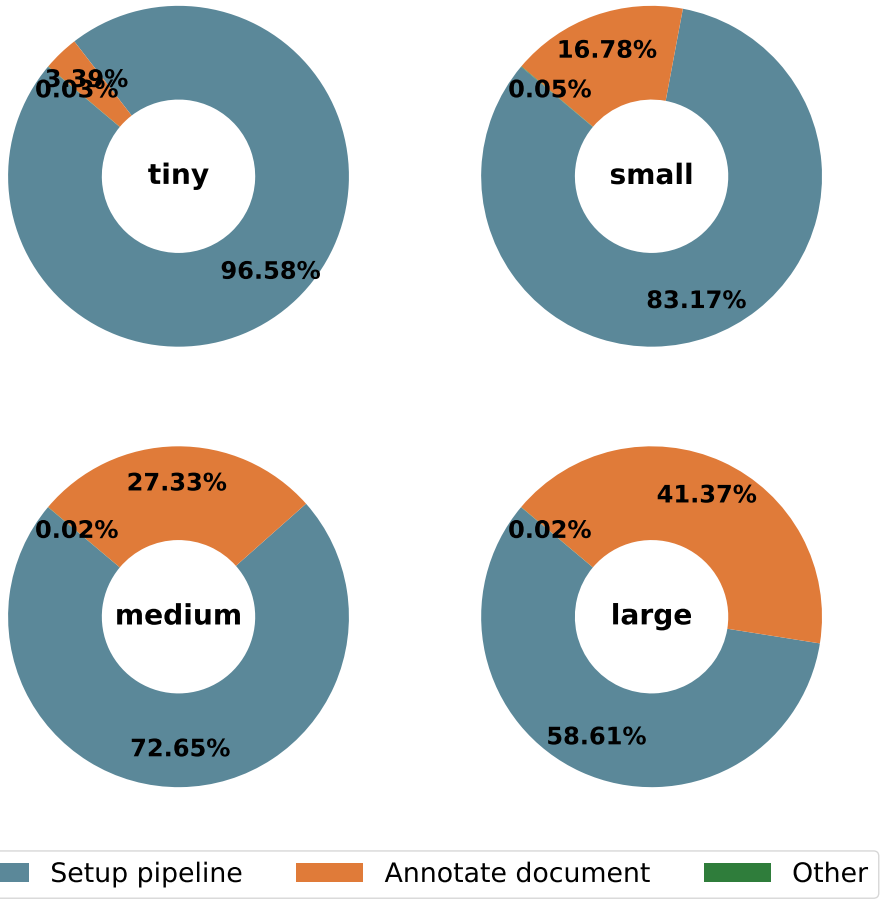


Figure 7.5: `stanfordnlp-preload` phase-execution ratios for different document types.

Table 7.3: NLP pipeline creation, document annotation, and GraalDoss operation times (in seconds) for different input document types. The final column represents the total execution time of the benchmark for both default and optimized document providers.

Document type	Setup pipeline	Annotate document	STORE document	LOAD document	Total Default	Total **GraalDoss**
tiny	34.74	1.22	4.13×10^{-3}	3.71×10^{-6}	35.97	$\mathbf{1.35 \times 10^{-4}}$
small	34.64	6.99	6.62×10^{-3}	3.83×10^{-6}	41.65	$\mathbf{1.36 \times 10^{-4}}$
medium	34.58	13.01	10.03×10^{-3}	3.99×10^{-6}	47.60	$\mathbf{1.38 \times 10^{-4}}$
large	34.92	24.65	27.91×10^{-3}	4.01×10^{-6}	59.58	$\mathbf{1.41 \times 10^{-4}}$

The *Setup pipeline* phase overhead depends on the NLP model used in the NLP pipeline. Since the same *StanfordNLP* English model is used in all experiments, the *Setup pipeline* phase takes constant amount of time, regardless of the size of the input document. For smaller input document sizes, such as *tiny* and *small*, the *Setup pipeline* phase dominates the total execution times. The *Annotate document* phase overhead depends on the size of the input document, and for larger document sizes approaches the overhead of the *Setup pipeline* phase. In general, the application can perform arbitrary operations on the NLP model output contained in the annotated document. In the case of `stanfordnlp-preload`, result reporting does not perform any complex operations, and its overhead is negligible compared to other phases.

GraalDoss can be used during an initial run of the application (cold start) to snapshot the state of the initialized pipeline, or the annotated result if the pipeline result is meant to be reusable. In subsequent runs, loading a snapshot of the warmed up pipeline or annotated result reduces the total execution time. This way, GraalDoss can eliminate the overhead of both *Setup pipeline* and *Annotate document* phases.

GraalDoss pre-initialization evaluation extends `stanfordnlp-preload` by abstracting the document annotation into annotated-document providers. Listing 7.2 shows the outline of the extended benchmark, which defines two annotated-document providers. The default provider (line 1) builds the NLP pipeline (line 12) and annotates the input document (line 18). The optimized provider (line 24) loads the annotated document from a GraalDoss snapshot via GraalDoss LOAD operation (line 26). The benchmark runs queries on resulting annotated documents for both providers (lines 4 and 27), and measures document setup, annotation, and total benchmark execution times.

The GraalDoss-enhanced benchmark is executed 10 times with a GraalDoss document provider in a cold-start mode that generates the document snapshot. During the cold-start runs, the benchmark measures combined GraalDoss SNAPSHOT and STORE times. After the cold-start runs, the benchmark is invoked 10 times using the GraalDoss provider in an optimized mode, during which the previously stored document snapshot is loaded and queried, and the LOAD operation times were recorded. All the obtained results are compared to numbers obtained by performing 10 benchmark runs using the default document provider.

Table 7.3 shows the NLP pipeline creation, document annotation, GraalDoss operation times, and total benchmark execution times for different input document types. In contrast to the default annotated-document provider, GraalDoss document provider cuts down the execution times by removing the pipeline setup and annotation phases. Loading a snapshot of the annotated document yields a performance improvement that spans several orders of magnitude. With GraalDoss, cold starts that are attributed to pipeline creation and document-annotation are removed, and document queries dominate the program's execution time.

Listing (7.2) GraalDoss integration in `stanfordnlp-preload`.

```
void benchmarkDefault() {
    String input = TextProvider.getText();
    CoreDocument doc = invokePipelineAndAnnotate(input);
    runQueries(doc);
}

CoreDocument invokePipelineAndAnnotate(String input) {
    // Setup pipeline properties
    var props = definePipelineProperties();

    // Create pipeline
    var pipeline = new StanfordCoreNLP(props);

    // Create input document
    var doc = new CoreDocument(input);

    // Annotate input document
    pipeline.annotate(doc);

    // Return annotated result
    return doc;
}

void benchmarkOptimized(Path p, int slotId) {
    DOSS.Slot slot = DOSS.heap().getSlot(slotId);
    CoreDocument doc = DOSS.provider().load(p, slot);
    runQueries(doc);
}
```

7.5 Data sharing evaluation

GraalDoss data sharing capabilities and improvements to memory footprint are evaluated using the `micronaut-cache` macrobenchmark contained in the S/D-*benchmarks* set. The benchmark consists of a microservice that serves monthly news headlines through a web API, deployed across multiple *GraalVM* memory isolates serving as isolated language VM instances. News headlines are loaded lazily from disk, with a *Micronaut* [242] in-memory cache implementation keeping loaded news headlines in memory.

Data-sharing experiments define an alternative *Micronaut* cache provider that leverages GraalDoss snapshots as a cache backend. During a cold start, the cache snapshot is not present on disk. Thus, the provider eagerly loads all news headlines, populates the cache, and creates a snapshot of the cache on disk. This snapshot is then loaded in subsequent optimized runs and shared with other cache-provider instances, in this case those created in other *GraalVM* instances.

The modified benchmark was executed 100 times separately for the default in-memory provider and the GraalDoss cache provider. An additional cold-start run was initiated prior to evaluating the GraalDoss cache provider, during which the cache snapshot was generated. An external request generator performed 1000 requests to all VM instances during each benchmark run. The benchmark recorded total process RSS, startup time, first-response time, time

to first response[2], and subsequent response times. The experiment was repeated with an increasing number of VM instances, starting from 1 memory isolate and reaching 8 concurrent VM instances.

The experiments did not uncover any regressions to the application startup times, which is expected, as GraalDoss initializes all of its metadata at image-build time. Figure 7.7 shows the observed server startup and request-response times with both the in-memory cache provider (blue) and the GraalDoss cache provider (orange). GraalDoss reduced the first-response time by 51% (10.3 ms, down from 21.2 ms) by loading the pre-populated cache snapshot. Furthermore, GraalDoss did not cause any regressions to server startup or subsequent-response times compared to the in-memory cache provider with the warmed-up cache. Overall, GraalDoss reduced the time to first response of the application by 34% (21.4 ms, down from 32.3 ms). By also taking RSS improvements into account, GraalDoss improved the overall microservice density by more than 40% for 8 application instances.

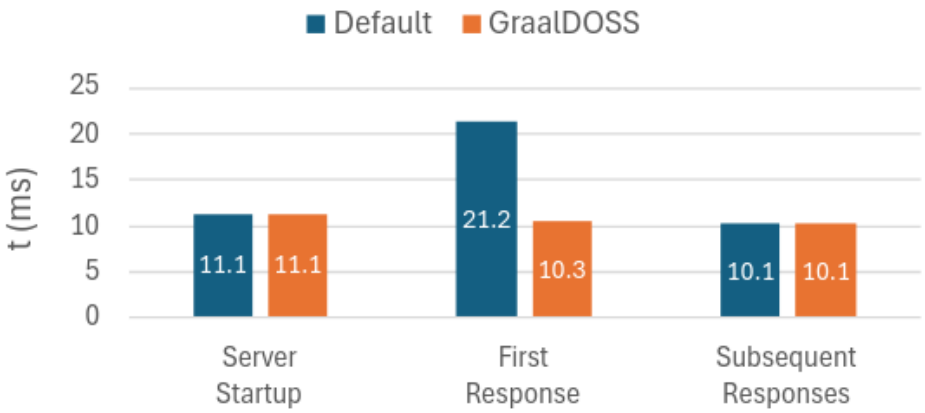


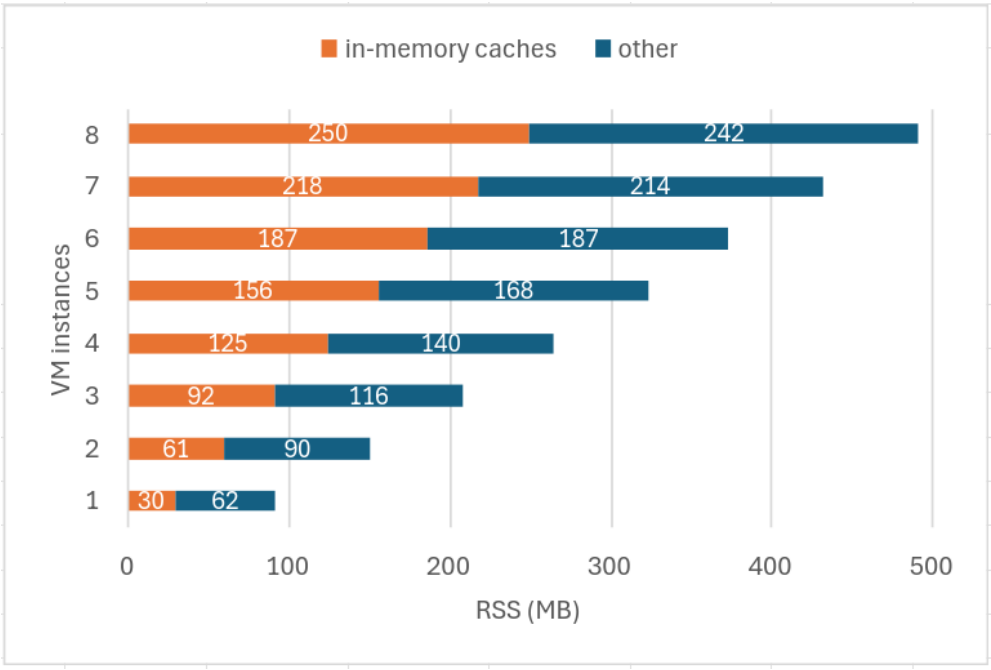
Figure 7.7: Server startup and response times (ms) for default in-memory (blue) and GraalDoss snapshot-backed (orange) cache providers.

Figure 7.8 shows the results of RSS measurements, with both the in-memory cache provider (7.8a) and GraalDoss cache provider (7.8b). The in-memory caching provider creates an additional cache instance replicated inside every VM instance. It is important to note that every VM instance also adds an observable memory overhead (blue). This overhead is quickly overshadowed by the data-cache overhead (orange), as every cache instance in these experiments adds around 30 MB to total RSS. As the number of VM instances increases, caches become a dominant factor in the application's total memory consumption. However, when using GraalDoss, the application maintains a constant data-cache overhead as the number of VM instances increases, as the cache snapshot is shared across all VM instances.

These results correspond to a scenario where VM instances only read the cache, i.e., the utilization of data sharing is maximized. Any cache modifications would result in private copies of modified memory pages for every application instance that modified the cache. In such cases, the total cache memory footprint would be increased by the total memory associated with modified cache pages in all VM instances.

[2]Time to first response includes the server startup time and the first-response time (see §2.4).

(a) In-memory cache provider.



(b) GraalDoss snapshot-backed cache provider.

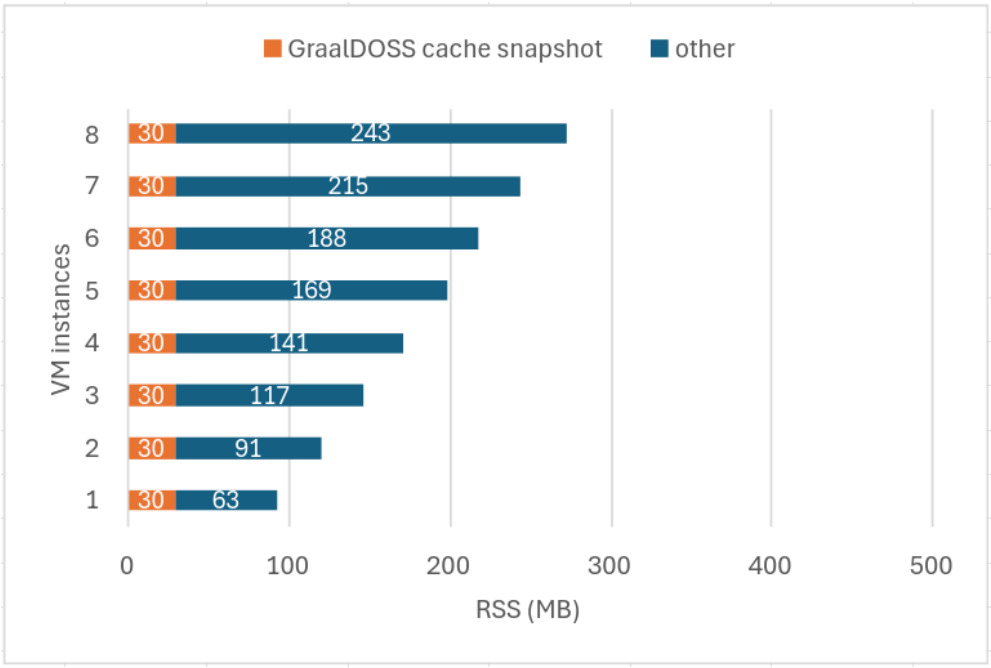


Figure 7.8: Breakdown of the total process memory consumption (RSS) as the number of VM instances (*GraalVM* memory isolates in this case) increases.

7.6 Summary

GraalDoss evaluation consists of correctness and robustness tests, microbenchmarks on a wide-range of workload objects that test the performance of GraalDoss, and macrobenchmarks that test the generality and scaling of GraalDoss by applying it to real-world data. GraalDoss is compared to other state-of-the-art C/R solutions, both qualitatively based on its features, and quantitatively based on its performance. In all cases, GraalDoss achieves performance improvements without changes in application behavior and correctness, which implicitly shows the correctness of the proposed approach and implementation.

Experimental results show that GraalDoss eliminates cold starts by restoring warmed-up data snapshots. This significantly improves the first-response times and total execution times of applications in all experiments. Microservice web-API first-response times were improved by 51%, while the NLP processing application execution times were improved by several orders of magnitude (from tens of seconds to microseconds). Restoring GraalDoss snapshots of pre-initialized application states improves the overall application memory footprint, maintaining a constant data-memory overhead in environments with multiple application instances that share data. The benefits of using GraalDoss on memory footprint increase with more application instances, depending on the cache size — in microservice in-memory-caching experiments, total process RSS was reduced by 44% for 8 application instances with a 30 MB data cache.

Chapter 8

Comparing DOSS to other C/R techniques

DOSS achieves object C/R by integrating with the executing language runtime specialized for cloud-native execution. Similar to DOSS, other C/R techniques can be used to perform efficient object C/R. This section qualitatively compares DOSS with other C/R techniques, and quantitatively compares the performance of DOSS reference implementation (GraalDOSS) against applicable state-of-the-art C/R techniques.

8.1 Qualitative comparison

C/R techniques are categorized by their scope (see §4). Therefore, qualitative comparison compares DOSS with different C/R techniques in terms of approach, applicability, architecture, performance, usability, language-runtime integration, external management requirements, and data-sharing capabilities.

System C/R techniques are orthogonal to DOSS due to the difference in granularity and overall usability. System snapshots capture the state of the entire virtualized operating system, including its working memory and processes. DOSS operates solely on the application data selected by the user and does not capture any system state. As a consequence, DOSS snapshots are significantly smaller in size compared to system-level snapshots, and operations involving such snapshots are faster.

Process C/R techniques operate in the process plane, as opposed to DOSS, which operates in the data plane, preventing any process-plane states (e.g., file handles or thread objects) from being stored as part of a snapshot. Unlike *CRIU* [86] and *Fireworks* [331], DOSS does not capture code compiled during application execution (e.g., during JIT compilations). Instead, DOSS augments systems that provide such functionalities, e.g., *GraalVM*, with data checkpoints during application execution.

Language-runtime C/R solutions, and specifically *JVM* C/R solutions such as *OpenJDK CRaC* [269], capture the entire state of the language runtime and its resident heap, whereas DOSS snapshots capture only a chosen subset of the application heap, providing more flexibility and functionality. Thus, DOSS does not need to alter GC policies or perform dynamic pruning to reduce the amount of information stored in a snapshot. Unlike other *JVM* snapshotting techniques, DOSS operates with multiple reusable and sharable snapshots simultaneously, and supports snapshot hot-reloading during application execution.

Runtime heap C/R techniques more closely resemble DOSS as they both create partial runtime-heap snapshots. Similar to *GraalVM* [387] and *V8* [372] snapshots, DOSS snapshots can be shared across multiple application instances, which is not the case for snapshots made by *Dart* [138], *Smalltalk* [161], and *Self* [369] runtimes. DOSS dynamically creates, loads, and unloads multiple snapshots simultaneously during application execution, providing a versatile C/R and sharing solution for data that evolves during the lifetime of the application process.

Object C/R introduces a performance overhead and potential data loss due to data transformations that occur during object S/D operations. DOSS simplifies the serialization phase and eliminates the deserialization phase in the S/D pipeline by creating direct snapshots of objects from the runtime heap, without any transformations of objects. Unlike most plain-text S/D techniques, DOSS snapshots retain all information contained in the original objects, and do not duplicate information. Binary S/D techniques [116, 272] aim to minimize the size of the serialized data, requiring additional CPU work during S/D operations. Direct object snapshots, combined with a native execution environment, allow DOSS to memory map snapshots during execution, and skip costly deserialization that would need to be performed in traditional environments (e.g., reference patching). Unlike hardware-accelerated S/D techniques, DOSS improves efficiency of the system without any specialized hardware.

Most relevant state-of-the-art C/R solutions that provide data serialization, transmission, and sharing that are comparable to DOSS are presented in Table 8.1. *JSON* S/D libraries [7, 146, 120, 257] perform significant object transformation due to the serialization format that limits efficient data sharing, but they do not require language-runtime integration and external management. Binary S/D libraries such as Kryo [116] use a more compact but *JVM*-specific binary format that supports reference tracking and reference cycles. Compared to *JSON* and binary S/D libraries, DOSS provides near-constant deserialization and data-sharing capabilities that significantly improve the overall performance and memory footprint of the system. *Skyway* [258] improves S/D overhead by transferring direct object graphs, but still requires reference patching on deserialization. *Naos* [356] and *RMMap* [231] eliminate S/D phases by leveraging *RDMA* networks. Compared to *Skyway*, *Naos*, and *RMMap*, DOSS performs no object transformation and requires no specialized hardware. *ZCOT* [391] introduces an external distributed and shared memory abstraction called the *transfer space*, with similar features to the DOSS snapshot heap. However, *ZCOT* transfer space does not provide data-sharing capabilities other than data deduplication during transfers. DOSS and *ZCOT* operate in different environments, as DOSS is tailored for cloud-native deployments on a single host, rather than a distributed big-data environment that *ZCOT* is designed for. Thus, unlike *ZCOT*, DOSS does not require an external service to manage the snapshot heap.

8.2 Quantitative comparison

Quantitative comparison measures the performance of GraalDOSS, the reference DOSS implementation, against relevant C/R techniques. GraalDOSS is built on top of *GraalVM*, which fulfils all of the requirements of DOSS. However, not all C/R solutions are applicable to cloud-native environments, most commonly because they require dynamic *JVM* features or heavy modifications to the language-runtime itself, which do not fit the AOT compilation strategy. Thus, GraalDOSS is compared against *Java* object S/D libraries that can be applied to the cloud-native setting, including plain-text S/D, and binary S/D libraries.

Table 8.1: Comparisons between most relevant state-of-the-art C/R solutions and DOSS.

Solution	JSON S/D	java Kryo	Skyway	Naos RMMap	ZCOT	**Doss**
S/D format	JSON	Binary	Binary	Binary	Binary	**Binary**
Language-runtime integration	No	No	Yes	Yes	Yes	**Yes**
Reference tracking	No	Yes	Yes	Yes	Yes	**Yes**
Object transformation	Yes	Yes	Partial	No	No	**No**
External management	None	None	None	RDMA networks	Metadata Server	**None**
Data loss	Yes	No	No	No	No	**No**
Data sharing	No	No	No	No	Partial	**Yes**
RSS improvements	Low	Low	Low	Low	Only for big-data	**High**

To that extent, quantitative GraalDOSS evaluation against other S/D libraries is performed using the *sd-jmh-native* microbenchmarks contained in the S/D-*benchmarks* set. *Java* S/D library versions used in the experiments can be found in Table 8.2. To ensure fairness in all modes when evaluating against different S/D libraries, *sd-jmh-native*:

i) Uses multiple warm-up runs and timed runs,

ii) Measures all tested libraries against the same workload set,

iii) Avoids library caches and S/D short-circuiting by randomizing the workload objects; that way, it obtains results that do not favor implementations that optimize for repeated workloads, and

iv) Avoids I/O operations in plain-text and binary S/D measurements by using only in-memory S/D (generated objects are serialized into in-memory byte arrays or streams backed by in-memory byte arrays and deserialized back into original heap objects).

Table 8.2: *Java* object S/D libraries compared against GraalDOSS.

Library	Developed by	Version	Release date
jackson-databind [120]	*FasterXML LLC*	`2.17.0`	Mar 2024
gson [146]	*Google LLC*	`2.11.0`	May 2024
dsljson [257]	*New Generation Software Ltd*	`2.0.2`	Aug 2023
fastjson2 [7]	*Alibaba Group*	`2.0.31`	May 2023
Kryo [116]	*EsotericSoftware*	`5.6.0`	Jan 2024

To provide a fair and extensive comparison, the experiments are performed both with default and modified settings for every S/D library. Modified settings are used in situations where reference tracking is supported (e.g., *Kryo*) to reduce the number of redundant copies when a cycle is encountered in the object graph. As GraalDoss tracks references during serialization, its performance is compared with other libraries separately for both reference tracking settings.

The experiments measure the sizes of resulting S/D snapshots of the same workload objects for all S/D libraries in a separate, non-timed benchmark run. S/D performance measurements were performed both for serialization and deserialization operations separately, with a warm-up phase consisting of 5s and a benchmark phase consisting of 10s for each workload object. The results are summarized below on a subset of representative workloads. Comprehensive results for all workloads can be found in Appendix A.

S/D snapshot sizes. Figure 8.1 shows the observed serialized data sizes for records and arrays of various sizes, while Figure 8.2 shows the observed serialization data sizes for list and map workloads. In most cases, GraalDoss snapshot sizes are higher compared to *JSON* or other binary formats, except for smaller objects such as records or simple data classes, where GraalDoss snapshots are smaller compared to *JSON*. The differences in sizes come from the fact that GraalDoss stores objects directly from the language-runtime heap, and does not compact or compress serialized data.

S/D performance. S/D throughput is measured and reported separately for each workload object type. This section summarizes the S/D results for the most representative workloads, such as records with `double` fields (Figure 8.3 and Figure 8.4), `double` arrays (Figure 8.5 and Figure 8.6), square matrices with `double` elements (Figure 8.7 and Figure 8.8), `Client` lists (Figure 8.9 and Figure 8.10), and hash maps from integers to `Clients` (Figure 8.11 and Figure 8.12).

Results show that GraalDoss serialization throughput is comparable to other S/D libraries. On average, *Kryo* with disabled reference tracking gave the highest serialization throughput. GraalDoss achieved similar serialization throughput as *Kryo* with reference tracking for larger workloads, such as lists and maps. The linear overhead of serialization for GraalDoss comes from the combination of language-runtime coordination and object-graph traversal and copying during snapshotting. Even though GraalDoss performs no object transformation, it still needs to copy the object memory, which is often cheaper than regular serialization but still needs to be performed for every object. For smaller object-graphs, the cost of language-runtime initialization dominates the serialization times. As the size of object graphs increases, the language-runtime initialization overhead diminishes and GraalDoss serialization performance approaches the serialization performance of other libraries.

Results also show that GraalDoss deserialization throughput is the greatest among all tested libraries. For smaller objects, such as records, the overhead of GraalDoss language-runtime coordination dominates the deserialization throughput and as a result GraalDoss throughput is worse than most other S/D libraries (Figure 8.4). However, for larger workloads, the linear deserialization overhead of other libraries starts to manifest. GraalDoss maintains consistent deserialization throughput, several orders of magnitude higher than *Kryo*, the second-fastest S/D library. GraalDoss might establish multiple memory mappings depending on the size of the snapshot and the object layout, which is reflected in the results for larger workload objects (Figure 8.10 and Figure 8.12).

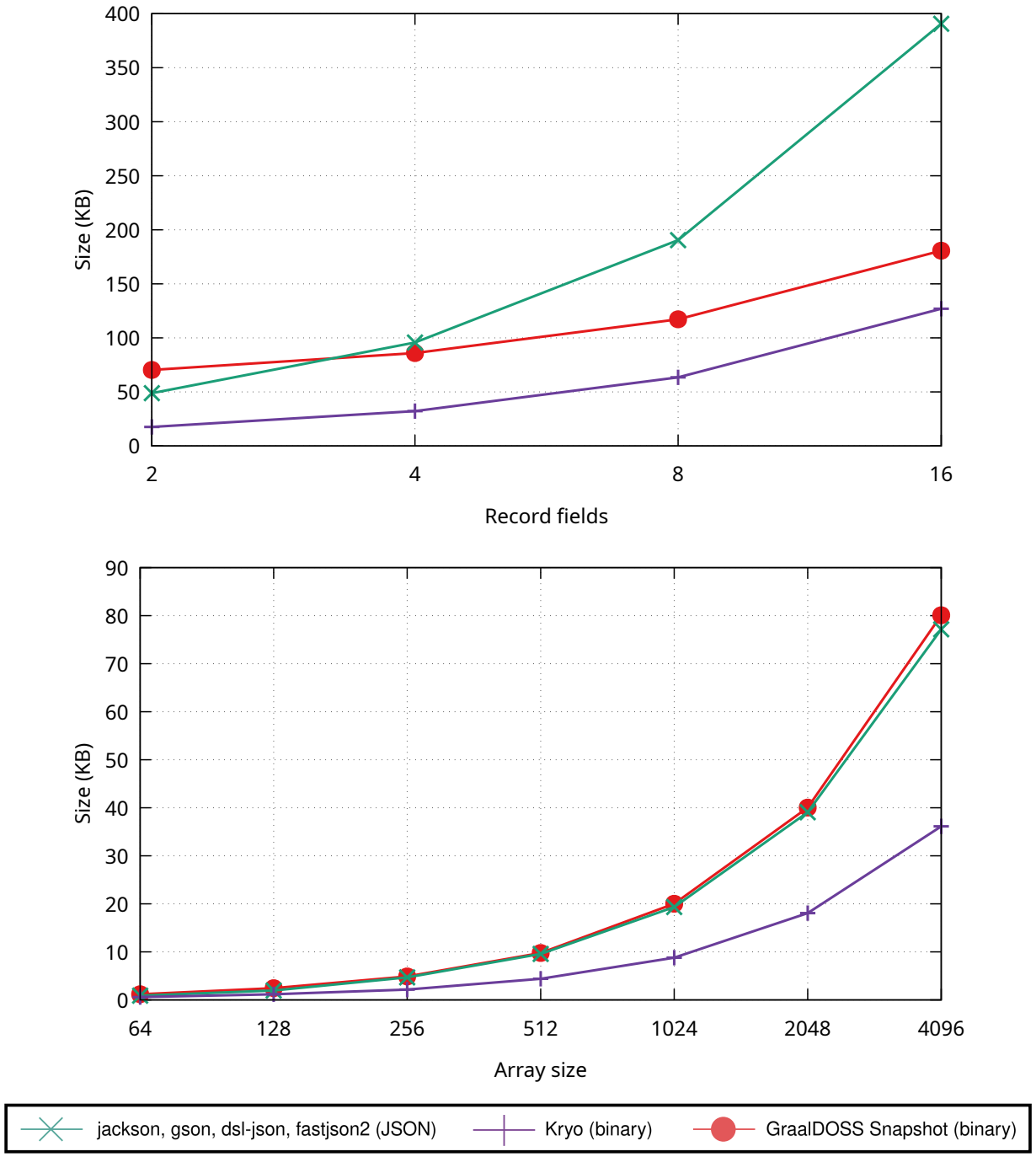


Figure 8.1: Summarized microbenchmark results for snapshot sizes in KB for records with `double` fields (up) and `double` arrays (down).

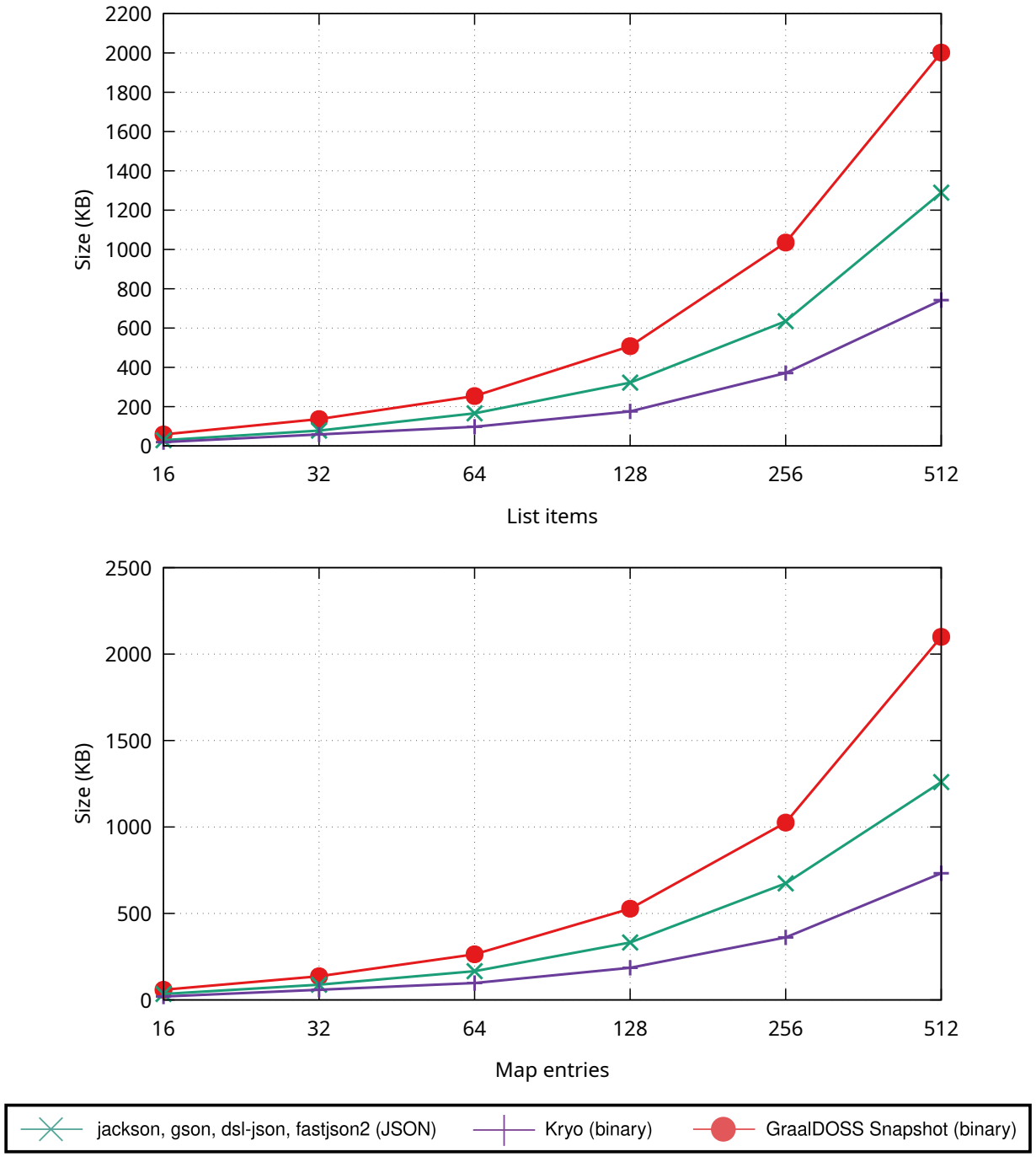


Figure 8.2: Summarized microbenchmark results for snapshot sizes in KB for `Client` lists (up), and hash maps from integers to `Clients` (down).

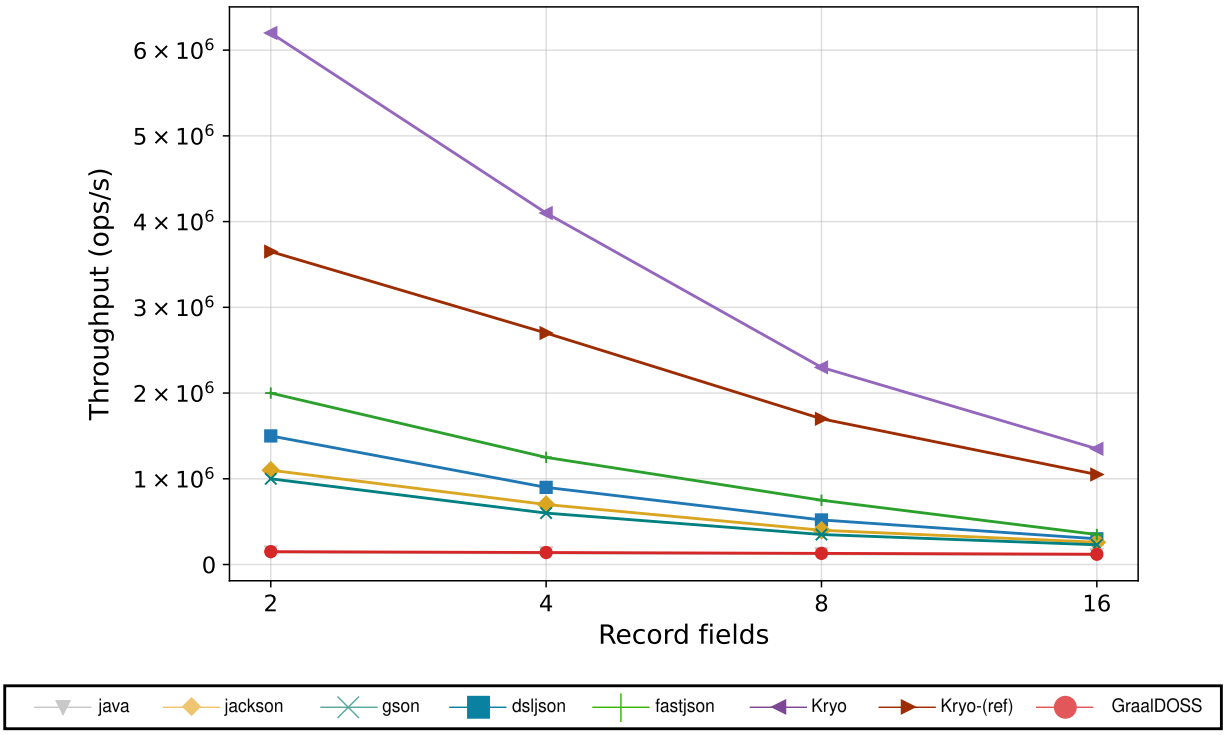


Figure 8.3: Serialization throughput for records with `double` fields measured in S/D operations per second and presented in linear scale.

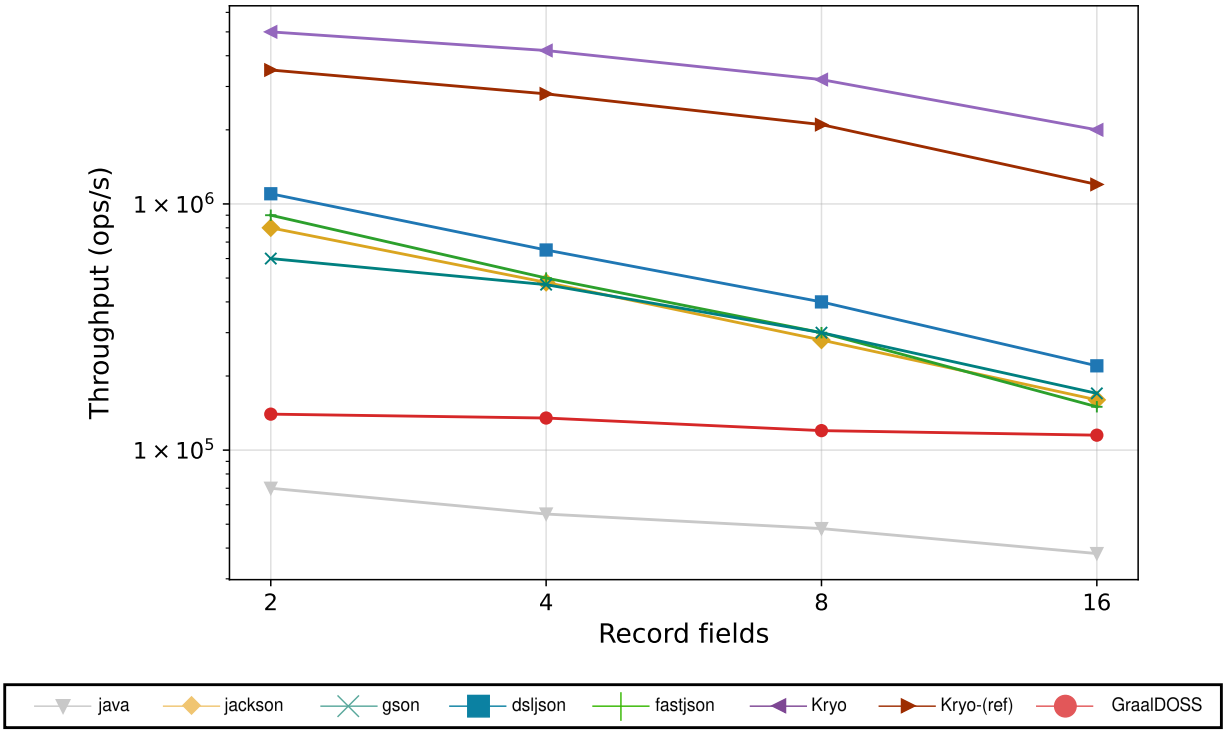


Figure 8.4: Deserialization throughput measured for records with `double` fields in S/D operations per second and presented in logarithmic scale.

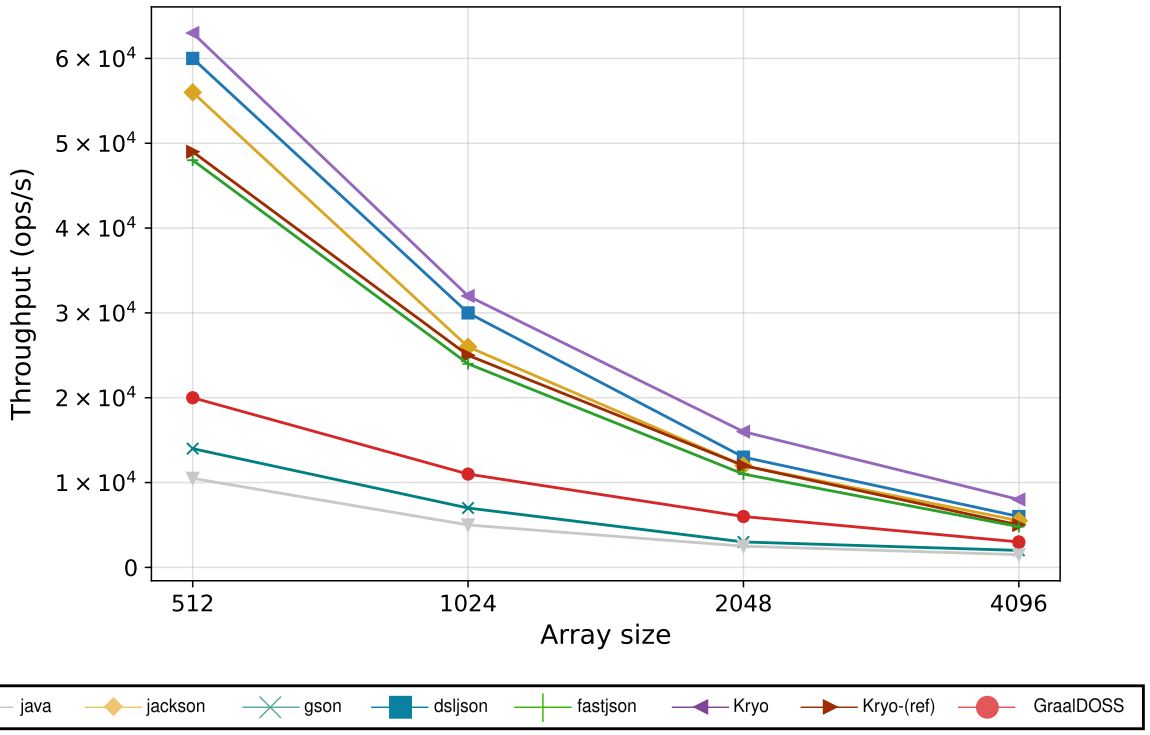


Figure 8.5: Serialization throughput for `double` arrays measured in S/D operations per second and presented in linear scale.

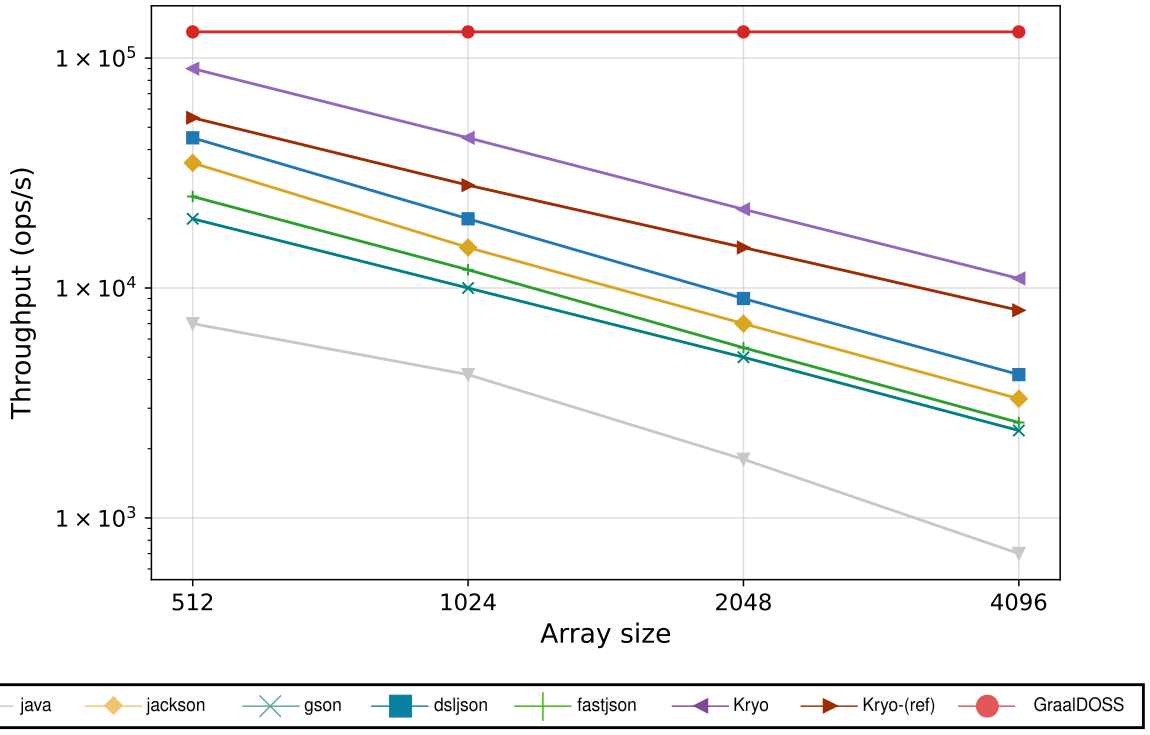


Figure 8.6: Deserialization throughput for `double` arrays measured in S/D operations per second and presented in logarithmic scale.

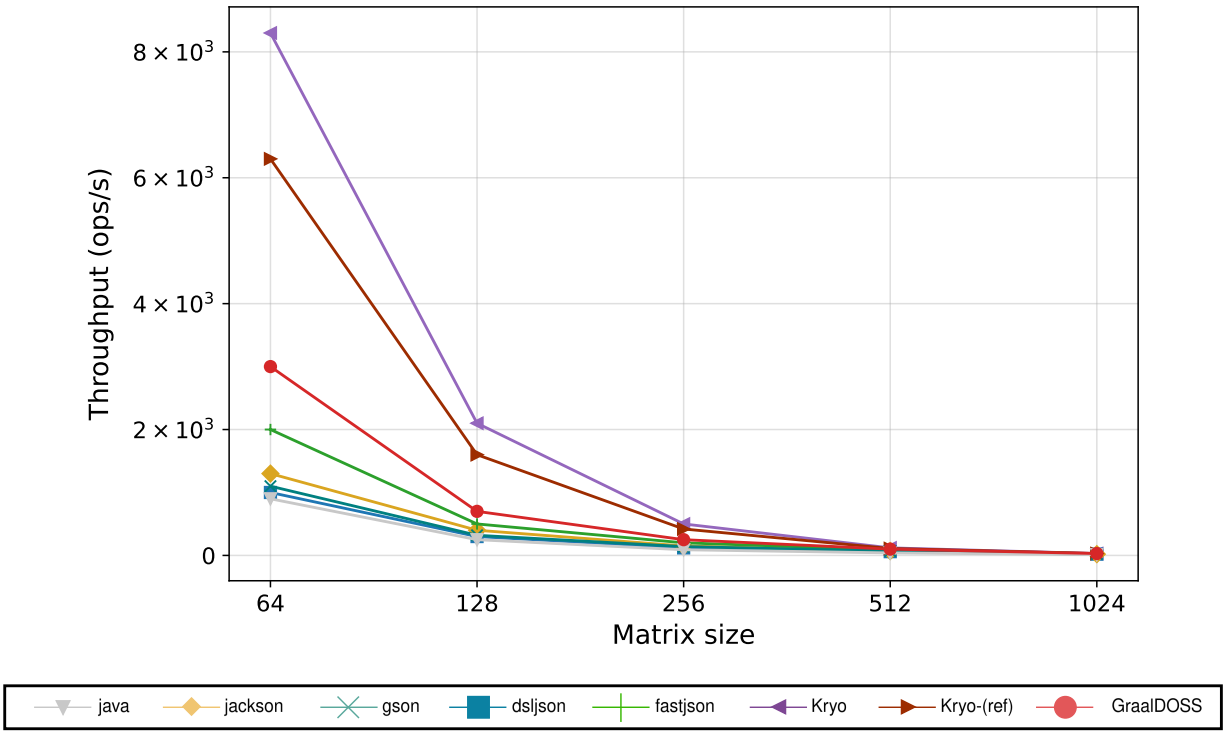


Figure 8.7: Serialization throughput for square matrices with `double` elements measured in S/D operations per second and presented in linear scale.

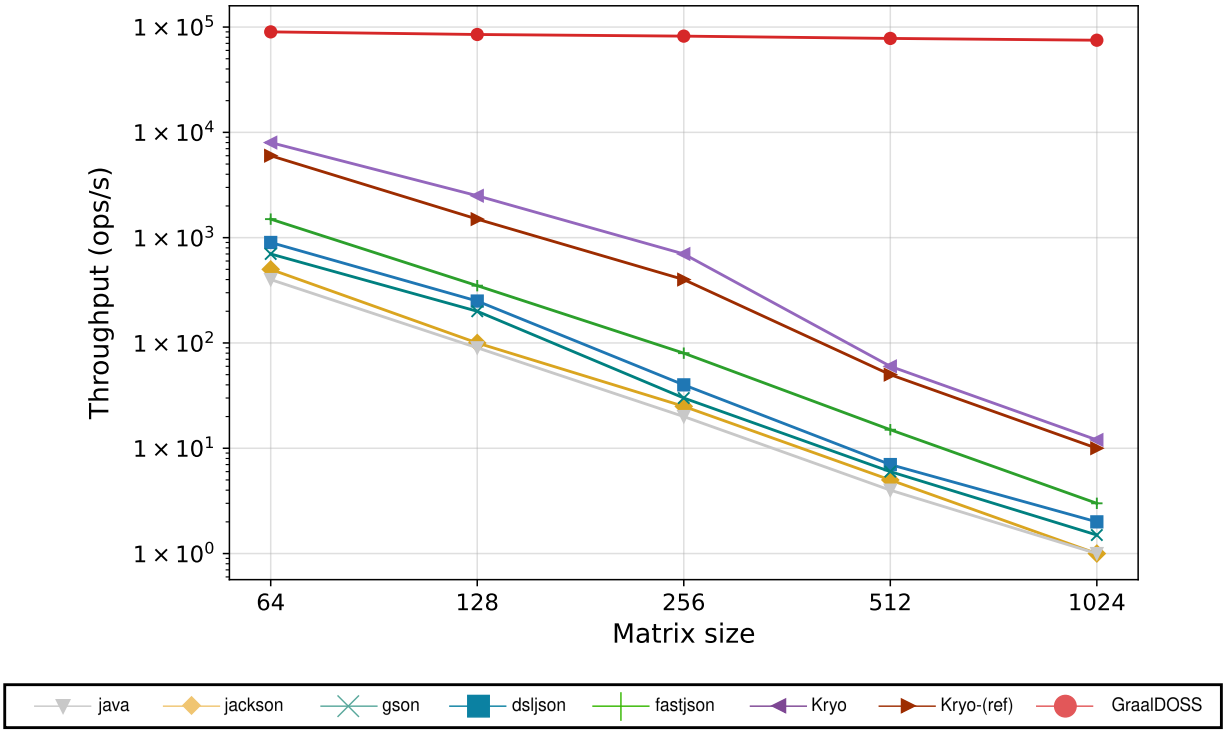


Figure 8.8: Deserialization throughput for square matrices with `double` elements measured in S/D operations per second and presented in logarithmic scale.

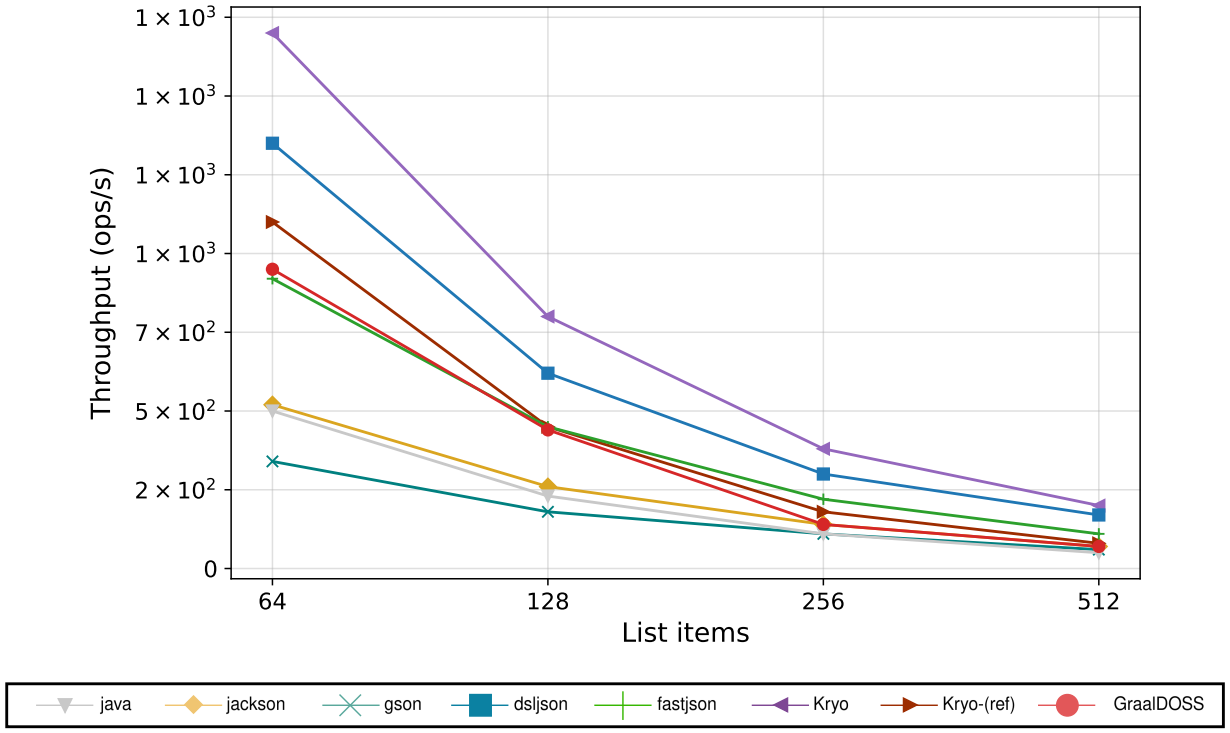


Figure 8.9: Serialization throughput for `Client` lists measured in S/D operations per second and presented in linear scale.

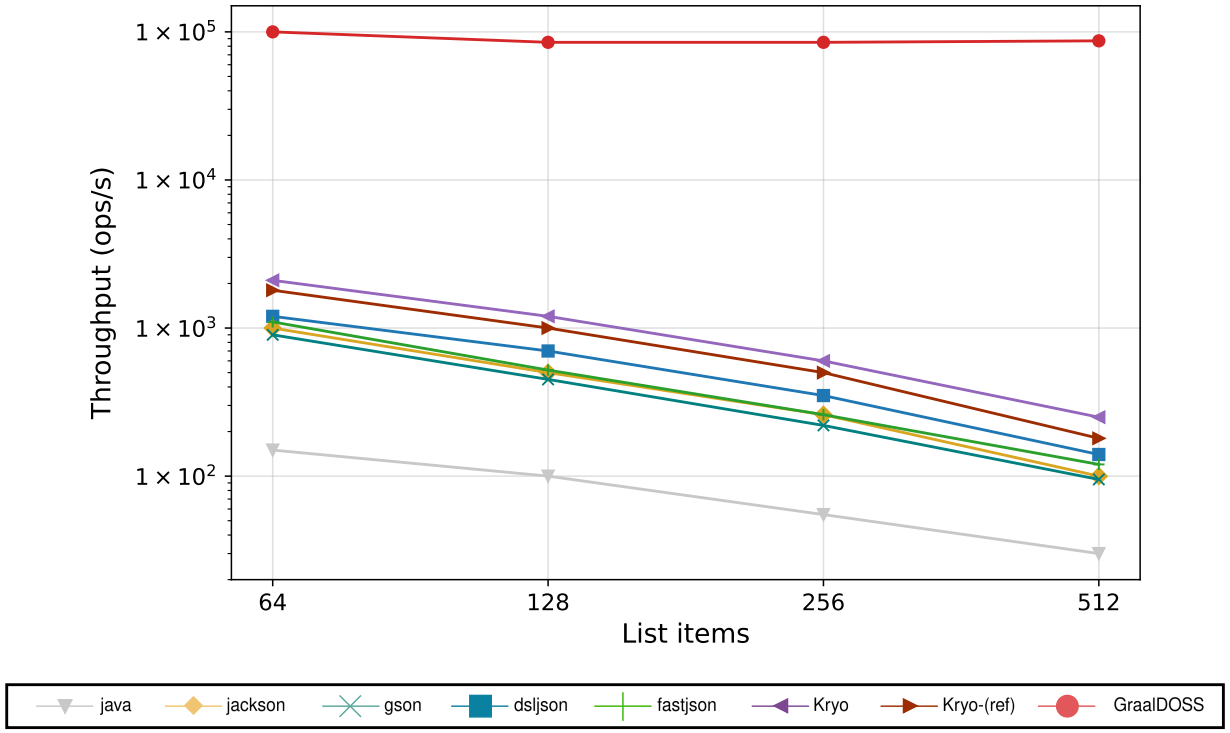


Figure 8.10: Deserialization throughput for `Client` lists measured in S/D operations per second and presented in logarithmic scale.

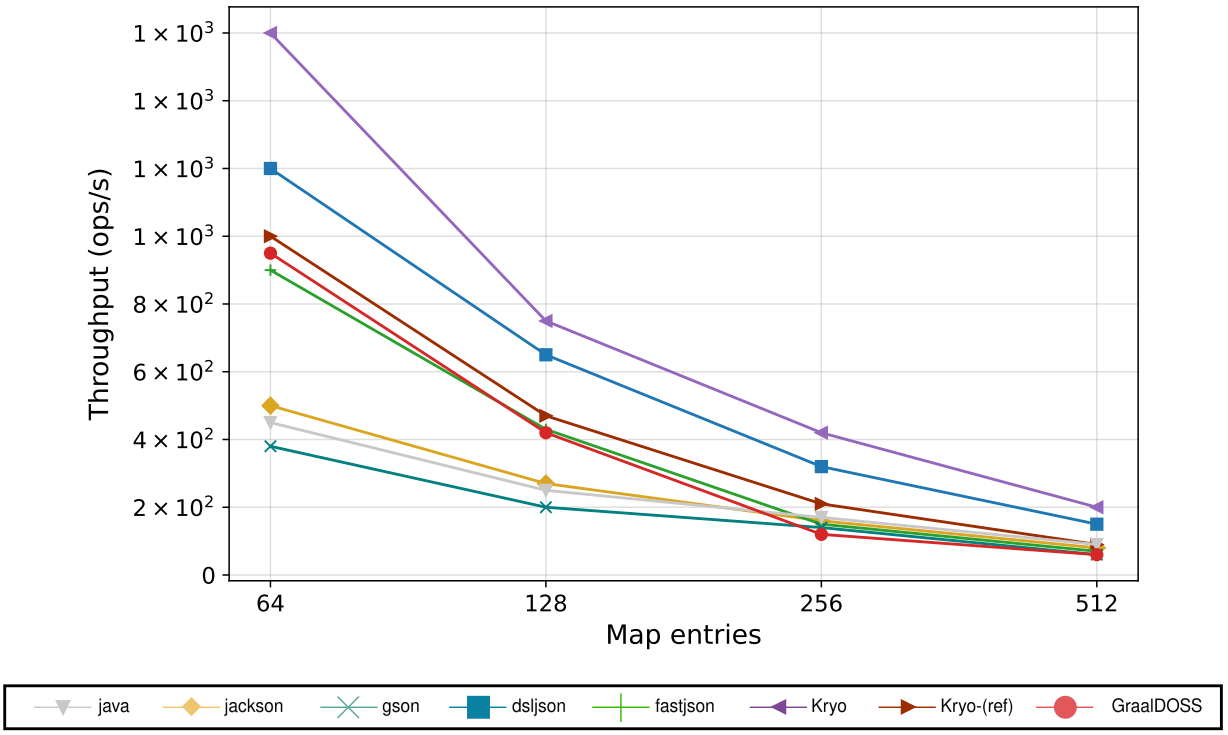


Figure 8.11: Serialization throughput for hash maps from integers to `Clients` measured in S/D operations per second and presented in linear scale.

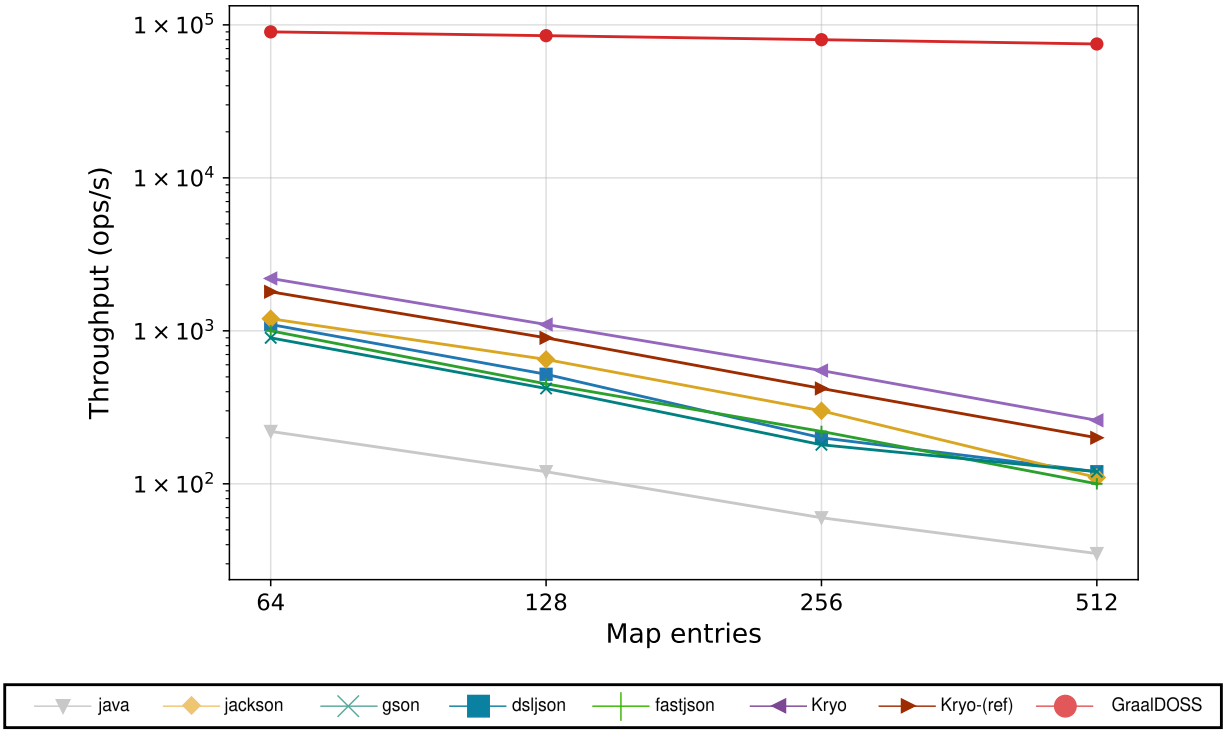


Figure 8.12: Deserialization throughput for hash maps from integers to `Clients` measured in S/D operations per second and presented in logarithmic scale.

Chapter 9

Discussion

This chapter discusses the proposed system for direct data snapshotting and sharing (§5), its reference implementation (§6), and the results obtained in the evaluation (§7), in the context of threats to validity (§9.1), security implications (§9.2), and the integration of the proposed solution in latency-sensitive environments (§9.3).

9.1 Threats to validity

Experiment-based research introduces various validity threats, both internal and external [390]. Internal threats originate within the system itself, often as a result of setup limitations or noise impacting the measurements. External threats refer to the generalization of the evaluated system to real-world environments and can affect the relevance of the obtained experimental results in a broader context.

Internal threats are remedied with a rigorous experimental setup (see §7.2). The experiments were conducted in isolation, ensuring no programs other than the operating system processes were running during benchmarking. Multiple isolated runs and noise-reduction techniques, such as disabling *Turbo Boost* and employing *CPU binding*, eliminate external factors that might influence the results and reduce benchmark stability. All the reported numbers were obtained by invoking the benchmarks in multiple warm-up and timed benchmark runs, minimizing the deviation between observed results. All performed experiments are reproducible, as the source code of the benchmark set and all the accompanying data are publicly available [307, 310]. The artifact containing the modified *GraalVM* augmented with GraalDoss is available on demand.

External threats are remedied with extensive testing and microbenchmark sets that thoroughly evaluate the correctness and performance of the system on a wide range of workloads. With that, it is ensured that the implementation is correct and the results are unbiased. Next, the macrobenchmark set shows that the implementation works as part of a larger system, expanding onto the microbenchmark workloads with data commonly used in microservice applications. Scaling GraalDoss from microbenchmarks to macrobenchmarks yielded expected results, confirming the relevance of results obtained from microbenchmarks and supporting their generalization in the broader context.

9.2 Security implications

In recent years, cloud-computing security has become an increasing priority due to the increase in demand, popularity, and wide attack surface [319, 64, 338, 186, 319]. In particular, C/R techniques, including object S/D, present a unique attack vector that is often exploitable. If not designed properly, object S/D can be the cause of a wide range of security issues [147, 93], including API insecurity, application crashes, data leakage, remote code execution, and denial-of-service attacks.

GraalDoss is designed to integrate into cloud-native applications, as an augmentation to language-runtime capabilities, or as an extension to microservice frameworks. The intended use of GraalDoss is internal, with its API restricted to the platform or framework, and not the end user. GraalDoss assumes that the execution domain is trusted, i.e., that it is running in an environment governed by trusted entities. This assumption often holds for environments that use *GraalVM* memory isolates as part of their virtualization stack [168]. GraalDoss inherits object S/D security issues, including API insecurity, data leakage, and denial of service, in addition to threats associated with its own design and implementation, such as external snapshot-heap access and external snapshot modification.

API insecurity involves vulnerabilities in the API of the system, allowing an attacker to misuse the API to gain access to or overwrite private data. API insecurity risks often come from unsanitized API parameters, whose values are mistreated by the system to cause various vulnerabilities. GraalDoss minimizes API security risks by highly constraining its API. The exposed interface is minimal, equivalent to the interfaces of other S/D libraries. The platform that integrates GraalDoss can choose to further constrain the API to prevent direct access to file paths and object graphs, as shown in Listing 9.1 via a sample microservice web API controller. In this conceptualization, the framework limits access to direct file paths and the objects contained in the snapshot by allowing users to only annotate methods and method parameters that should be snapshotted or loaded.

Denial of service risks are associated with application delays or crashes, which can impact overall service availability in the cloud. GraalDoss is an extension to the executing language runtime and its garbage collector, and GraalDoss operations are considered core routines that have privileged access to the language-runtime and garbage-collector metadata. As such, the correctness and robustness of those operations need to be thoroughly tested to ensure that GraalDoss does not block execution or cause application crashes. Furthermore, GraalDoss increases the risk of accessing invalid pointers through loaded snapshots or faulty snapshot-heap memory management. However, denial of service risks are remedied through testing of GraalDoss using a wide range of correctness and performance tests, which prove that GraalDoss operates without failures, and that its operations incur pauses that are smaller than garbage-collector pauses.

Data leakage risks are associated with privileged data being accessed or captured inside GraalDoss snapshots. Privileged data may include private or system information, and can implicate the application or language-runtime instance that created the GraalDoss snapshot. By design, GraalDoss snapshots do not capture any additional objects other than those reachable from the requested snapshot-root object. Furthermore, GraalDoss employs active measures to prevent snapshotting execution-specific states. In addition to objects that constitute the

Listing (9.1) A conceptualization of a microservice web API controller that uses GraalDoss through a constrained API. In this example, `POST` updates to a `Repository` object are persisted as snapshots, which are then dynamically loaded and shared by the framework across application instances during `GET` requests.

```
@Post
@StoreResultAsSnapshot
public Repository update(
    @LoadFromSnapshot Repository repository,
    Person person
) {
    repository.update(person);
    return repository;
}

@Get("/{id}")
@Returns(Person.class)
public Person findById(
    @LoadFromSnapshot Repository repository,
    int id
) {
    return repository.findPerson(id);
}
```

snapshot, GraalDoss also includes a snapshot header in the snapshot file, which contains GraalDoss metadata needed to safely load the snapshot. Snapshot header metadata contains minimal information about the language-runtime instance that created the snapshot — its unique identifier (which can be calculated arbitrarily) and image-heap-base offset of the slot where the snapshot needs to be loaded. The language-runtime identifier is computed from the executing language-runtime instance, but cannot be reverse-engineered to retrieve details such as language-runtime metadata or settings. Thus, a malicious party can only know whether multiple snapshots were created by the same language-runtime instance, but not deduce any details about that instance, e.g., VM or GC version, metadata, and configuration.

External snapshot-heap access risks correspond to the possibility of external access and modification of objects in the snapshot heap during application execution. These risks are increased because GraalDoss snapshot heap position is known in advance — snapshot heap is placed between the image heap and the runtime heap. GraalDoss allows the user to increase the offset of the snapshot heap and to some extent obfuscate that information, but the offset can always be deduced during program execution. For example, a malicious party could perform the SNAPSHOT operation to a particular snapshot slot and deduce the base address of the slot through the reference to the snapshot-root object. Then, an attacker can deduce the snapshot-heap base pointer, and access snapshot-heap metadata. GraalDoss does not perform any access protections on the snapshot-heap as the domain in which GraalDoss operates is assumed to be fully trusted. However, GraalDoss does protect the entire snapshot heap memory range to prevent remote code execution. Further isolation of individual snapshot slots can be enforced using advances in modern hardware, such as memory-protection keys (abbr. *MPK*) [237]. Such solutions come with a performance penalty, and are often deployed in security domains where ultimate trust is not guaranteed.

External snapshot modification involves modifying the persisted GraalDoss snapshot externally prior to its loading, effectively replacing or injecting data into the language-runtime instance that loads the snapshot. Modifications to the snapshot can include changes to references to access off-heap memory or objects in other snapshot slots, deduced based on the slot offset stored in the snapshot header. GraalDoss prevents loading of incompatible snapshots through metadata stored in the snapshot header. GraalDoss assumes that the header has not been tampered, as no mechanism can verify that no tampering has occurred, without introducing additional complexity to S/D operations.[1] Similarly, a verification phase during snapshot loading would introduce a significant overhead that scales linearly with the size of the snapshot.

9.3 Latency-sensitive environments

Latency-sensitive environments include real-time systems and applications that require minimal and stable latency [178, 126, 124, 68]. An example of a latency-sensitive application might be a software that governs autonomous vehicles, where the on-board computer has a very limited time window to recognize and adapt to changes in the environment. In such systems, unexpected changes in pauses for stop-the-world operations such as garbage collections are unacceptable. Hence, latency-sensitive environments often use low-latency garbage collectors such as *ZGC* [400].

GraalDoss operations are performed in safepoints that pause all executing threads and, as such, could trigger latency spikes that can impact latency-sensitive environments. The overhead of entering a safepoint was measured to take a few milliseconds, depending on the number of executing threads. GraalDoss enters safepoints during SNAPSHOT, LOAD and UNLOAD operations. Their overhead is added on top of the overhead of entering a safepoint.

SNAPSHOT. Snapshotting an object graph can take an arbitrary amount of time, depending on the amount and size of the object graph being snapshotted. GraalDoss evaluation showed that SNAPSHOT operation times were comparable to S/D times of other libraries, exceeding 1 millisecond only for very large object graphs. In the limit, GraalDoss SNAPSHOT operation performance is an order of magnitude slower compared to `memcpy` for the same amount of memory. Depending on the size of objects that need to be snapshotted and the frequency of creating snapshots during application execution, it might not be acceptable to repeatedly snapshot objects in latency-sensitive environments.

LOAD. Loading an object consists of memory mappings that add constant and predictable overhead. The LOAD operation could operate without entering a safepoint, but then it would have to pause garbage collections that could occur during loading until the loading is over. GraalDoss enters a safepoint during loading to simplify the GC integration and to prevent possible inconsistent reads of snapshot-heap metadata. The LOAD operation, on average, operates in 1-2 microseconds, with the overhead increasing in cases when multiple memory-mappings need to be established, in which case it takes 3-4 microseconds on average. LOAD operation is around three times slower compared to a single `mmap` call.

[1] Introducing, for example, a checksum of the header does not solve the issue if the attacker can modify the snapshot, as the checksum would need to be stored in the snapshot itself. Encrypting the snapshot header using a key derived from the language-runtime identifier is possible, but requires additional work during S/D operations, which could severely impact performance.

UNLOAD. Unloading an object can trigger a potentially expensive snapshot-slot scan, which is a modified form of garbage collection that moves objects from the snapshot slot to the runtime heap. Evaluation of the snapshot-slot scan overhead showed that regular garbage-collection pauses introduce greater overhead compared to the snapshot-slot scan. At the time of writing, *GraalVM* support for *ZGC* is under development, hence GraalDoss does not implement such support, and the evaluation covered only *GraalVM Serial* GC. However, the architecture of GraalDoss allows it to integrate with arbitrary garbage collectors in the future, including *G1* GC [97] and *ZGC*.

Chapter 10

Conclusions and future work

Direct object snapshotting and sharing (DOSS) system checkpoints object graphs from the runtime heap without transformations during application execution, persists them on disk in the form of reusable snapshot files, and restores and shares persisted snapshots across application instances in subsequent executions. GraalDOSS, the reference implementation of DOSS on top of *GraalVM*, can be used to pre-initialize, share, version-control, and hot-reload data on demand without rebuilding or restarting the application.

10.1 Scientific contributions

DOSS is a novel C/R mechanism that operates on direct object snapshots, i.e., transitive closures of all objects reachable from the selected root object in the runtime heap. Contrary to traditional S/D solutions, DOSS does not use a custom plain-text or binary format inside snapshots. Instead, it performs C/R without any transformations of objects. Normally, such a snapshot would be invalid in subsequent application executions, as it contains object references that are valid only in the address space of the process that created the snapshot. DOSS integrates with the executing language runtime, leveraging relative object references to support loading of previously persisted direct snapshots in subsequent executions. Upon loading a snapshot, DOSS only establishes memory mappings, without a linear pass over the loaded snapshot contents that is normally needed for reference patching. This allows DOSS to eliminate the deserialization overhead when loading snapshots.

DOSS architecture introduces the snapshot heap — an auxiliary managed heap separated from the runtime heap. The snapshot heap is divided into snapshot slots, each capable of hosting a single DOSS snapshot. The architecture of the snapshot heap allows its slots to be shared across multiple application instances, leveraging copy-on-write memory mappings. DOSS dynamically manages memory for the snapshot heap during application execution, allocating and releasing memory for individual snapshot slots.

Objects inside snapshots are laid out using several object laying-out strategies that minimize snapshot size and memory fragmentation. DOSS supports linear and chunked object laying-out strategies, depending on the object layout of the runtime heap. In the linear strategy, objects are laid out sequentially in access order, whereas in the chunked strategy, objects are grouped into aligned chunks that contain multiple smaller objects and unaligned chunks that each hold a single large object. DOSS integrates with the garbage collector to provide seamless compatibility between objects in the runtime heap and objects in the snapshot heap.

Fine-grained snapshot-heap architecture allows DOSS to operate with multiple snapshots

simultaneously during application execution. Snapshots can be versioned, loaded, and unloaded on demand, allowing the application to support hot reloading of objects during execution. Normally, replacing application configuration or embedded pre-initialized data requires the application to be restarted or, in some cases, rebuilt with a new set of options. DOSS allows the application to seamlessly replace objects during execution, applying data updates or rollbacks without downtimes.

DOSS is developed as a C/R mechanism tailored for cloud-native architectures, with a set of requirements that make it applicable to arbitrary language runtimes. GraalDOSS represents an end-to-end solution that integrates DOSS into *Oracle GraalVM 24.1*. It augments the *GraalVM* language runtime with an extensible and scalable snapshot heap, as well as operations that allow the runtime to SNAPSHOT, STORE, LOAD, and UNLOAD objects during application execution.

GraalDOSS evaluation is performed on a novel benchmark set [307, 310] that consists of C/R unit tests, object S/D performance microbenchmarks, and web microservice and big-data macrobenchmarks. The benchmark set is extensible in terms of C/R framework support and the workload set used for benchmarking. All benchmarks support both regular *JVM* and cloud-native deployment modes.

GraalDOSS eliminates cold starts by efficiently loading snapshots of application-critical data pre-initialized in a warm-up run. GraalDOSS S/D performance is evaluated against most popular *Java* object S/D libraries — four *JSON* [7, 146, 120, 257] and two binary [272, 116] serializers. In all experiments, GraalDOSS achieved constant deserialization performance that overshadows other libraries by several orders of magnitude, while maintaining a competitive serialization performance. In macrobenchmarks that perform natural-language processing, GraalDOSS eliminated cold starts associated with language-processing pipeline preparation and input processing by reducing the linear time complexity of the entire application down to constant time complexity. Application speedup associated with GraalDOSS pre-initialization increases with larger models and inputs, reaching four orders of magnitude for English language *StanfordNLP* [294] models and inputs consisting of 3000 tokens.

Web API microservices often use in-memory caches during execution, which introduces data duplication and increases the memory footprint with multiple application instances. GraalDOSS pre-initializes and shares such caches across application instances, maintaining a constant memory footprint for shared caches, instead of a linear increase in traditional deployments. For 8 application instances and data caches of 30 MB, GraalDOSS improved the total memory footprint by 44% and first-response times by 34%. GraalDOSS significantly improved application density, allowing almost two times as many application instances to be deployed at the cost of the same amount of working memory.

10.2 Published results

DOSS architecture and reference implementation (GraalDOSS) were published in 2026 [315], including the S/D-*benchmarks* set [307, 310] used for GraalDOSS evaluation. A United States patent for DOSS was filed in 2025 under *ORC25139810-US-NPR* [314]. Insights from DOSS research were used to improve *GraalVM* language-runtime support for *Truffle* language implementation framework (see §3.4) code caching using an auxiliary heap that corresponds to DOSS snapshot heap. These improvements yielded close to an order of magnitude improvement in snapshotting times (6 seconds, down from 40 seconds for a 200 MB code cache), and a threefold improvement in code-cache loading times (5 seconds, down from 15 seconds).

Related research was focused on language-runtime pre-initialization problems and investigated ways of warming up language runtimes. As a product of such research, an early prototype of DOSS snapshot heap was used in combination with *Truffle* language implementation framework *JVM* context caching to speed up *JVM* initialization times [309]. The resulting context snapshot contained most of the pre-initialized classes contained in the *JDK* necessary for *JVM* to execute even the simplest *Java* code. As a result, the startup performance was improved by 49% on *Java* version 8 and 31% on *Java* versions 11 and 17, the three main long-term-supported *Java* versions at the time.

During this work, other techniques for memory-footprint optimizations were developed, meant to be used alongside DOSS to further optimize cloud-native deployments. To that extent, an application-guided garbage-collection policy was developed, which dynamically tunes runtime-heap configuration to tailor the runtime heap to allocation patterns observed in the application, and avoid garbage collections inside critical allocation regions [190, 189]. As a result, the policy achieved a geometric mean of 30% in density improvement when evaluated on multiple microservice benchmarks.

DOSS research greatly benefited from related work put into predictable bytecode analysis in *GraalVM Native Image* [346, 347], optimal graph-traversal strategies [312, 311] leveraging machine-learning [87, 313], and language-invariant abstract syntax trees [308]. This research gave valuable insight into graph properties and helped optimize GraalDOSS serialization performance during object-graph traversals. GraalDOSS evaluation was performed using insights gained from previous research on distributed *GraalVM* benchmarking [344], which helped design a properly isolated benchmarking setup that gave precise measurements and high measurement confidence. Research into distributed *GraalVM* evaluation allowed GraalDOSS to be evaluated on a wide range of workload objects in a relatively short amount of time.

10.3 Future work

DOSS provides several unique opportunities and research directions that are planned as future work. These include, but are not limited to, generalization of requirements for the language-runtime compatibility for GraalDOSS snapshots, introducing support for storing runtime-compiled code into GraalDOSS snapshots, providing a reusable snapshot heap interface, snapshot-file compression, and a safe snapshot-loading mode.

Generalization of requirements that GraalDOSS imposes for language-runtime compatibility and snapshot compatibility would yield more reusable snapshots. However, this is not an easy problem since it requires synchronization across multiple language-runtime instances. For example, the order of class initialization would need to be defined, and class hubs would need to be synchronized so that every class hub resides in the same place for every language-runtime instance. Otherwise, the snapshot would need to be patched prior to loading, introducing S/D overhead and hindering snapshot sharing. Solutions such as *ZCOT* [391] solve this problem by using a remote transfer space with a distributed class-data archive, which is managed by an external metadata server. However, *GraalVM* provides multiple mechanisms that can be leveraged to dynamically load dynamic data and support snapshot loading in otherwise non-compatible language-runtime instances. Future work in this direction would involve research on language-runtime coordination and metadata exchange, possibly through a modified *GraalVM* language runtime and enhanced GraalDOSS metadata section that combined would account for language-runtime incompatibilities.

One of the most important research directions is the ability to snapshot code compiled during program execution into direct snapshots. Frameworks such as the *Truffle language implementation framework* (see §3.4) are crucial for multi-language support in serverless and cloud-native environments (see §3.5). *Truffle* compiles guest-language code during program execution and stores it in the form of abstract syntax trees. The ability to dynamically snapshot code compiled during program execution, and then load it and share it across application instances, would greatly improve performance and warmup times of applications and functions whose initialization requires heavy CPU work. Dynamic evolution and sharing of code caches would greatly benefit serverless function execution. Although snapshotting code is possible with GraalDoss, the code would need to be installed manually during loading, which requires manual intervention and would most likely involve copy-on-write modifications of the code due to patching and other transformations. Thus, an important research direction would be providing native support in GraalDoss for compiled code snapshots.

GraalDoss snapshot heap architecture allows it to be easily extended and repurposed. One future research direction in this area is using the snapshot heap as a transfer space or a messaging queue for passing objects between language-runtime isolates. Such capabilities would enhance communication between microservices, which normally communicate through the networking stack, serializing and deserializing data on both ends. Microservices are often running in incompatible environments, with a strict security policy. In *GraalOS* however (see §3.5), *Truffle* language implementation framework allows unified code execution of multiple languages on the *GraalVM* language runtime (see §3.4) in the same process, with isolation being enforced through *GraalVM* memory isolates (see §3.3). Extending GraalDoss to support direct object passing between *GraalVM* memory isolates would bypass the networking stack and remove S/D overhead, greatly improving the performance of applications that involve high-frequency messaging and remote procedure calls (abbr. *RPC*).

Direct snapshots are less compact compared to their plain-text and binary counterparts and, as such, they take up more space on disk. Depending on the data inside and object layout strategy, GraalDoss snapshots can be very sparse due to page alignment and memory fragmentation inside individual aligned object chunks. Compressed snapshots are a research direction that would be beneficial for systems with limited disk capacity or systems that mount remote disks over a network. Future work in this area includes compression of snapshots during the storage process and their uncompression before loading. Another option is integrating GraalDoss into specialized in-memory and compressed file systems for compression transparency and high efficiency.

Finally, an important improvement to the overall security of GraalDoss involves introducing a safe-loading mode for GraalDoss snapshots, which enhances operation safety and usability at the cost of performance. In such a mode, the SNAPSHOT operation would allow the user to provide object replacement rules, allowing object modification before snapshotting. Such rules can be used to redact sensitive or private data, or to allow snapshotting objects that contain states such as open files and started threads. Upon loading a snapshot, the functionality of such objects can be restored through user-provided routines. To ensure that malicious snapshots do not get loaded, the safe-loading mode would augment the LOAD operation with a linear scan during loading and abort loading if it detects pointers to off-heap memory. Finally, in order to enforce isolation of snapshot slot, future work involves using enhancements in modern hardware such as memory-protection keys [237].

Appendix A

Detailed microbenchmark results

We group the results into multiple tables based on the performed operation (serialization or deserialization) and the workload type. Workload types include primitive types such as integers and doubles (`Int`, `Dbl`), strings of length `n` (`S[n]`), primitive arrays of size `n` (`Int[n]` or `Dbl[n]`), square matrices of size `n` (`Int[n][]` or `Dbl[n][]`), records with `n` fields (`R<>[n]`), lists of various types and sizes, hash maps of various types for keys and values with different sizes, and microservice POJOs, e.g., `Client` (abbr. `Client`). The breakdown of the `Client` and `Partner` POJO fields is shown in Tables A.1 and A.2, respectively.

Tables A.3 and A.4 show the serialization throughput for arrays and square matrices of primitive types, while Table A.5 shows the serialization throughput for strings, records, and POJOs. Tables A.6, A.7, and A.8 show serialization throughput for our list and map workloads. Throughput is measured in the number of serialization operations per second (ops/s, higher is better). Sizes correspond to serialized object sizes in JSON, binary (`Bin`), or a snapshot (`Snap`) generated by GraalDoss, measured in bytes. Similarly, tables A.9, A.10, and A.11 show deserialization throughput for arrays and square matrices of primitive types, as well as strings, records, and POJOs, while tables A.12, A.13, and A.14 show deserialization throughput for list and map workloads.

Table A.1: Breakdown of the `Client` POJO fields.

Field	Type	Description	Constraints
`id`	`long`	Client identifier	Unique ID
`index`	`int`	Work contract index	Positive integers
`isActive`	`boolean`	Account activity	None
`balance`	`BigDecimal`	Financial balance	None
`picture`	`String`	Picture URL	100 chars
`age`	`int`	Age	Positive integers
`eyeColor`	`EyeColor`	Eye color, enum, one of: `BROWN`, `BLUE`, or `GREEN`	Valid enum values
`name`	`String`	Full name	20 chars
`gender`	`String`	Gender	20 chars
`company`	`String`	Company name	20 chars
`emails`	`String[]`	Email list	10 items, each 20 chars
`phones`	`long[]`	Phone number list	10 items
`address`	`String`	Address	20 chars
`about`	`String`	Additional information	20 chars
`registered`	`LocalDate`	Registration date	None
`latitude`	`double`	Latitude	$[-180, 180]$
`longitude`	`double`	Longitude	$[-90, 90]$
`tags`	`List<String>`	Custom tags	50 items, each 10 chars
`partners`	`List<Partner>`	List of partners	30 items

Table A.2: Breakdown of the `Partner` POJO fields.

Field	Type	Description	Constraints
id	`long`	Partner identifier	Unique ID
name	`String`	Full name of the partner	20 chars
partnershipTime	`OffsetDateTime`	Local time of partnership in the time zone of the partner	Year ≥ 2000

Table A.3: Serialized size (in B) and throughput for arrays of primitive types, with array length `n` (`Int[n]` and `Dbl[n]`).

Object	Size (B) JSON	Bin	Snap	java [272]	jackson [120]	gson [146]	dsljson [257]	fastjson2 [7]	Kryo [116] def	ref	**Graal DOSS**
`Int[ 64]`	701	381	824	72 876	383 293	106 259	355 606	458 475	**482 214**	366 439	88 325
`Int[ 128]`	1417	768	1592	38 626	209 828	52 869	176 585	237 707	**246 108**	193 174	60 317
`Int[ 256]`	2809	1522	3128	18 863	102 367	27 056	88 381	119 573	**126 163**	96 301	34 875
`Int[ 512]`	5628	3036	6200	10 219	55 966	13 373	47 622	59 488	**62 862**	48 777	20 075
`Int[1024]`	11 271	6078	12 344	4915	25 616	6833	23 526	29 589	**31 635**	24 755	10 941
`Int[2048]`	22 568	12 174	24 632	2422	11 067	3082	10 965	13 031	**15 760**	12 451	5650
`Int[4096]`	44 959	24 333	49 208	1262	5306	1650	4913	5668	**7894**	6208	2877
`Dbl[ 64]`	1232	578	1336	74 589	122 463	74 097	128 149	68 435	**526 384**	387 222	90 194
`Dbl[ 128]`	2469	1155	2616	37 881	61 992	37 992	62 343	34 625	**266 194**	200 838	56 318
`Dbl[ 256]`	4920	2307	5176	18 577	30 961	18 905	32 438	17 314	**135 400**	102 350	37 574
`Dbl[ 512]`	9863	4611	10 296	9227	15 590	9163	16 077	8545	**67 783**	52 383	21 012
`Dbl[1024]`	19 746	9219	20 536	4889	7394	4602	8178	4265	**34 003**	25 671	11 361
`Dbl[2048]`	39 463	18 435	41 016	2263	3463	2224	3885	2100	**17 172**	13 079	5823
`Dbl[4096]`	78 898	36 867	81 976	1271	1277	1057	2011	1037	**8570**	6526	3101

Table A.4: Serialized size (in KB) and throughput for square matrices of primitive types, with matrix size `n` (`Int[n][]` and `Dbl[n][]`).

Object	Size (KB) JSON	Bin	Snap	java [272]	jackson [120]	gson [146]	dsljson [257]	fastjson2 [7]	Kryo [116] def	ref	**Graal DOSS**
`Int[  16][]`	2.8	1.6	5.4	16 641	93 851	24 729	111 178	77 516	**111 535**	82 350	70 001
`Int[  32][]`	11.3	6.2	14.8	4899	24 538	6517	27 414	22 579	**30 053**	23 007	25 747
`Int[  64][]`	45.1	24.4	52.0	1241	5378	1658	5483	4545	**7651**	5924	7532
`Int[ 128][]`	180.2	97.6	200.2	283	701	371	1291	1012	1904	1505	**2736**
`Int[ 256][]`	720.1	389.8	793.7	70	181	90	321	128	482	370	**642**
`Int[ 512][]`	2880	1558	3171	14	46	23	78	41	120	93	**145**
`Int[1024][]`	11 519	6229	12 365	3	12	6	20	12	30	23	**38**
`Int[2048][]`	46 068	24 908	48 225	0.6	3.0	1.2	5.0	2.4	7.5	5.9	**8.9**
`Dbl[  16][]`	4.9	2.3	7.4	16 714	31 726	18 113	16 758	29 845	**118 362**	88 118	33 993
`Dbl[  32][]`	19.8	9.3	22.9	4707	7700	4576	4176	7741	**32 146**	23 620	10 505
`Dbl[  64][]`	79.0	37.0	84.8	1115	1315	1029	1028	1992	**8277**	6283	2994
`Dbl[ 128][]`	315.9	147.8	331.3	285	301	238	253	441	**2089**	1591	676
`Dbl[ 256][]`	1263	590.6	1320	71	74	58	64	74	**505**	389	167
`Dbl[ 512][]`	5052	2361	5277	14	18	16	16	22	**128**	99	41
`Dbl[1024][]`	20 208	9440	20 577	2.9	4.9	4.0	4.0	5.6	**31.4**	24.3	10.2
`Dbl[2048][]`	80 823	37 755	80 250	0.5	1.3	1.1	1.0	1.5	**10.1**	4.2	2.3

Table A.5: Serialized size (in B) and throughput for strings of length `n` (`S[n]`), records with `n` primitive and string fields (`R<Int>[n]`, `R<Dbl>[n]`, `R<S[32]>[n]`), and the `Client` POJO (`Client`).

Object	Size (B) JSON	Bin	Snap	java [272]	jackson [120]	gson [146]	dsljson [257]	fastjson2 [7]	Kryo [116] def	ref	**Graal DOSS**
`S[ 32]`	34	33	104	2 241 883	2 754 863	2 369 467	3 328 652	3 915 962	**5 288 242**	3 938 018	163 558
`S[ 64]`	66	67	136	1 709 166	2 282 946	1 677 156	2 242 981	2 397 117	**3 268 123**	2 746 566	164 441
`S[128]`	130	131	200	1 217 154	1 728 841	1 126 503	1 534 674	1 323 160	**1 805 080**	1 610 783	150 201
`S[256]`	258	259	328	768 092	933 839	630 088	852 175	710 302	**987 798**	914 266	158 663
`S[512]`	514	515	584	532 695	**635 679**	353 321	468 058	377 507	511 301	488 447	160 607
`R<Int>[ 2]`	34	11	64	90 422	1 841 817	1 176 691	3 031 141	3 927 055	**5 695 287**	3 577 847	167 974
`R<Int>[ 4]`	64	21	72	88 565	1 462 874	668 767	2 024 897	2 567 878	**3 817 212**	2 618 882	171 947
`R<Int>[ 8]`	128	39	88	82 724	956 442	364 036	1 264 176	1 528 011	**2 318 674**	1 699 181	170 047
`R<Int>[16]`	265	79	120	80 345	649 660	180 676	745 516	829 724	**1 304 425**	987 960	169 461
`R<Dbl>[ 2]`	50	17	72	91 233	1 115 429	955 713	1 989 719	1 540 758	**6 184 348**	3 664 740	168 498
`R<Dbl>[ 4]`	98	33	88	87 541	781 492	541 215	1 231 020	842 728	**4 126 507**	2 690 437	170 842
`R<Dbl>[ 8]`	195	65	120	82 873	474 059	300 174	696 656	442 229	**2 271 038**	1 714 764	168 362
`R<Dbl>[16]`	399	129	184	81 415	272 412	152 431	357 116	219 439	**1 309 864**	966 526	153 952
`R<S[32]>[ 2]`	81	67	176	82 312	1 321 182	855 561	1 784 800	1 849 155	**2 331 535**	1 812 089	145 895
`R<S[32]>[ 4]`	161	133	296	80 141	883 566	479 465	1 044 143	1 022 215	**1 270 820**	1 039 862	143 886
`R<S[32]>[ 8]`	321	265	536	71 961	549 762	264 385	580 113	538 678	**655 664**	555 273	139 099
`R<S[32]>[16]`	648	529	1016	60 961	297 934	136 412	306 498	276 474	**350 095**	291 343	117 780
`Client`	4428	971	3536	22 855	61 848	23 198	125 557	65 900	**224 808**	59 968	48 293

Table A.6: Serialized size (in KB) and throughput for lists of variable types and sizes.

L<T>		Size (KB)			java	jackson	gson	dsljson	fastjson2	Kryo [116]		**Graal**
T	Size	JSON	Bin	Snap	[272]	[120]	[146]	[257]	[7]	def	ref	**DOSS**
Int	16	0.17	0.08	0.27	211 222	973 584	390 972	1 230 444	1 391 217	**1 709 295**	1 216 789	139 718
	32	0.35	0.16	0.46	129 301	572 663	199 174	641 337	767 759	**925 683**	679 103	109 308
	64	0.71	0.32	0.85	69 401	325 918	109 853	368 162	407 857	**475 298**	372 589	86 624
	128	1.40	0.64	1.62	36 342	181 021	52 355	176 781	209 401	**253 383**	197 637	58 178
	256	2.82	1.27	3.15	18 061	100 511	24 355	92 895	103 874	**133 223**	100 929	33 362
	512	5.66	2.53	6.22	9484	47 151	12 319	46 835	52 655	**66 349**	51 661	19 392
Dbl	16	0.31	0.13	0.40	208 689	388 654	257 188	447 714	266 826	**1 751 841**	1 252 367	137 192
	32	0.62	0.26	0.72	116 398	194 145	152 567	229 116	134 999	**1 010 276**	726 890	114 500
	64	1.24	0.52	1.36	66 989	115 522	64 136	117 541	68 377	**534 489**	399 355	88 985
	128	2.46	1.03	2.64	37 993	60 743	38 458	59 376	34 051	**285 069**	211 767	57 251
	256	4.95	2.05	5.20	18 212	30 473	16 981	31 035	16 768	**143 115**	108 551	37 074
	512	9.84	4.10	10.3	8967	15 653	8275	15 473	8243	**73 406**	55 051	20 912
S[32]	16	0.56	0.53	1.04	194 866	**439 221**	193 314	375 553	322 901	401 206	330 588	117 012
	32	1.12	1.06	2.00	98 910	**238 716**	107 159	205 065	168 737	206 942	173 598	85 091
	64	2.24	2.12	3.92	54 646	**125 536**	49 590	104 890	85 276	104 659	84 687	56 145
	128	4.48	4.23	7.76	26 092	**62 890**	26 001	50 770	43 190	52 722	43 210	34 078
	256	8.96	8.45	15.4	14 127	**30 678**	13 359	25 971	21 581	26 650	21 749	19 145
	512	17.92	16.90	30.80	6871	**15 440**	6717	12 936	10 848	13 349	10 723	10 552
R<Int>[2]	16	0.54	0.18	0.40	5172	257 034	79 543	237 037	357 612	**463 070**	309 381	125 402
	32	1.10	0.35	0.72	2618	146 666	38 019	117 457	179 090	**238 337**	159 150	108 496
	64	2.18	0.70	1.36	1297	74 913	19 800	65 377	89 676	**121 913**	82 301	85 257
	128	4.37	1.40	2.64	650	37 547	9834	33 906	44 672	**62 062**	42 588	53 780
	256	8.69	2.78	5.20	321	19 250	4592	14 794	21 642	31 254	20 718	**35 001**
	512	17.37	5.58	10.32	165	8993	2351	7536	10 694	15 711	10 680	**19 853**
R<Int>[16]	16	4.24	1.29	1.30	4998	49 936	11 824	44 336	54 110	80 653	65 507	**128 916**
	32	8.46	2.56	2.51	2599	22 974	5901	24 182	27 048	40 663	33 289	**95 323**
	64	16.96	5.14	4.94	1273	11 308	2913	11 045	12 983	20 406	16 564	**75 558**
	128	33.92	10.26	9.81	645	5449	1473	5320	6026	10 226	8352	**47 228**
	256	67.79	20.48	19.54	326	1882	728	2681	2934	5104	4248	**27 190**
	512	135.4	40.95	38.99	159	835	346	1189	1425	2574	2070	**15 535**
R<Dbl>[2]	16	0.81	0.28	0.53	5095	125 336	62 650	142 450	111 239	**463 835**	312 511	131 868
	32	1.61	0.55	0.98	2568	65 827	31 712	75 303	55 091	**236 733**	166 091	116 246
	64	3.24	1.09	1.87	1307	35 117	17 293	36 465	27 254	124 964	83 795	**90 269**
	128	6.48	2.18	3.66	655	17 957	8051	17 361	13 611	61 023	43 527	**52 895**
	256	12.94	4.36	7.25	320	8661	4008	8631	6935	31 345	21 584	**31 934**
	512	25.83	8.71	14.42	165	4504	1978	4259	3407	15 922	10 700	**18 732**
R<Dbl>[16]	16	6.36	2.07	2.32	5104	19 232	9871	22 152	13 935	82 772	69 263	**109 734**
	32	12.74	4.13	4.56	2480	9600	4907	11 563	6886	43 576	33 762	**91 517**
	64	25.42	8.26	9.04	1282	4650	2173	5260	3457	21 772	17 261	**63 551**
	128	50.83	16.52	18.00	640	2287	1240	2815	1676	11 076	8531	**39 409**
	256	101.6	33.03	35.92	319	887	484	1412	849	5520	4290	**22 979**
	512	203.6	66.05	71.76	156	386	286	640	415	2797	2178	**12 637**
R<S[32]>[2]	16	1.31	1.08	2.19	4973	153 136	61 283	125 457	139 818	**166 292**	118 716	67 227
	32	2.62	2.15	4.30	2446	75 328	30 705	65 057	71 044	**84 104**	61 596	48 273
	64	5.25	4.29	8.53	1221	39 412	15 201	32 726	34 661	**42 560**	29 855	28 370
	128	10.50	8.58	16.98	612	20 023	7747	16 380	17 870	**21 438**	16 020	16 354
	256	20.99	17.18	33.87	306	10 086	4025	7961	8802	**10 790**	7936	9181
	512	41.99	34.31	67.66	151	5016	1834	4280	4413	**5383**	3790	4546
R<S[32]>[16]	16	10.38	8.47	15.63	3592	21 710	8924	19 843	17 686	**22 030**	17 683	18 475
	32	20.77	16.93	31.18	1853	**11 132**	4615	9267	8864	10 994	9027	9777
	64	41.54	33.86	62.29	921	5308	2288	4660	4390	**5461**	4440	5111
	128	83.07	67.72	124.5	465	1708	1164	2463	2191	**2772**	2245	2654
	256	166.1	135.4	248.9	235	767	512	1112	1104	**1371**	1022	1355
	512	332.3	270.9	497.7	115	380	254	567	546	**690**	477	660
Client	16	34.99	22.20	69.14	2157	3763	1398	4869	4657	**7950**	4392	3605
	32	76.47	47.55	147.3	1062	1193	766	1890	2471	**2887**	2270	2008
	64	165.0	93.20	263.7	548	521	349	913	1322	**1693**	1081	940
	128	321.0	177.2	520.2	285	254	184	426	608	**776**	437	462
	256	645.1	375.0	1059.4	133	127	91	216	309	**384**	203	143
	512	1317.6	766.0	2060.0	62	61	44	55	156	**198**	101	53

Table A.7: Serialized size (in KB) and throughput for hash maps that have integer keys.

M<Int,V> V	Size	Size (B) JSON	Bin	Snap	java [272]	jackson [120]	gson [146]	dsljson [257]	fastjson2 [7]	Kryo [116] def	ref	**Graal DOSS**
Int	16	0.38	0.19	0.86	125 739	329 551	150 532	602 847	634 330	**807 345**	624 029	86 006
	32	0.77	0.38	1.63	69 778	180 311	72 962	327 795	334 056	**402 604**	300 519	60 775
	64	1.53	0.76	3.17	36 859	90 366	39 231	182 356	174 076	**195 661**	160 474	35 798
	128	3.08	1.52	6.24	19 135	49 071	18 550	87 185	88 670	**96 382**	78 559	20 861
	256	6.16	3.05	12.38	8943	25 122	9187	47 288	44 307	**47 359**	38 564	11 202
	512	12.30	6.08	24.67	4444	11 742	4563	21 981	20 881	**23 096**	19 260	6215
S[8]	16	0.39	0.26	1.25	124 807	301 891	145 507	486 001	**515 005**	485 191	356 563	76 547
	32	0.77	0.51	2.40	68 232	151 620	76 117	258 121	**270 656**	235 151	183 812	46 551
	64	1.53	1.02	4.70	33 573	75 775	39 187	125 688	**137 820**	117 881	90 608	29 170
	128	3.07	2.05	9.31	16 954	38 724	18 250	66 813	**68 780**	59 434	44 011	17 640
	256	6.13	4.08	18.53	8977	20 177	9593	31 769	**33 702**	29 789	21 431	9323
	512	12.30	8.15	36.96	4302	9449	4693	**16 191**	15 894	14 636	10 891	4910
S[32]	16	0.77	0.64	1.63	99 786	210 413	117 191	275 986	251 340	**278 966**	230 822	78 871
	32	1.54	1.28	3.17	53 679	119 467	55 416	138 304	126 988	**142 102**	114 414	46 561
	64	3.08	2.56	6.24	28 621	58 446	29 544	70 120	63 334	**70 942**	57 073	27 380
	128	6.15	5.12	12.38	13 616	28 952	15 382	**37 154**	31 138	35 383	28 391	16 920
	256	12.30	10.22	24.67	6687	14 198	7616	**18 335**	15 333	17 620	14 138	9402
	512	24.56	20.44	49.25	3523	7599	3679	**9088**	7567	8941	7290	4724
R<Int>[2]	16	0.75	0.29	0.99	4961	159 476	58 612	224 436	283 974	**308 280**	222 224	82 193
	32	1.50	0.57	1.89	2492	79 818	29 088	116 876	144 182	**151 303**	108 341	60 716
	64	3.00	1.14	3.68	1260	40 898	15 001	62 644	73 831	**77 727**	54 474	34 350
	128	6.02	2.29	7.26	624	20 310	7120	29 916	36 210	**37 358**	26 970	21 250
	256	11.99	4.56	14.43	310	10 315	3692	15 583	17 221	**19 066**	13 743	10 992
	512	24.00	9.10	28.77	154	4921	1854	7372	7950	**9510**	6964	6153
R<Int>[16]	16	4.45	1.40	1.89	4780	43 645	11 143	46 854	52 750	71 176	61 960	**84 232**
	32	8.88	2.79	3.68	2471	20 573	5494	21 837	25 726	36 670	29 662	**54 517**
	64	17.76	5.55	7.26	1241	10 223	2773	10 917	12 369	18 251	15 138	**32 125**
	128	35.55	11.12	14.43	601	5074	1413	5164	5660	9291	7507	**18 371**
	256	71.01	22.24	28.77	305	1791	685	2622	2772	4647	3736	**10 425**
	512	142.2	44.50	57.44	152	746	323	1158	1330	2308	1834	**5355**
R<Dbl>[2]	16	1.02	0.39	1.12	4841	98 145	50 856	124 256	96 200	**291 950**	212 208	80 887
	32	2.03	0.77	2.14	2480	49 834	25 580	67 919	48 713	**149 467**	106 310	55 002
	64	4.06	1.53	4.19	1247	25 671	12 968	34 063	24 321	**74 411**	53 872	36 086
	128	8.11	3.06	8.29	628	12 562	6339	16 113	12 046	**37 175**	26 679	20 776
	256	16.27	6.13	16.48	312	6106	3220	7944	6015	**18 213**	13 271	11 367
	512	32.49	12.25	32.86	156	3290	1642	4211	2944	**9149**	6575	5957
R<Dbl>[16]	16	6.57	2.18	2.91	4821	18 887	8907	22 931	13 793	**74 807**	59 867	74 545
	32	13.12	4.35	5.73	2374	9076	4178	11 309	6717	37 189	30 171	**53 642**
	64	26.26	8.70	11.36	1222	4573	2062	5368	3351	19 101	14 926	**31 098**
	128	52.56	17.40	22.62	616	2071	1194	2639	1677	9558	7323	**17 207**
	256	105.1	34.80	45.15	298	864	469	1314	815	4884	3678	**9313**
	512	210.1	69.61	90.21	152	369	276	623	403	2411	1850	**4776**
R<S[32]>[2]	16	1.52	1.19	2.78	4721	104 851	50 571	119 027	117 316	**136 765**	103 075	50 424
	32	3.04	2.37	5.47	2378	57 153	23 689	58 511	58 351	**68 397**	51 396	33 556
	64	6.09	4.74	10.85	1189	28 947	12 814	29 253	29 465	**34 319**	25 969	18 463
	128	12.16	9.47	21.60	589	14 755	6138	15 470	14 551	**16 955**	12 525	10 423
	256	24.30	18.93	43.10	289	6926	3052	7729	7321	**8395**	6330	5354
	512	48.62	37.85	86.11	146	3354	1565	3633	3680	**4289**	3170	2775
R<S[32]>[16]	16	10.59	8.58	16.22	3556	19 846	8119	19 507	17 165	**21 454**	17 440	16 352
	32	21.19	17.15	32.35	1725	9721	4443	9799	8686	**10 778**	8621	8640
	64	42.38	34.30	64.61	886	5005	2048	4610	4334	**5293**	4332	4758
	128	84.73	68.60	129.1	442	1462	904	2440	2147	**2664**	2159	2402
	256	169.5	137.2	258.1	218	730	461	1088	1081	**1335**	967	1240
	512	338.9	274.4	516.2	113	362	224	526	529	**667**	470	594
Client	16	42.31	25.17	54.69	2131	3839	1657	4246	4352	**5904**	5365	3686
	32	95.29	44.48	138.8	1091	946	807	2142	2549	**2906**	2528	1502
	64	158.4	93.54	266.3	515	487	386	926	1189	**1621**	995	878
	128	332.5	189.2	541.4	268	269	184	462	633	**751**	461	422
	256	679.1	368.8	1051.6	133	122	89	216	320	**416**	217	141
	512	1291.3	750.8	2145.2	65	61	45	56	150	**201**	102	55

Table A.8: Serialized size (in KB) and throughput for hash maps with length 8 string keys.

M<S[8],V> V	Size	Size (KB) JSON	Bin	Snap	java [272]	jackson [120]	gson [146]	dsljson [257]	fastjson2 [7]	Kryo [116] def	ref	**Graal DOSS**
Int	16	0.35	0.26	1.25	123 577	454 082	171 609	508 278	**532 540**	468 579	360 978	75 807
	32	0.70	0.51	2.40	69 267	237 925	92 859	256 536	**286 623**	234 617	182 293	49 563
	64	1.41	1.02	4.70	34 049	127 766	47 034	132 404	**144 626**	118 729	90 723	29 123
	128	2.81	2.04	9.31	17 195	64 408	22 616	70 852	**73 111**	58 259	43 468	17 504
	256	5.62	4.08	18.53	8798	33 948	11 731	35 348	**35 435**	29 235	21 741	9421
	512	11.24	8.15	36.96	4355	16 412	5462	16 301	**16 634**	14 566	10 843	5029
S[8]	16	0.35	0.32	1.63	143 624	357 731	163 565	381 517	**447 498**	392 678	283 864	72 310
	32	0.70	0.64	3.17	78 081	192 511	81 270	207 108	**229 504**	197 097	142 799	44 143
	64	1.41	1.28	6.24	40 704	106 020	43 011	107 347	**111 749**	100 907	70 763	26 346
	128	2.82	2.56	12.38	20 469	51 810	21 989	54 460	**56 141**	49 344	33 789	15 517
	256	5.63	5.12	24.67	9998	26 126	10 467	26 243	**27 669**	24 211	16 413	8156
	512	11.27	10.24	49.25	4975	13 881	5721	13 309	**13 554**	12 168	8522	4292
S[32]	16	0.74	0.71	2.02	121 667	252 365	119 407	**266 866**	235 505	247 589	195 049	73 165
	32	1.47	1.41	3.94	64 130	**144 002**	63 089	134 697	118 770	126 767	98 639	42 811
	64	2.94	2.82	7.78	31 120	**70 063**	32 853	65 882	59 210	63 896	49 129	25 752
	128	5.89	5.64	15.46	14 819	**35 304**	16 337	34 502	29 329	32 072	24 174	15 095
	256	11.78	11.27	30.82	7513	**18 134**	8629	17 386	14 790	15 966	12 300	7876
	512	23.55	22.53	61.54	3531	**9098**	3969	8694	7348	7893	6021	4159
R<Int>[2]	16	0.72	0.35	1.38	4992	171 218	60 918	162 321	**251 007**	248 462	170 422	82 526
	32	1.45	0.70	2.66	2481	92 181	31 867	80 678	**130 451**	125 307	87 194	48 974
	64	2.88	1.40	5.22	1248	48 955	15 189	38 271	**64 073**	61 929	44 323	30 824
	128	5.74	2.81	10.34	622	24 760	7924	19 638	29 880	**31 656**	21 880	17 381
	256	11.51	5.60	20.58	312	12 250	4025	9695	14 708	**15 823**	11 040	9378
	512	23.06	11.19	41.06	152	6178	2011	4620	7213	**7851**	5483	5066
R<Int>[16]	16	4.42	1.46	2.27	4836	45 676	11 264	43 581	51 441	**71 941**	56 657	68 311
	32	8.85	2.92	4.45	2468	22 219	5596	20 123	24 745	35 200	27 822	**46 537**
	64	17.62	5.82	8.80	1239	11 040	2723	10 722	12 111	17 787	13 811	**29 324**
	128	35.30	11.66	17.50	611	5393	1431	5126	5717	8675	6953	**16 140**
	256	70.62	23.31	34.91	312	1888	697	2236	2787	4386	3499	**8503**
	512	141.1	46.58	69.73	153	778	325	1060	1335	2187	1732	**4382**
R<Dbl>[2]	16	0.98	0.45	1.50	4929	108 094	53 704	113 163	91 654	**250 680**	171 245	77 323
	32	1.98	0.90	2.91	2480	58 975	27 004	52 799	46 017	**124 924**	88 133	50 818
	64	3.94	1.79	5.73	1250	26 607	12 998	27 642	23 430	**63 033**	43 966	30 609
	128	7.86	3.59	11.36	615	14 801	6766	13 983	11 607	**31 718**	21 992	16 934
	256	15.75	7.17	22.62	313	6586	3348	6942	5767	**15 632**	10 999	8918
	512	31.47	14.34	45.15	152	3253	1720	3325	2849	**7930**	5526	4882
R<Dbl>[16]	16	6.55	2.24	3.30	4764	19 113	9003	21 329	13 449	**72 139**	56 507	65 271
	32	13.12	4.48	6.50	2430	9738	4817	10 496	6748	35 394	28 034	**41 929**
	64	26.14	8.96	12.90	1201	4476	2098	5417	3347	17 990	13 987	**26 574**
	128	52.25	17.92	25.70	601	2171	1159	2574	1644	9247	7068	**15 228**
	256	104.5	35.84	51.30	299	828	576	1340	819	4615	3536	**7852**
	512	209.1	71.68	102.5	153	379	276	571	404	2316	1743	**4035**
R<S[32]>[2]	16	1.49	1.25	3.17	4827	114 336	53 173	105 648	113 564	**126 688**	96 718	50 720
	32	2.98	2.50	6.24	2394	63 266	26 620	51 472	57 393	**64 721**	48 029	29 220
	64	5.95	4.99	12.38	1150	30 896	12 351	25 692	29 005	**31 855**	24 668	17 478
	128	11.90	9.99	24.67	590	15 587	6267	13 594	14 535	**16 089**	12 127	9862
	256	23.81	19.97	49.25	299	**8162**	3183	6761	7321	8064	6002	5019
	512	47.62	39.94	98.40	143	**4106**	1527	3658	3039	4066	3017	2610
R<S[32]>[16]	16	10.56	8.64	16.61	3680	**21 283**	8178	18 088	16 871	21 068	16 813	16 012
	32	21.12	17.28	33.12	1829	10 232	4124	9511	8564	**10 606**	8514	8466
	64	42.24	34.56	66.14	900	**5364**	2265	4785	4268	5240	4253	4553
	128	84.48	69.12	132.2	458	1598	1009	2196	2162	**2648**	2134	2333
	256	169.0	138.2	264.3	227	749	470	1060	1075	**1320**	966	1188
	512	337.9	276.5	528.5	110	374	233	528	529	**659**	426	582
Client	16	43.90	26.06	68.13	2167	3179	1884	4145	5229	**7260**	4574	3523
	32	78.42	46.62	139.2	1152	1249	604	2154	2294	**3078**	2292	1531
	64	154.5	98.44	272.6	540	466	358	906	1361	**1444**	964	893
	128	319.0	185.4	540.0	248	247	177	421	653	**768**	461	443
	256	676.5	381.3	1082.5	126	123	87	221	315	**388**	216	144
	512	1291.0	754.0	2141.3	67	62	45	57	156	**204**	97	55

Table A.9: Deserialization throughput for arrays of primitive types, with array length `n` (`Int[n]` and `Dbl[n]`).

Object	java [272]	jackson [120]	gson [146]	dsljson [257]	fastjson2 [7]	Kryo [116] def	Kryo [116] ref	**Graal DOSS**
`Int[  64]`	37 618	251 389	172 243	197 461	301 280	**617 634**	436 596	187 658
`Int[ 128]`	27 337	129 786	80 699	95 099	152 594	**322 108**	218 517	173 765
`Int[ 256]`	14 339	66 919	41 272	49 478	79 613	**159 025**	114 955	157 818
`Int[ 512]`	7408	33 284	20 879	25 868	41 588	85 035	56 679	**167 273**
`Int[1024]`	4194	16 671	10 275	12 438	19 470	42 908	30 451	**147 903**
`Int[2048]`	1663	8439	5027	5973	8841	21 527	15 177	**153 234**
`Int[4096]`	724	4128	2553	2977	4074	10 673	7534	**139 654**
`Dbl[  64]`	32 048	39 766	41 167	125 662	127 046	**665 010**	428 417	133 265
`Dbl[ 128]`	23 932	18 366	19 276	61 390	62 689	**308 502**	222 255	149 701
`Dbl[ 256]`	8250	9735	9056	32 490	29 692	**174 801**	116 688	139 459
`Dbl[ 512]`	7225	4625	4661	16 681	16 055	77 095	57 699	**137 014**
`Dbl[1024]`	2745	2348	2351	7872	7683	44 229	28 802	**134 839**
`Dbl[2048]`	1932	1154	1122	4207	3735	22 544	14 885	**130 185**
`Dbl[4096]`	784	568	579	1797	1904	10 980	7472	**124 579**

Table A.10: Deserialization throughput for square matrices of primitive types, with matrix size `n` (`Int[n][]` and `Dbl[n][]`).

Object	java [272]	jackson [120]	gson [146]	dsljson [257]	fastjson2 [7]	Kryo [116] def	Kryo [116] ref	**Graal DOSS**
`Int[  16][]`	10 969	48 856	30 075	51 276	36 412	116 637	72 324	**148 900**
`Int[  32][]`	2590	12 352	7526	12 188	9165	31 445	19 051	**130 455**
`Int[  64][]`	653	2983	1823	2627	2093	8664	5232	**119 560**
`Int[ 128][]`	154	767	494	725	534	1990	1265	**99 005**
`Int[ 256][]`	38	178	117	168	127	519	325	**86 221**
`Int[ 512][]`	10	44	30	40	33	130	77	**80 312**
`Int[1024][]`	2	10	7	8	8	29	17	**71 356**
`Int[2048][]`	0.4	2.2	0.9	1.3	1.2	4.2	3.0	**70 143**
`Dbl[  16][]`	10 493	6692	6857	15 877	25 378	**133 322**	74 357	117 549
`Dbl[  32][]`	2826	1601	1645	3972	6001	36 028	20 751	**106 896**
`Dbl[  64][]`	491	392	409	907	1365	8558	5214	**92 341**
`Dbl[ 128][]`	145	92	97	232	345	2275	1405	**87 422**
`Dbl[ 256][]`	38	22	23	42	71	607	353	**80 875**
`Dbl[ 512][]`	8	5	4	8	14	49	43	**75 918**
`Dbl[1024][]`	2	1	1	2	3	11	9	**71 126**
`Dbl[2048][]`	0.4	0.2	0.3	0.5	0.6	2.1	1.7	**67 389**

Table A.11: Deserialization throughput for strings of length `n` (`S[n]`), records with `n` primitive and string fields (`R<Int>[n]`, `R<Dbl>[n]`, `R<S[32]>[n]`), and the `Client` POJO (`Client`).

Object	java [272]	jackson [120]	gson [146]	dsljson [257]	fastjson2 [7]	Kryo [116] def	Kryo [116] ref	**Graal DOSS**
`S[ 32]`	1 872 283	3 395 379	1 783 320	5 711 572	5 623 927	**10 484 551**	6 744 406	142 806
`S[ 64]`	1 225 173	2 772 522	1 586 876	4 427 345	4 211 187	**5 318 256**	4 210 199	136 620
`S[128]`	893 498	2 014 139	1 238 484	3 021 643	3 025 440	**4 432 917**	3 389 994	129 365
`S[256]`	550 360	1 520 298	835 444	1 901 453	2 091 338	**2 383 228**	2 143 596	125 483
`S[512]`	324 940	880 758	559 229	1 048 605	1 083 752	**1 390 559**	1 215 616	123 382
`R<Int>[ 2]`	64 133	1 596 469	1 108 177	1 923 354	2 947 283	**7 022 272**	4 109 568	122 866
`R<Int>[ 4]`	58 411	1 097 617	741 262	1 120 396	2 042 536	**5 407 925**	3 282 370	122 866
`R<Int>[ 8]`	37 267	749 483	515 023	649 814	1 328 728	**3 485 859**	2 316 106	126 549
`R<Int>[16]`	32 762	404 625	326 185	315 811	814 840	**2 125 323**	1 441 567	130 116
`R<Dbl>[ 2]`	65 040	729 969	530 718	931 167	1 217 279	**6 539 167**	4 125 750	131 838
`R<Dbl>[ 4]`	50 376	382 239	379 623	468 674	674 288	**5 096 519**	3 130 102	129 743
`R<Dbl>[ 8]`	45 633	239 340	225 561	255 494	415 544	**3 690 160**	2 247 270	116 525
`R<Dbl>[16]`	35 264	124 674	122 359	146 674	230 628	**1 979 432**	1 340 712	112 843
`R<S[32]>[ 2]`	52 554	1 275 606	904 955	1 916 064	2 556 405	**3 930 739**	2 477 765	121 600
`R<S[32]>[ 4]`	39 203	954 692	650 438	1 037 638	1 774 669	**2 356 955**	1 490 479	123 341
`R<S[32]>[ 8]`	34 090	588 885	444 548	606 663	1 060 371	**1 424 540**	1 036 089	127 946
`R<S[32]>[16]`	24 652	316 717	267 182	296 766	557 797	**766 168**	547 654	100 702
`Client`	5043	14 461	12 999	52 885	77 629	**153 808**	132 619	136 255

Table A.12: Deserialization throughput for lists of variable types and sizes.

L<T> T	Size	java [272]	jackson [120]	gson [146]	dsljson [257]	fastjson2 [7]	Kryo [116] def	Kryo [116] ref	**Graal DOSS**
Int	16	63 543	733 459	491 930	570 570	1 003 186	**2 336 265**	1 550 977	123 205
	32	44 667	394 429	303 044	296 722	572 248	**1 355 123**	940 566	121 261
	64	29 396	216 936	157 534	176 853	301 176	**719 201**	442 639	120 434
	128	22 962	116 134	83 157	87 973	142 653	**360 335**	242 924	117 061
	256	12 372	57 842	42 277	43 434	76 602	**192 718**	133 845	137 530
	512	6574	29 171	21 068	22 489	42 218	87 097	65 342	**126 211**
Dbl	16	58 323	135 116	143 919	373 050	437 126	**2 428 473**	1 494 059	125 111
	32	45 960	74 388	73 539	220 514	203 700	1 291 844	874 392	137 879
	64	33 996	40 019	38 509	115 841	105 418	**716 957**	403 663	105 711
	128	18 292	19 694	19 361	58 965	57 224	**394 596**	248 336	120 885
	256	10 653	9940	9517	30 345	30 136	**185 807**	122 217	132 871
	512	5989	4386	4739	15 620	14 292	101 480	56 966	**129 397**
S[32]	16	69 094	444 789	353 922	561 950	706 063	**812 142**	595 884	101 744
	32	57 202	237 612	215 359	293 955	350 053	**463 531**	314 910	127 058
	64	32 995	129 037	116 386	175 020	212 555	**233 012**	164 749	124 749
	128	19 420	62 101	60 386	77 470	106 831	121 343	89 406	**126 986**
	256	6445	32 425	32 676	40 600	55 845	61 174	43 233	**123 936**
	512	2957	17 020	16 931	21 024	28 116	29 953	22 135	**124 477**
R<Int>[2]	16	23 048	202 827	152 337	181 513	254 403	**640 752**	417 594	107 371
	32	14 823	110 846	78 023	92 599	129 163	**339 555**	196 390	118 270
	64	8445	56 734	40 865	49 222	64 824	**183 930**	115 251	115 528
	128	4808	26 572	20 393	24 855	32 607	95 781	59 224	**115 665**
	256	2569	14 488	10 716	12 313	16 253	47 544	30 301	**140 177**
	512	1386	6963	5262	5663	7826	23 320	15 397	**135 637**
R<Int>[16]	16	6656	37 281	19 506	26 552	36 593	**165 260**	106 084	114 803
	32	3166	18 729	9765	14 653	17 626	70 220	54 640	**141 795**
	64	1339	9237	4199	7126	8565	40 575	27 718	**127 626**
	128	656	4319	2339	3396	4216	17 713	14 111	**120 450**
	256	515	2076	1226	1518	2099	8869	7035	**140 981**
	512	265	1023	599	706	1062	5037	3508	**133 370**
R<Dbl>[2]	16	22 065	59 940	55 511	136 327	149 901	**635 866**	391 664	110 276
	32	13 638	26 982	28 756	64 872	69 227	**347 520**	208 637	116 408
	64	7082	14 814	13 903	37 633	34 661	**181 307**	110 097	104 597
	128	4123	7573	7116	19 021	18 740	90 792	57 103	**127 205**
	256	2583	3659	3450	9237	9002	46 929	28 249	**128 898**
	512	1275	1668	1745	4283	4261	23 577	14 229	**132 578**
R<Dbl>[16]	16	4819	7434	6958	18 545	18 512	**164 297**	103 472	109 652
	32	2182	3903	3326	10 726	9374	85 090	53 639	**132 478**
	64	1190	1962	1632	5228	4092	41 877	27 131	**101 741**
	128	670	948	799	2549	2001	18 374	13 807	**122 009**
	256	483	463	405	1139	992	11 338	7114	**112 534**
	512	227	232	201	581	562	5837	3515	**123 746**
R<S[32]>[2]	16	16 683	148 800	110 609	185 022	231 669	**272 033**	222 455	124 939
	32	8933	71 381	67 292	96 973	122 425	**162 994**	115 738	132 585
	64	5104	41 275	35 373	51 701	61 195	85 183	59 341	**130 968**
	128	2560	20 301	18 957	25 910	26 271	41 484	30 165	**135 227**
	256	1227	10 600	7406	14 417	15 213	21 621	15 361	**129 841**
	512	478	5490	4618	6927	6853	10 908	6519	**127 971**
R<S[32]>[16]	16	2098	23 169	16 620	28 799	30 974	54 866	39 584	**128 563**
	32	1563	11 528	8577	15 595	15 324	26 980	19 948	**130 283**
	64	616	5929	4252	7888	7235	13 678	9833	**123 003**
	128	393	3241	2137	2639	3175	6787	5011	**101 941**
	256	117	1647	1040	1447	1886	3386	2445	**91 439**
	512	92	771	414	624	913	1670	1190	**92 197**
Client	16	714	4415	3548	5449	5265	10 683	5795	**118 936**
	32	360	1889	1954	2138	2121	4959	3272	**111 801**
	64	151	1058	866	1177	1064	2087	1683	**97 880**
	128	102	521	427	547	514	1227	880	**82 802**
	256	53	275	224	284	266	632	418	**79 725**
	512	30	131	97	127	104	272	193	**78 140**

Table A.13: Deserialization throughput for hash maps that have integer keys.

M<Int,V> V	Size	java [272]	jackson [120]	gson [146]	dsljson [257]	fastjson2 [7]	Kryo [116] def	Kryo [116] ref	**Graal DOSS**
Int	16	48 176	377 743	284 141	335 981	421 008	**814 154**	738 622	110 533
	32	34 372	213 398	161 757	180 221	215 293	**426 125**	385 086	124 924
	64	23 567	113 942	84 679	92 515	108 800	**276 257**	202 597	114 604
	128	10 202	60 079	44 197	45 455	55 781	**141 703**	100 302	122 797
	256	6606	31 357	21 475	22 826	26 109	69 458	50 834	**115 173**
	512	3268	14 792	10 426	11 496	13 080	28 684	23 218	**106 296**
S[8]	16	44 440	389 399	362 258	385 504	440 400	**663 566**	463 145	115 231
	32	32 083	214 756	192 384	202 503	231 949	**347 166**	247 379	118 859
	64	18 927	123 815	101 013	109 355	117 919	**175 637**	126 145	124 669
	128	10 353	63 597	49 818	54 564	54 573	97 385	66 320	**121 085**
	256	4301	30 006	26 862	27 914	27 865	45 714	31 508	**123 498**
	512	2918	14 153	13 244	11 789	13 760	24 653	16 729	**109 894**
S[32]	16	33 918	299 765	237 492	310 991	364 675	**510 597**	371 010	115 842
	32	25 147	163 430	136 857	131 653	183 012	**263 483**	200 999	119 547
	64	18 224	82 772	73 279	85 049	93 689	**140 741**	102 447	114 099
	128	8993	42 379	37 575	42 488	35 933	66 689	50 801	**120 797**
	256	3388	20 135	17 817	21 063	21 476	35 705	25 526	**116 820**
	512	1046	9714	8969	11 010	10 730	17 863	13 601	**110 696**
R<Int>[2]	16	18 203	151 250	114 102	176 201	195 310	**453 507**	300 201	106 234
	32	11 018	83 291	64 332	90 360	99 380	**237 951**	157 061	107 383
	64	5916	45 550	31 569	46 243	48 621	116 812	83 078	**123 878**
	128	4133	22 687	16 231	22 977	24 330	59 625	40 827	**115 350**
	256	2036	10 738	8042	12 054	11 660	30 272	21 108	**115 547**
	512	1077	5280	3941	5644	5412	15 004	10 433	**115 181**
R<Int>[16]	16	5635	33 131	19 026	31 652	35 757	**143 774**	98 041	120 826
	32	3013	17 495	9352	15 464	17 800	74 023	50 169	108 989
	64	1373	8178	4668	7651	8527	37 280	25 220	**104 517**
	128	797	3883	2371	3641	4067	13 674	12 441	**104 223**
	256	455	1893	1137	1536	1968	9335	6389	**114 394**
	512	253	949	554	804	991	4449	3060	**119 523**
R<Dbl>[2]	16	14 983	51 619	49 789	128 794	115 430	**425 025**	279 772	125 202
	32	9547	27 081	24 276	68 737	56 435	**219 120**	148 085	121 056
	64	6350	13 674	13 177	33 955	32 829	**113 279**	76 250	110 717
	128	3962	6705	6305	17 540	14 332	60 183	39 672	**103 001**
	256	1602	3410	3140	8656	7275	30 342	20 401	**102 441**
	512	981	1616	1493	4357	3524	14 625	9997	**109 946**
R<Dbl>[16]	16	4711	7773	6569	21 666	18 517	**140 847**	92 543	127 650
	32	2410	3636	3329	11 259	9272	74 921	46 746	**107 365**
	64	1363	1888	1580	4261	4764	37 066	24 112	**107 769**
	128	546	912	806	2734	2233	18 851	12 655	**111 181**
	256	486	435	389	1201	1093	9411	6107	**116 818**
	512	253	224	197	602	532	4638	3031	**121 746**
R<S[32]>[2]	16	14 248	115 453	112 701	158 044	162 899	**247 413**	172 765	124 491
	32	6456	62 574	55 209	80 019	80 339	94 213	87 872	**127 710**
	64	4280	31 071	29 076	42 267	40 618	66 816	46 679	**113 063**
	128	2411	15 758	14 766	21 464	21 342	33 442	23 240	**116 800**
	256	1073	8026	7243	10 740	10 166	16 831	11 416	**102 717**
	512	582	4187	3535	5345	4959	6235	5808	**110 904**
R<S[32]>[16]	16	3363	21 295	15 837	28 484	27 993	49 645	36 114	**120 659**
	32	1383	11 363	8280	14 334	13 952	25 464	18 084	**113 688**
	64	663	5731	4092	7750	7296	12 782	8978	**116 276**
	128	410	2859	2055	2886	3487	6099	4497	**101 791**
	256	208	1455	982	1506	1719	3147	2218	**87 404**
	512	110	736	515	711	797	1555	1106	**86 673**
Client	16	674	4620	3778	5670	4314	9226	6942	**99 180**
	32	272	2002	1512	2170	2229	4654	3411	**95 990**
	64	207	986	883	1077	1098	2358	1749	**94 508**
	128	114	519	407	537	493	1148	903	**86 272**
	256	54	249	201	264	184	589	405	**77 296**
	512	31	122	99	92	121	284	192	**70 614**

Table A.14: Deserialization throughput for hash maps with length 8 string keys.

M<Int,V> V	Size	java [272]	jackson [120]	gson [146]	dsljson [257]	fastjson2 [7]	Kryo [116] def	Kryo [116] ref	**Graal DOSS**
Int	16	36 336	398 158	282 571	410 973	424 300	**644 522**	444 012	119 826
	32	32 986	211 345	161 563	209 724	218 264	**328 591**	237 010	112 769
	64	18 776	115 193	82 095	113 334	115 633	**171 014**	123 639	110 649
	128	12 753	57 873	40 646	56 405	56 392	91 907	63 755	**111 384**
	256	5060	30 916	20 786	26 315	27 778	42 862	30 398	**115 362**
	512	2415	15 752	10 690	12 606	12 836	22 502	12 106	**110 228**
S[8]	16	78 122	371 029	333 760	**473 723**	461 902	588 928	371 083	117 120
	32	30 205	209 410	210 156	222 099	**238 855**	202 262	198 515	113 373
	64	31 143	114 160	110 072	125 379	126 012	**145 706**	98 017	109 744
	128	11 609	58 010	57 869	59 122	61 227	74 374	51 648	**110 324**
	256	5250	28 007	29 213	30 861	27 675	38 424	24 692	**110 781**
	512	2915	14 675	13 731	16 887	13 265	19 067	13 109	**110 160**
S[32]	16	39 135	300 370	257 497	371 105	384 129	**458 113**	320 638	118 881
	32	43 894	142 676	142 992	177 965	195 486	**235 891**	166 319	105 768
	64	25 197	77 776	72 212	95 205	95 172	**122 060**	86 690	107 028
	128	10 933	39 113	35 395	48 291	48 178	61 222	42 731	**123 335**
	256	3893	19 551	19 222	24 275	22 941	31 380	21 396	**112 798**
	512	885	9732	9262	12 315	11 091	15 335	11 224	**103 888**
R<Int>[2]	16	16 981	170 067	119 965	165 436	195 383	**343 355**	164 604	114 081
	32	9352	88 735	61 783	89 861	100 855	**185 549**	121 171	117 802
	64	6938	41 045	32 333	45 053	51 827	95 412	65 013	**117 685**
	128	3949	22 446	16 220	22 157	24 461	48 509	22 789	**120 945**
	256	1991	10 862	7991	11 042	11 465	24 526	16 420	**113 555**
	512	1058	5404	3951	5632	5557	12 775	8454	**114 960**
R<Int>[16]	16	6102	33 361	18 686	30 413	36 250	**132 431**	90 677	117 207
	32	2383	17 566	9457	15 855	17 677	67 682	46 796	**109 509**
	64	1649	7857	4412	7678	8509	34 010	23 519	**106 041**
	128	882	3722	2333	3712	3808	16 644	11 337	**111 909**
	256	455	1973	1170	1508	1943	8359	5765	**97 948**
	512	244	992	571	808	983	3995	2811	**101 882**
R<Dbl>[2]	16	18 907	53 462	49 622	127 764	136 093	**334 287**	229 423	114 000
	32	10 088	27 171	27 031	66 416	60 500	**179 825**	120 608	110 243
	64	6912	12 930	12 961	36 670	31 365	96 215	63 474	**103 133**
	128	4067	6999	6318	17 379	15 364	46 539	31 874	**116 387**
	256	1825	3232	3112	8712	7281	24 727	16 341	**99 343**
	512	903	1647	1548	4477	3616	12 822	8417	**105 721**
R<Dbl>[16]	16	4759	7136	6752	22 466	18 269	**136 020**	85 580	120 824
	32	2967	3444	3275	11 218	9200	69 322	43 933	**123 015**
	64	1394	1780	1610	5503	4780	36 509	22 805	**118 286**
	128	502	909	789	2759	2212	17 556	10 897	**119 643**
	256	432	454	398	1211	1131	8957	5816	**111 162**
	512	259	223	191	625	563	4420	2899	**113 037**
R<S[32]>[2]	16	11 209	113 681	110 279	166 588	164 009	**235 087**	156 057	119 402
	32	10 193	59 790	58 515	85 660	86 141	**115 155**	80 650	100 468
	64	5846	32 058	31 272	46 346	41 250	63 315	42 969	**105 559**
	128	2194	17 172	15 834	22 968	20 013	30 947	21 420	**103 920**
	256	984	8566	7975	8093	10 304	14 851	10 811	**110 692**
	512	598	4346	3976	5968	5170	7546	5286	**95 699**
R<S[32]>[16]	16	2989	22 660	16 397	27 744	27 634	49 038	35 326	**102 180**
	32	1534	10 768	8313	15 668	13 922	25 479	17 841	**110 535**
	64	687	5509	4296	7881	7294	12 289	8689	**110 234**
	128	240	2817	2115	2754	3502	6243	4432	**97 991**
	256	202	1529	1061	1428	1773	3063	2167	**80 352**
	512	108	748	543	711	844	1547	1092	**75 980**
Client	16	678	4188	3125	4654	4296	11 015	6396	**101 696**
	32	357	2401	1687	1961	2034	4655	3226	**95 073**
	64	217	1026	759	1109	1002	2539	1091	**88 477**
	128	74	500	396	596	538	1184	876	**68 037**
	256	62	254	196	276	248	586	426	**69 012**
	512	31	124	93	146	127	273	192	**65 686**

Bibliography

[1] Gojko Adzic and Robert Chatley. "Serverless Computing: Economic and Architectural Impact". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Paderborn Germany: ACM, Aug. 21, 2017, pp. 884–889. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3117767.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[3] Arif Ahmed, Apoorve Mohan, Gene Cooperman, and Guillaume Pierre. "Docker Container Deployment in Distributed Fog Infrastructures with Checkpoint/Restart". In: *2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. Oxford, UK: IEEE, Aug. 2020, pp. 55–62. ISBN: 978-1-7281-1035-6. DOI: 10.1109/MobileCloud48802.2020.00016.

[4] *Akamai Cloud*. Akamai. URL: https://www.akamai.com/cloud (visited on Nov. 26, 2025).

[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. "SAND: Towards High-Performance Serverless Computing". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 923–935. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/akkus.

[6] Alibaba. *Alibaba Cloud*. URL: https://www.alibabacloud.com/iat/hompepage-copy-2 (visited on Nov. 26, 2025).

[7] *Alibaba/Fastjson2*. Alibaba, Nov. 20, 2025. URL: https://github.com/alibaba/fastjson2 (visited on Nov. 20, 2025).

[8] Juncal Alonso, Leire Orue-Echevarria, Valentina Casola, Ana Isabel Torre, Maider Huarte, Eneko Osaba, and Jesus L. Lobo. "Understanding the Challenges and Novel Architectural Models of Multi-Cloud Native Applications – a Systematic Literature Review". In: *Journal of Cloud Computing* 12.1 (Jan. 12, 2023), p. 6. ISSN: 2192-113X. DOI: 10.1186/s13677-022-00367-6.

[9] Amazon. *Amazon DynamoDB*. Amazon Web Services, Inc. URL: https://aws.amazon.com/dynamodb/ (visited on Nov. 27, 2025).

[10] Amazon. *Amazon EC2 High Memory Instances*. Amazon Web Services, Inc. URL: https://aws.amazon.com/ec2/instance-types/high-memory/ (visited on Nov. 13, 2025).

[11] Amazon. *Amazon ECS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/ecs/ (visited on Nov. 26, 2025).

[12] Amazon. *Amazon Relational Database Service (RDS) - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/rds/ (visited on Nov. 27, 2025).

[13] Amazon. *Amazon S3 - Cloud Object Storage - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/s3/ (visited on Nov. 27, 2025).

[14] Amazon. *Amazon Web Services (AWS)*. Amazon Web Services, Inc. URL: https://aws.amazon.com/ (visited on Nov. 26, 2025).

[15] Amazon. *API Management - Amazon API Gateway - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/api-gateway/ (visited on Nov. 27, 2025).

[16] Amazon. *Elastic Block Storage - Amazon EBS - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/ebs/ (visited on Nov. 27, 2025).

[17] Amazon. *Managed Container Apps Service - AWS App Runner - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/apprunner/ (visited on Nov. 27, 2025).

[18] Amazon. *Managed Kubernetes - Amazon Elastic Kubernetes Service (EKS) - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/eks/ (visited on Nov. 27, 2025).

[19] Amazon. *Private Cloud - Amazon Virtual Private Cloud (VPC) - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/vpc/ (visited on Nov. 27, 2025).

[20] Amazon. *Serverless Compute - AWS Fargate - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/fargate/ (visited on Nov. 27, 2025).

[21] Amazon. *Serverless Function, FaaS Serverless - AWS Lambda - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/lambda/ (visited on Nov. 27, 2025).

[22] Amazon. *Web App Deployment - AWS Elastic Beanstalk - AWS*. Amazon Web Services, Inc. URL: https://aws.amazon.com/elasticbeanstalk/ (visited on Nov. 27, 2025).

[23] Fabio Andrijauskas, Igor Sfiligoi, Diego Davila, Aashay Arora, Jonathan Guiang, Brian Bockelman, Greg Thain, and Frank Würthwein. "CRIU - Checkpoint Restore in Userspace for Computational Simulations and Scientific Applications". In: *EPJ Web of Conferences* 295 (2024). Ed. by R. De Vita, X. Espinal, P. Laycock, and O. Shadura, p. 07046. ISSN: 2100-014X. DOI: 10.1051/epjconf/202429507046.

[24] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. "Comparing Public-Cloud Providers". In: *IEEE Internet Computing* 15.2 (Mar. 2011), pp. 50–53. ISSN: 1089-7801. DOI: 10.1109/MIC.2011.36.

[25] Jason Ansel, Kapil Arya, and Gene Cooperman. "DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. Rome: IEEE, May 2009, pp. 1–12. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161063.

[26] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. "Sprocket: A Serverless Video Processing Framework". In: *Proceedings of the ACM Symposium on Cloud Computing*. Carlsbad CA USA: ACM, Oct. 11, 2018, pp. 263–274. ISBN: 978-1-4503-6011-1. DOI: 10.1145/3267809.3267815.

[27] Lixiang Ao, George Porter, and Geoffrey M. Voelker. "FaaSnap: FaaS Made Fast Using Snapshot-Based VMs". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. Rennes France: ACM, Mar. 28, 2022, pp. 730–746. ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3524270.

[28] Apache. *Apache Mesos*. Apache Mesos. URL: https://mesos.apache.org/ (visited on Nov. 26, 2025).

[29] *Apache OpenWhisk: A Serverless, Open Source Cloud Platform*. URL: https://openwhisk.apache.org/ (visited on Nov. 26, 2025).

[30] *Apache Spark™ - Unified Engine for Large-Scale Data Analytics*. URL: https://spark.apache.org/ (visited on Nov. 22, 2025).

[31] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Melbourne Victoria Australia: ACM, May 27, 2015, pp. 1383–1394. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742797.

[32] Asana. *Asana*. URL: https://asana.com/ (visited on Nov. 28, 2025).

[33] Atlassian. *Atlassian Bitbucket: Code & CI/CD on the Atlassian Platform*. URL: https://www.atlassian.com/software/bitbucket (visited on Nov. 28, 2025).

[34] Atlassian. *Confluence*. URL: https://www.atlassian.com/software/confluence (visited on Nov. 28, 2025).

[35] Atlassian. *Jira*. URL: https://www.atlassian.com/software/jira (visited on Nov. 28, 2025).

[36] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. "Fast, Effective Dynamic Compilation". In: *ACM SIGPLAN Notices* 31.5 (May 1996), pp. 149–159. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/249069.231409.

[37] *Backblaze: Low Cost, High Performance S3 Compatible Object Storage*. URL: https://www.backblaze.com/cloud-storage (visited on Nov. 26, 2025).

[38] Arshdeep Bahga and Vijay K. Madisetti. *Cloud Computing: A Hands-on Approach*. s.l, 2014. 454 pp. ISBN: 978-1-4944-3514-1.

[39] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software* 33.3 (May 2016), pp. 42–52. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2016.64.

[40] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Migrating to Cloud-Native Architectures Using Microservices: An Experience Report". In: *Advances in Service-Oriented and Cloud Computing*. Ed. by Antonio Celesti and Philipp Leitner. Vol. 567. Cham: Springer International Publishing, 2016, pp. 201–215. ISBN: 978-3-319-33312-0. DOI: 10.1007/978-3-319-33313-7_15.

[41] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. "The Serverless Trilemma: Function Composition for Serverless Computing". In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Vancouver BC Canada: ACM, Oct. 25, 2017, pp. 89–103. ISBN: 978-1-4503-5530-8. DOI: 10.1145/3133850.3133855.

[42] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. "Stateful Serverless Computing with Crucial". In: *ACM Transactions on Software Engineering and Methodology* 31.3 (July 31, 2022), pp. 1–38. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3490386.

[43] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the Art of Virtualization". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. Bolton Landing NY USA: ACM, Oct. 19, 2003, pp. 164–177. ISBN: 978-1-58113-757-6. DOI: 10.1145/945445.945462.

[44] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. "Virtual Machine Warmup Blows Hot and Cold". In: *Proceedings of the ACM on Programming Languages* 1 (OOPSLA Oct. 12, 2017), pp. 1–27. ISSN: 2475-1421. DOI: 10.1145/3133876.

[45] Leonardo Bautista-Gomez, Anne Benoit, Sheng Di, Thomas Herault, Yves Robert, and Hongyang Sun. "A Survey on Checkpointing Strategies: Should We Always Checkpoint à La Young/Daly?" In: *Future Generation Computer Systems* 161 (Dec. 2024), pp. 315–328. ISSN: 0167739X. DOI: 10.1016/j.future.2024.07.022.

[46] Loris Belcastro, Jesus Carretero, and Domenico Talia. "Edge-Cloud Solutions for Big Data Analysis and Distributed Machine Learning - 1". In: *Future Generation Computer Systems* 159 (Oct. 2024), pp. 323–326. ISSN: 0167739X. DOI: 10.1016/j.future.2024.05.023.

[47] Jonathan Bell and Luís Pina. "CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs". In: *LIPIcs, Volume 109, ECOOP 2018* 109 (2018). Ed. by Todd Millstein, 17:1–17:31. ISSN: 1868-8969. DOI: 10.4230/LIPICS.ECOOP.2018.17.

[48] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator." In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 46. California, USA, 2005, pp. 10–55.

[49] *Bencode Specification*. URL: https://bittorrent.org/beps/bep_0003.html (visited on Nov. 21, 2025).

[50] Alexander Benlian, Thomas Hess, and Peter Buxmann. "Drivers of SaaS-Adoption – An Empirical Study of Different Application Types". In: *Business & Information Systems Engineering* 1.5 (Oct. 2009), pp. 357–369. ISSN: 1867-0202. DOI: 10.1007/s12599-009-0068-x.

[51] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Reliable Benchmarking: Requirements and Solutions". In: *International Journal on Software Tools for Technology Transfer* 21.1 (Feb. 6, 2019), pp. 1–29. ISSN: 1433-2779, 1433-2787. DOI: 10.1007/s10009-017-0469-y.

[52] Philip Bianco, Rick Kotermanski, and Paulo F. Merson. "Evaluating a Service-Oriented Architecture". In: (2007), 2086489 Bytes. DOI: 10.1184/R1/6573479.V1.

[53] Andrew D. Birrell and Bruce Jay Nelson. "Implementing Remote Procedure Calls". In: *ACM Transactions on Computer Systems* 2.1 (Feb. 1984), pp. 39–59. ISSN: 0734-2071, 1557-7333. DOI: 10.1145/2080.357392.

[54] *BitTorrent.Org*. URL: https://www.bittorrent.org/index.html (visited on Nov. 21, 2025).

[55] Joshua Bloch. *Effective Java*. Third edition. Boston: Addison-Wesley, 2018. 1 p. ISBN: 978-0-13-468609-7.

[56] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. "Putting the Micro Back in Microservice". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 645–650.

[57] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. 3rd ed. Sebastopol: O'Reilly Media, Inc, 2008. 944 pp. ISBN: 978-0-596-00565-8.

[58] Don Box and Chris Sells. "Essential .NET. 1: The Common Language Runtime". In: Boston, Mass. Munich: Addison-Wesley, 2002. ISBN: 978-0-201-73411-9.

[59] Gilad Bracha. *The Dart Programming Language*. Boston Munich: Addison-Wesley, 2016. 201 pp. ISBN: 978-0-321-92770-5.

[60] Mike Brown, Hersey Cartwright, Martin Gavanda, Andrea Mauro, Karel Novak, and Paolo Valsecchi. *The Complete VMware vSphere Guide: Design a Virtualized Data Center with VMware vSphere 6.7*. Erscheinungsort nicht ermittelbar: Packt Publishing, 2019. 754 pp. ISBN: 978-1-83898-575-2.

[61] Rodrigo Bruno and Paulo Ferreira. "ALMA: GC-assisted JVM Live Migration for Java Server Applications". In: *Proceedings of the 17th International Middleware Conference*. Trento Italy: ACM, Nov. 28, 2016, pp. 1–14. ISBN: 978-1-4503-4300-8. DOI: 10.1145/2988336.2988341.

[62] Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. "Compiler-assisted object inlining with value fields". In: ACM, 2021, pp. 128–141. DOI: 10.1145/3453483.3454034. URL: https://doi.org/10.1145/3453483.3454034.

[63] *BSON (Binary JSON): Specification*. URL: https://bsonspec.org/spec.html (visited on Nov. 19, 2025).

[64] Umer Ahmed Butt, Rashid Amin, Muhammad Mehmood, Hamza Aldabbas, Mafawez T. Alharbi, and Nasser Albaqami. "Cloud Security Threats and Solutions: A Survey". In: *Wireless Personal Communications* 128.1 (Jan. 2023), pp. 387–413. ISSN: 0929-6212, 1572-834X. DOI: 10.1007/s11277-022-09960-z.

[65] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. "SEUSS: Skip Redundant Paths to Make Serverless Fast". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. Heraklion Greece: ACM, Apr. 15, 2020, pp. 1–15. ISBN: 978-1-4503-6882-7. DOI: 10.1145/3342195.3392698.

[66] David Calavera and Lorenzo Fontana. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking*. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2020. 1 p. ISBN: 978-1-4920-5020-9.

[67] *Cap'n Proto*. URL: https://capnproto.org/ (visited on Nov. 21, 2025).

[68] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. "State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing". In: *Proceedings of the VLDB Endowment* 10.12 (Aug. 2017), pp. 1718–1729. ISSN: 2150-8097. DOI: 10.14778/3137765.3137777.

[69] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. "Cirrus: A Serverless Framework for End-to-end ML Workflows". In: *Proceedings of the ACM Symposium on Cloud Computing*. Santa Cruz CA USA: ACM, Nov. 20, 2019, pp. 13–24. ISBN: 978-1-4503-6973-2. DOI: 10.1145/3357223.3362711.

[70] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. "From warm to hot starts". In: ACM, 2021, pp. 58–64. DOI: 10.1145/3458336.3465305. URL: https://doi.org/10.1145/3458336.3465305.

[71] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. "The Rise of Serverless Computing". In: *Communications of the ACM* 62.12 (Nov. 21, 2019), pp. 44–54. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3368454.

[72] Yang Chen. "Checkpoint and Restore of Micro-service in Docker Containers". In: *Proceedings of the 3rd International Conference on Mechatronics and Industrial Informatics*. Zhuhai, China: Atlantis Press, 2015. ISBN: 978-94-6252-131-5. DOI: 10.2991/icmii-15.2015.160.

[73] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Open Source Software Development Series. Upper Saddle River, N.J: Prentice Hall, 2008. 286 pp. ISBN: 978-0-13-234971-0.

[74] Susanta Nanda Tzi-cker Chiueh and Stony Brook. "A Survey on Virtualization Technologies". In: *Rpe Report* 142 (2005).

[75] Sang-Hoon Choi and Ki-Woong Park. "iContainer: Consecutive Checkpointing with Rapid Resilience for Immortal Container-Based Services". In: *Journal of Network and Computer Applications* 208 (Dec. 2022), p. 103494. ISSN: 10848045. DOI: 10.1016/j.jnca.2022.103494.

[76] Chih Chieh Chou, Yuan Chen, Dejan Milojicic, Narasimha Reddy, and Paul Gratz. "Optimizing Post-Copy Live Migration with System-Level Checkpoint Using Fabric-Attached Memory". In: *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. Denver, CO, USA: IEEE, Nov. 2019, pp. 16–24. ISBN: 978-1-7281-6007-8. DOI: 10.1109/MCHPC49590.2019.00010.

[77] Cisco. *Cisco Cloud Solutions*. Cisco. URL: https://www.cisco.com/c/en/us/solutions/cloud/index.html (visited on Nov. 26, 2025).

[78] Cisco. *Webex*. URL: https://www.webex.com/ (visited on Nov. 28, 2025).

[79] *Cloud Native Computing Foundation*. CNCF. URL: https://www.cncf.io/ (visited on Nov. 29, 2025).

[80] Cloudflare Inc. *Cloudflare Workers Platform*. URL: https://workers.cloudflare.com/ (visited on Nov. 19, 2025).

[81] Codeberg. *Codeberg.Org*. URL: https://codeberg.org/ (visited on Nov. 28, 2025).

[82] Harry Colvin. *VirtualBox: An Ultimate Guide Book on Virtualization with VirtualBox*. S.l.: Eisenbrauns, 2015. ISBN: 978-1-5227-6988-0.

[83] Douglas Comer. *The Cloud Computing Book: The Future of Computing Explained*. First edition. Boca Raton London New York: CRC Press, Taylor & Francis Group, 2021. 1 p. ISBN: 978-0-367-70680-7.

[84] Autonomic Computing et al. "An Architectural Blueprint for Autonomic Computing". In: *IBM White Paper* 31.2006 (2006), pp. 1–6.

[85] Gene Cooperman, Jason Ansel, and Xiaoqin Ma. "Adaptive Checkpointing for Master-Worker Style Parallelism". In: *2005 IEEE International Conference on Cluster Computing*. Burlington, MA, USA: IEEE, Sept. 2005, pp. 1–2. ISBN: 978-0-7803-9485-8. DOI: 10.1109/CLUSTR.2005.347096.

[86] *CRIU*. URL: https://criu.org/ (visited on Nov. 14, 2025).

[87] Milan Čugurović, Ivan Ristović, Strahinja Stanojević, Marko Spasić, Vesna Marinković, and Milena Vujosević Janicić. "ML-driven Prediction of Optimal CFG Traversal Algorithm in Modern JVM-based Applications". In: *Proceedings of the 12th International Conference on Electrical, Electronic and Computing Engineering*. June 2025. DOI: 10.1109/IcETRAN66854.2025.11114103.

[88] Milan Čugurović, Milena Vujošević Janičić, Vojin Jovanović, and Thomas Würthinger. "GraalSP: Polyglot, Efficient, and Robust Machine Learning-Based Static Profiler". In: *Journal of Systems and Software* 213 (July 2024), p. 112058. ISSN: 01641212. DOI: 10.1016/j.jss.2024.112058.

[89] Michael Cusumano. "Cloud Computing and SaaS as New Computing Platforms". In: *Communications of the ACM* 53.4 (Apr. 2010), pp. 27–29. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1721654.1721667.

[90] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. "Event-Driven Programming for Robust Software". In: *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop: Beyond the PC - EW10*. Saint-Emilion, France: ACM Press, 2002, p. 186. DOI: 10.1145/1133373.1133410.

[91] Dargslan. *Linux Virtualization Stack: QEMU, KVM, Libvirt, and Virt-Manager*. Kindle Edition. Dargslan s.r.o., June 7, 2025.

[92] Datadog. *State of Containers and Serverless*. 0:00:00 -0500 EST. URL: https://www.datadoghq.com/state-of-containers-and-serverless/ (visited on Jan. 4, 2026).

[93] Yaniv David, Neophytos Christou, Andreas D. Kellas, Vasileios P. Kemerlis, and Junfeng Yang. "QUACK: Hindering Deserialization Attacks via Static Duck Typing". In: *Proceedings 2024 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2024. ISBN: 978-1-891562-93-8. DOI: 10.14722/ndss.2024.241015.

[94] Lorenzo De Lauretis. "From Monolithic Architecture to Microservices Architecture". In: *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. Berlin, Germany: IEEE, Oct. 2019, pp. 93–96. ISBN: 978-1-7281-5138-0. DOI: 10.1109/ISSREW.2019.00050.

[95] Dell. *Dell Cloud Solutions*. Dell. URL: https://www.dell.com/en-us/lp/dt/cloud-computing (visited on Nov. 26, 2025).

[96] Peter J. Denning. "Virtual Memory". In: *ACM Computing Surveys* 2.3 (Sept. 1970), pp. 153–189. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/356571.356573.

[97] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. "Garbage-First Garbage Collection". In: *Proceedings of the 4th International Symposium on Memory Management*. Vancouver BC Canada: ACM, Oct. 24, 2004, pp. 37–48. ISBN: 978-1-58113-945-7. DOI: 10.1145/1029873.1029879.

[98] Sheng Di, Yves Robert, Frédéric Vivien, Derrick Kondo, Cho-Li Wang, and Franck Cappello. "Optimization of Cloud Task Processing with Checkpoint-Restart Mechanism". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Denver Colorado: ACM, Nov. 17, 2013, pp. 1–12. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503217.

[99] *DigitalOcean | The Unified Agentic Cloud*. URL: https://www.digitalocean.com/ (visited on Nov. 26, 2025).

[100] Nicola Dimitri. "Pricing Cloud IaaS Computing Services". In: *Journal of Cloud Computing* 9.1 (Dec. 2020), p. 14. ISSN: 2192-113X. DOI: 10.1186/s13677-020-00161-2.

[101] Namiot Dmitry and Sneps-Sneppe Manfred. "On Micro-Services Architecture". In: *International Journal of Open Information Technologies* 2.9 (2014), pp. 24–27.

[102] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. "Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Lausanne Switzerland: ACM, Mar. 9, 2020, pp. 467–481. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378512.

[103] Yunjie Du, Xuanhua Shi, Hai Jin, Song Wu, and Laurence T. Yang. "FITDOC: Fast Virtual Machines Checkpointing with Delta Memory Compression". In: *The Journal of Supercomputing* 72.9 (Sept. 2016), pp. 3328–3347. ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-015-1429-5.

[104] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. "Speculation without Regret: Reducing Deoptimization Meta-Data in the Graal Compiler". In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. Cracow Poland: ACM, Sept. 23, 2014, pp. 187–193. ISBN: 978-1-4503-2926-2. DOI: 10.1145/2647508.2647521.

[105] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. "An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler". In: *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*. Indianapolis Indiana USA: ACM, Oct. 28, 2013, pp. 1–10. ISBN: 978-1-4503-2601-8. DOI: 10.1145/2542142.2542143.

[106] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. "Photons: Lambdas on a Diet". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. Virtual Event USA: ACM, Oct. 12, 2020, pp. 45–59. ISBN: 978-1-4503-8137-6. DOI: 10.1145/3419111.3421297.

[107] Pranay Dutta, Prashant Dutta, and Xoriant, Pune, Maharashtra, India. "Comparative Study of Cloud Services Offered by Amazon, Microsoft and Google". In: *International Journal of Trend in Scientific Research and Development* Volume-3 (Issue-3 Apr. 30, 2019), pp. 981–985. ISSN: 2456-6470. DOI: 10.31142/ijtsrd23170.

[108] Bernhard Egger, Younghyun Cho, Changyeon Jo, Eunbyun Park, and Jaejin Lee. "Efficient Checkpointing of Live Virtual Machines". In: *IEEE Transactions on Computers* 65.10 (Oct. 1, 2016), pp. 3041–3054. ISSN: 0018-9340. DOI: 10.1109/TC.2016.2519890.

[109] Niklas Eiling, Jonas Baude, Stefan Lankes, and Antonello Monti. "Cricket: A Virtualization Layer for Distributed Execution of CUDA Applications with Checkpoint/Restart Support". In: *Concurrency and Computation: Practice and Experience* 34.14 (June 25, 2022), e6474. ISSN: 1532-0626, 1532-0634. DOI: 10.1002/cpe.6474.

[110] Niklas Eiling, Stefan Lankes, and Antonello Monti. "Checkpoint/Restart for CUDA Kernels". In: *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. Denver CO USA: ACM, Nov. 12, 2023, pp. 1729–1737. ISBN: 979-8-4007-0785-8. DOI: 10.1145/3624062.3624254.

[111] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. "The State of Serverless Applications: Collection, Characterization, and Community Consensus". In: *IEEE Transactions on Software Engineering* 48.10 (Oct. 1, 2022), pp. 4152–4166. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2021.3113940.

[112] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. *A Review of Serverless Use Cases and Their Characteristics*. Version 2. 2020. DOI: 10.48550/ARXIV.2008.11110. Pre-published.

[113] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. "Serverless Applications: Why, When, and How?" In: *IEEE Software* 38.1 (Jan. 2021), pp. 32–39. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2020.3023302.

[114] Thomas Erl. *Service-Oriented Architecture: Analysis and Design for Services and Microservices*. Second edition. The Prentice Hall Service Technology Series from Thomas Erl. Upper Saddle River, NJ: Prentice Hall, 2017. 393 pp. ISBN: 978-0-13-385858-7.

[115] Thomas Erl and Eric Barceló Monroy. *Cloud Computing: Concepts, Technology, Security & Architecture*. Second edition. The Pearson Digital Enterprise Series from Thomas Erl. Hoboken, NJ Toronto London: Pearson, 2024. 559 pp. ISBN: 978-0-13-805225-6.

[116] EsotericSoftware. *EsotericSoftware/Kryo: Java Binary Serialization and Cloning: Fast, Efficient, Automatic*. 2008. URL: https://github.com/EsotericSoftware/kryo.

[117] Eric Evans and Martin Fowler. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Upper Saddle River, NJ: Addison-Wesley, 2019. 529 pp. ISBN: 978-0-321-12521-7.

[118] *Extensible Data Notation (EDN)*. edn-format, Nov. 20, 2025. URL: https://github.com/edn-format/edn (visited on Nov. 20, 2025).

[119] *Extensible Markup Language (XML)*. URL: https://www.w3.org/XML/ (visited on Nov. 19, 2025).

[120] *FasterXML/Jackson*. FasterXML, LLC, Nov. 19, 2025. URL: https://github.com/FasterXML/jackson (visited on Nov. 20, 2025).

[121] Michael C. Feathers. *Working Effectively with Legacy Code*. 14. print. Robert C. Martin Series. Upper Saddle River, N.J: Prentice Hall, 2013. 434 pp. ISBN: 978-0-13-117705-5.

[122] Kurt B. Ferreira, Rolf Riesen, Ron Brighwell, Patrick Bridges, and Dorian Arnold. "Libhashckpt: Hash-Based Incremental Checkpointing Using GPU's". In: *Recent Advances in the Message Passing Interface*. Ed. by Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra. Vol. 6960. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 272–281. ISBN: 978-3-642-24448-3. DOI: 10.1007/978-3-642-24449-0_31.

[123] *Fine-Grained Sandboxing with V8 Isolates*. InfoQ. URL: https://www.infoq.com/presentations/cloudflare-v8/ (visited on Nov. 19, 2025).

[124] Daniel Fireman, Paulo Silva, Thiago Emmanuel Pereira, Luis Mafra, and Dalton Valadares. "Prebaking Runtime Environments to Improve the FaaS Cold Start Latency". In: *Future Generation Computer Systems* 155 (June 2024), pp. 287–299. ISSN: 0167739X. DOI: 10.1016/j.future.2024.01.019.

[125] David Flanagan. *JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language*. Seventh edition. Beijing [China] Sebastopol, CA: O'Reilly Media, Inc, 2020. 1 p. ISBN: 978-1-4919-5202-3.

[126] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 363–376. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[127] Martin Fowler. *Patterns of Enterprise Application Architecture*. Nineteenth printing. The Addison-Wesley Signature Series. Boston San Francisco New York Toronto Montreal London Munich Paris Madrid Capetown: Addison-Wesley, 2013. 533 pp. ISBN: 978-0-321-12742-6.

[128] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. 28. printing. The Addison-Wesley Object Technology Series. Boston: Addison-Wesley, 2013. 431 pp. ISBN: 978-0-201-48567-7.

[129] Abhilash G B. *VMware vSphere 6.7 Cookbook: Practical Recipes to Deploy, Configure, and Manage VMware vSphere 6.7 Components, 4th Edition*. 4th ed. Birmingham: Packt Publishing, 2019. 1 p. ISBN: 978-1-78995-300-8.

[130] Rohan Garg, Komal Sodha, and Gene Cooperman. *A Generic Checkpoint-Restart Mechanism for Virtual Machines*. Dec. 8, 2012. DOI: 10.48550/arXiv.1212.1787. arXiv: 1212.1787 [cs]. Pre-published.

[131] Taha Gharaibeh, Steven Seiden, Mohamed Abouelsaoud, Elias Bou-Harb, and Ibrahim Baggili. "Don't, Stop, Drop, Pause: Forensics of CONtainer CheckPOINTs (ConPoint)". In: *Proceedings of the 19th International Conference on Availability, Reliability and Security*. Vienna Austria: ACM, July 30, 2024, pp. 1–11. ISBN: 979-8-4007-1718-5. DOI: 10.1145/3664476.3670895.

[132] *GitHub*. 2025. URL: https://github.com/ (visited on Nov. 28, 2025).

[133] *GitLab*. URL: https://gitlab.com/ (visited on Nov. 28, 2025).

[134] Tod Golding. *Building Multi-Tenant SaaS Architectures*. 1st ed. Sebastopol: O'Reilly Media, Incorporated, 2024. 1 p. ISBN: 978-1-0981-4064-9.

[135] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, and Ion Stoica. "Graphx: Graph Processing in a Distributed Dataflow Framework". In: *OSDI* (Jan. 2014), pp. 599–613.

[136] Google. *Cloud Run Functions*. Google Cloud. URL: https://cloud.google.com/functions (visited on Nov. 28, 2025).

[137] Google. *Custom Startup Snapshots*. online. Sept. 2015. URL: https://v8.dev/blog/custom-startup-snapshots.

[138] Google. *Dart Language Snapshots*. online. 2019. URL: https://dart.dev/tools/dart-compile.

[139] Google. *FlatBuffers*. URL: https://flatbuffers.dev/ (visited on Nov. 21, 2025).

[140] Google. *Gmail*. Gmail. URL: https://mail.google.com/ (visited on Dec. 27, 2025).

[141] Google. *Google Cloud*. URL: https://cloud.google.com/ (visited on Nov. 26, 2025).

[142] Google. *Google Docs*. Google Docs. URL: https://workspace.google.com/products/docs/ (visited on Dec. 27, 2025).

[143] Google. *Google Drive*. URL: https://drive.google.com/ (visited on Nov. 28, 2025).

[144] Google. *Protocol Buffers*. URL: https://protobuf.dev/ (visited on Nov. 21, 2025).

[145] *Google Meet: Online Web and Video Conferencing Calls*. Google Workspace. URL: https://meet.google.com (visited on Nov. 28, 2025).

[146] *Google/Gson*. Google, Nov. 20, 2025. URL: https://github.com/google/gson (visited on Nov. 20, 2025).

[147] Pierre Graux, Jean-François Lalande, Valérie Viet Triem Tong, and Pierre Wilke. "Preventing Serialization Vulnerabilities through Transient Field Detection". In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. Virtual Event Republic of Korea: ACM, Mar. 22, 2021, pp. 1598–1606. ISBN: 978-1-4503-8104-8. DOI: 10.1145/3412841.3442033.

[148] *gRPC*. gRPC. URL: https://grpc.io/ (visited on Nov. 21, 2025).

[149] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jf Bastien. "Bringing the Web up to Speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Barcelona Spain: ACM, June 14, 2017, pp. 185–200. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062363.

[150] Paul H Hargrove and Jason C Duell. "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters". In: *Journal of Physics: Conference Series* 46 (Sept. 1, 2006), pp. 494–499. ISSN: 1742-6588, 1742-6596. DOI: 10.1088/1742-6596/46/1/067.

[151] Hassan B. Hassan, Saman A. Barakat, and Qusay I. Sarhan. "Survey on Serverless Computing". In: *Journal of Cloud Computing* 10.1 (July 12, 2021), p. 39. ISSN: 2192-113X. DOI: 10.1186/s13677-021-00253-7.

[152] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Serverless Computation with OpenLambda". In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, June 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[153] Matt et. al Heon, Dan Walsh, Brent Baude, Urvashi Mohnani, Ashley Cui, Tom Sweeney, Giuseppe Scrivano, Chris Evich, Valentin Rothberg, Miloslav Trmač, Jhon Honce, Qi Wang, Lokesh Mandvekar, Adrian Reber, Eduardo Santiago, Sascha Grunert, Nalin Dahyabhai, Anders Bjorklund, Kunal Kushwaha, Sujil Ashwin Sha, Yiqiao Pu, Zhangguanzhang, Matej Vasek, and Podman Community. *Podman: A Tool for Managing OCI Containers and Pods*. Version v1.0 and beyond. Currently at v3.0.1. Zenodo, Jan. 11, 2018. DOI: 10.5281/ZENODO.4735634.

[154] Kai-Yuan Hou, Kang G. Shin, and Jan-Lung Sung. "Application-Assisted Live Migration of Virtual Machines with Java Applications". In: *Proceedings of the Tenth European Conference on Computer Systems*. Bordeaux France: ACM, Apr. 17, 2015, pp. 1–15. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741950.

[155] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, Mengting Lu, Tao Ma, Haifeng Gong, and YongWei Wu. "TrEnv: Transparently Share Serverless Execution Environments Across Different Functions and Nodes". In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. Austin TX USA: ACM, Nov. 4, 2024, pp. 421–437. ISBN: 979-8-4007-1251-7. DOI: 10.1145/3694715.3695967.

[156] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. "A Domain-Specific Language for Building Self-Optimizing AST Interpreters". In: *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences*. Västerås Sweden: ACM, Sept. 15, 2014, pp. 123–132. ISBN: 978-1-4503-3161-6. DOI: 10.1145/2658761.2658776.

[157] Judith Hurwitz and Daniel Kirsch. *Cloud Computing for Dummies*. 2nd edition. For Dummies. Hoboken: Wiley, 2020. 1 p. ISBN: 978-1-119-54665-8.

[158] Mohamed K. Hussein, Mohamed H. Mousa, and Mohamed A. Alqarni. "A Placement Architecture for a Container as a Service (CaaS) in a Cloud Environment". In: *Journal of Cloud Computing* 8.1 (Dec. 2019), p. 7. ISSN: 2192-113X. DOI: 10.1186/s13677-019-0131-1.

[159] David Hutchins. *Just in Time*. Repr. Aldershot, Hants: Gower Technical Pr, 1990. 215 pp. ISBN: 978-0-291-39751-5.

[160] IBM. *IBM Cloud*. URL: https://www.ibm.com/solutions/cloud (visited on Nov. 26, 2025).

[161] Daniel Ingalls. "The Evolution of Smalltalk: From Smalltalk-72 through Squeak". In: *Proceedings of the ACM on Programming Languages* 4 (HOPL June 14, 2020), pp. 1–101. ISSN: 2475-1421. DOI: 10.1145/3386335.

[162] Joseph Ingeno. *Software Architect's Handbook: Become a Successful Software Architect by Implementing Effective Architecture Concepts*. 1st ed. Place of publication not identified: Packt Publishing, 2018. 1 p. ISBN: 978-1-78862-406-0.

[163] Alexandru Iosup, Radu Prodan, and Dick Epema. "IaaS Cloud Benchmarking: Approaches, Challenges, and Experience". In: *Cloud Computing for Data-Intensive Applications*. Ed. by Xiaolin Li and Judy Qiu. New York, NY: Springer New York, 2014, pp. 83–104. ISBN: 978-1-4939-1904-8. DOI: 10.1007/978-1-4939-1905-5_4.

[164] *Iron.Io Open Source - Functions*. URL: https://open.iron.io/ (visited on Nov. 28, 2025).

[165] *ISO - International Organization for Standardization*. ISO. Nov. 21, 2025. URL: https://www.iso.org/home.html (visited on Nov. 23, 2025).

[166] *ISO/IEC 22123-1:2023*. ISO. URL: https://www.iso.org/standard/82758.html (visited on Nov. 23, 2025).

[167] *ISO/IEC 22123-2:2023*. ISO. URL: https://www.iso.org/standard/80351.html (visited on Nov. 23, 2025).

[168] Serhii Ivanenko, Vasyl Lanko, Rudi Horn, Vojin Jovanovic, and Rodrigo Bruno. *Hydra: Virtualized Multi-Language Runtime for High-Density Serverless Platforms*. Version 3. 2022. DOI: 10.48550/ARXIV.2212.10131. Pre-published.

[169] David Jackson and Gary Clynch. "An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. Zurich: IEEE, Dec. 2018, pp. 154–160. ISBN: 978-1-7281-0359-4. DOI: 10.1109/UCC-Companion.2018.00050.

[170] Joab Jackson. *Return of the Monolith: Amazon Dumps Microservices for Video Monitoring*. The New Stack. May 4, 2023. URL: https://thenewstack.io/return-of-the-monolith-amazon-dumps-microservices-for-video-monitoring/ (visited on Nov. 28, 2025).

[171] Nancy Jain and Sakshi Choudhary. "Overview of Virtualization in Cloud Computing". In: *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. Indore, Madhya Pradesh, India: IEEE, Mar. 2016, pp. 1–4. ISBN: 978-1-5090-0669-4. DOI: 10.1109/CDAN.2016.7570950.

[172] Shashank Mohan Jain. *Linux Containers and Virtualization: A Kernel Perspective*. Berkeley, CA: Apress L. P, 2020. 1 p. ISBN: 978-1-4842-6282-5.

[173] Michał Tomasz Jakóbczyk. *Practical Oracle Cloud Infrastructure: Infrastructure as a Service, Autonomous Database, Managed Kubernetes, and Serverless*. Berkeley, CA: Apress, 2020. ISBN: 978-1-4842-5505-6. DOI: 10.1007/978-1-4842-5506-3.

[174] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. "A Specialized Architecture for Object Serialization with Applications to Big Data Analytics". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, May 2020, pp. 322–334. ISBN: 978-1-7281-4661-4. DOI: 10.1109/ISCA45697.2020.00036.

[175] Ludvig Janiuk. *Direct Heap Snapshotting in the Java HotSpot VM: A Prototype*. Master's thesis. Stockholm, Sweden: KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, 2020.

[176] *Jenkins*. URL: https://www.jenkins.io/ (visited on Nov. 28, 2025).

[177] *JEP 243: Java-Level JVM Compiler Interface*. URL: https://openjdk.org/jeps/243 (visited on Dec. 3, 2025).

[178] Zhipeng Jia and Emmett Witchel. "Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Virtual USA: ACM, Apr. 19, 2021, pp. 152–166. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446701.

[179] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. "Towards Demystifying Serverless Machine Learning Training". In: *Proceedings of the 2021 International Conference on Management of Data*. Virtual Event China: ACM, June 9, 2021, pp. 857–871. ISBN: 978-1-4503-8343-1. DOI: 10.1145/3448016.3459240.

[180] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Version 1. 2019. DOI: 10.48550/ARXIV.1902.03383. Pre-published.

[181] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. 2nd ed. New York: Chapman and Hall/CRC, June 1, 2023. ISBN: 978-1-003-27614-2. DOI: 10.1201/9781003276142.

[182] *JSON*. URL: https://www.json.org/json-en.html (visited on Nov. 19, 2025).

[183] Paulo Souza Junior, Daniele Miorandi, and Guillaume Pierre. "Good Shepherds Care For Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes". In: *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*. Messina, Italy: IEEE, May 2022, pp. 26–33. ISBN: 978-1-6654-9524-0. DOI: 10.1109/ICFEC54809.2022.00011.

[184] Dharmesh Kakadia. *Apache Mesos Essentials: Build and Execute Robust and Scalable Applications Using Apache Mesos*. Birmingham, UK: Packt Publishing, 2015. ISBN: 978-1-78328-876-2.

[185] Kamatera. *Kamatera Cloud*. Kamatera. URL: https://www.kamatera.com/ (visited on Nov. 26, 2025).

[186] Balachandra Reddy Kandukuri, Ramakrishna Paturi V., and Atanu Rakshit. "Cloud Security Issues". In: *2009 IEEE International Conference on Services Computing*. Bangalore, India: IEEE, 2009, pp. 517–520. ISBN: 978-1-4244-5183-8. DOI: 10.1109/SCC.2009.84.

[187] Işıl Karabey Aksakalli, Turgay Çelik, Ahmet Burak Can, and Bedir Tekinerdoğan. "Deployment and Communication Patterns in Microservice Architectures: A Systematic Literature Review". In: *Journal of Systems and Software* 180 (Oct. 2021), p. 111014. ISSN: 01641212. DOI: 10.1016/j.jss.2021.111014.

[188] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. "A Hardware Accelerator for Protocol Buffers". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Virtual Event Greece: ACM, Oct. 18, 2021, pp. 462–478. ISBN: 978-1-4503-8557-2. DOI: 10.1145/3466752.3480051.

[189] Milica Karličić, Ivan Ristović, and Milena Vujošević Janičić. "Metrics Visualization Using Profiling-Based Adaptive GC Policy for Serverless". In: *Proceedings of the 14th Symposium "Mathematics and Applications"*. Dec. 2025.

[190] Milica Karličić, Ivan Ristović, and Milena Vujošević Janičić. "Profiling-Based Adaptive GC Policy for Serverless". In: *Proceedings of the 14th Symposium "Mathematics and Applications"*. Dec. 2024.

[191] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. "Instant OS Updates via Userspace Checkpoint-and-Restart". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 605–619. ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/kashyap.

[192] Michael Kavis. *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS)*. Wiley CIO Series. Hoboken, N.J: Wiley, 2014. 1 p. ISBN: 978-1-118-61761-8.

[193] Kiyokuni Kawachiya, Kazunori Ogata, Daniel Silva, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. "Cloneable JVM: A New Approach to Start Isolated Java Applications Faster". In: *Proceedings of the 3rd International Conference on Virtual Execution Environments*. San Diego California USA: ACM, June 13, 2007, pp. 1–11. ISBN: 978-1-59593-630-1. DOI: 10.1145/1254810.1254812.

[194] Andrew J. Kennedy. "Functional Pearl Pickler Combinators". In: *Journal of Functional Programming* 14.6 (Nov. 2004), pp. 727–739. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796804005209.

[195] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. "Kvm: The Linux Virtual Machine Monitor". In: *Proceedings of the Linux Symposium*. Vol. 1. 8. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[196] Andi Kleen. "A NUMA API for Linux". In: *Novel Inc* (2005).

[197] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems*. " O'Reilly Media, Inc.", 2017.

[198] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. "Understanding Ephemeral Storage for Serverless Analytics". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 789–794. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/klimovic-serverless.

[199] *Knative*. URL: https://knative.dev/ (visited on Nov. 26, 2025).

[200] Gary D. Knott. *Interpreting LISP: Programming and Data Structures*. 2nd ed. Springer-Link Bücher. Berkeley, CA: Apress, 2017. 150 pp. ISBN: 978-1-4842-2706-0. DOI: 10.1007/978-1-4842-2707-7.

[201] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. "Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts". In: ACM, 2024, pp. 298–316. DOI: 10.1145/3627703.3629556. URL: https://doi.org/10.1145/3627703.3629556.

[202] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. "Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts". In: *Proceedings of the Nineteenth European Conference on Computer Systems*. Athens Greece: ACM, Apr. 22, 2024, pp. 298–316. ISBN: 979-8-4007-0437-6. DOI: 10.1145/3627703.3629556.

[203] Kirill Kolyshkin. "Virtualization in Linux". In: *White paper, OpenVZ* 3.39 (2006), p. 8.

[204] David Kozak, Vojin Jovanovic, Codrut Stancu, Tomáš Vojnar, and Christian Wimmer. "Comparing Rapid Type Analysis with Points-To Analysis in GraalVM Native Image". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. Cascais Portugal: ACM, Oct. 19, 2023, pp. 129–142. ISBN: 979-8-4007-0380-5. DOI: 10.1145/3617651.3622980.

[205] David Kozak, Codrut Stancu, Tomáš Vojnar, and Christian Wimmer. "SkipFlow: Improving the Precision of Points-to Analysis Using Primitive Values and Predicate Edges". In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. Las Vegas NV USA: ACM, Mar. 2025, pp. 347–361. ISBN: 979-8-4007-1275-3. DOI: 10.1145/3696443.3708932.

[206] Stefan Kraemer, Rainer Leupers, Dietmar Petras, and Thomas Philipp. "A Checkpoint/Restore Framework for Systemc-Based Virtual Platforms". In: *2009 International Symposium on System-on-Chip*. Tampere, Finland: IEEE, Oct. 2009, pp. 161–167. ISBN: 978-1-4244-4465-6. DOI: 10.1109/SOCC.2009.5335656.

[207] Nane Kratzke and Peter-Christian Quint. "Understanding Cloud-Native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study". In: *Journal of Systems and Software* 126 (Apr. 2017), pp. 1–16. ISSN: 01641212. DOI: 10.1016/j.jss.2017.01.001.

[208] Kyriakos Kritikos and Pawel Skrzypek. "A Review of Serverless Frameworks". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. Zurich: IEEE, Dec. 2018, pp. 161–168. ISBN: 978-1-7281-0359-4. DOI: 10.1109/UCC-Companion.2018.00051.

[209] *Kubernetes: Production-Grade Container Orchestration*. Kubernetes. URL: https://kubernetes.io/ (visited on Nov. 26, 2025).

[210] Jörn Kuhlenkamp, Sebastian Werner, Maria C. Borges, Karim El Tal, and Stefan Tai. "An Evaluation of FaaS Platforms as a Foundation for Serverless Big Data Processing". In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. Auckland New Zealand: ACM, Dec. 2, 2019, pp. 1–9. ISBN: 978-1-4503-6894-0. DOI: 10.1145/3344341.3368796.

[211] Gurudatt Kulkarni. "Cloud Computing-Software as Service". In: *International Journal of Cloud Computing and Services Science (IJ-CLOSER)* 1.1 (Jan. 29, 2012), pp. 11–16. ISSN: 2089-3337. DOI: 10.11591/closer.v1i1.218.

[212] Manoj Kumar. "Serverless Architectures Review, Future Trend and the Solutions to Open Problems". In: *American Journal of Software Engineering* 6.1 (Mar. 12, 2019), pp. 1–10. ISSN: 2379-5271. DOI: 10.12691/ajse-6-1-1.

[213] Kathryn B. Laskey and Kenneth Laskey. "Service Oriented Architecture". In: *WIREs Computational Statistics* 1.1 (July 2009), pp. 101–105. ISSN: 1939-5108, 1939-0068. DOI: 10.1002/wics.8.

[214] Tom Laszewski. *Cloud Native Architectures: Design High-Availability and Cost-Effective Applications for the Cloud*. 1st ed. Birmingham: Packt Publishing Limited, 2018. 1 p. ISBN: 978-1-78728-054-0.

[215] C. Lattner and V. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* San Jose, CA, USA: IEEE, 2004, pp. 75–86. ISBN: 978-0-7695-2102-2. DOI: 10.1109/CGO.2004.1281665.

[216] Nikita Lazarev, Varun Gohil, James Tsai, Andy Anderson, Bhushan Chitlur, Zhiru Zhang, and Christina Delimitrou. "Sabre: Hardware-Accelerated Snapshot Compression for Serverless MicroVMs". In: *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 1–18. ISBN: 978-1-939133-40-3. URL: https://www.usenix.org/conference/osdi24/presentation/lazarev.

[217] Jeongmin Lee, Hyeongbin Kang, Hyeon-jin Yu, Ji-Hyun Na, Jungbin Kim, Jae-hyuck Shin, and Seo-Young Noh. "MDB-KCP: Persistence Framework of in-Memory Database with CRIU-based Container Checkpoint in Kubernetes". In: *Journal of Cloud Computing* 13.1 (July 25, 2024), p. 124. ISSN: 2192-113X. DOI: 10.1186/s13677-024-00687-9.

[218] Sang-Gun Lee, Seung Hoon Chae, and Kyung Min Cho. "Drivers and Inhibitors of SaaS Adoption in Korea". In: *International Journal of Information Management* 33.3 (June 2013), pp. 429–440. ISSN: 02684012. DOI: 10.1016/j.ijinfomgt.2013.01.006.

[219] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. "CloudCmp: Comparing Public Cloud Providers". In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. Melbourne Australia: ACM, Nov. 2010, pp. 1–14. ISBN: 978-1-4503-0483-2. DOI: 10.1145/1879141.1879143.

[220] Dongyang Li, Fei Wu, Yang Weng, Qing Yang, and Changsheng Xie. "HODS: Hardware Object Deserialization Inside SSD Storage". In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Boulder, CO: IEEE, Apr. 2018, pp. 157–164. ISBN: 978-1-5386-5522-1. DOI: 10.1109/FCCM.2018.00033.

[221] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. "Understanding and Addressing Quality Attributes of Microservices Architecture: A Systematic Literature Review". In: *Information and Software Technology* 131 (Mar. 2021), p. 106449. ISSN: 09505849. DOI: 10.1016/j.infsof.2020.106449.

[222] Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Chengzhong Xu. "Serverless Computing: State-of-the-Art, Challenges and Opportunities". In: *IEEE Transactions on Services Computing* 16.2 (Mar. 1, 2023), pp. 1522–1539. ISSN: 1939-1374, 2372-0204. DOI: 10.1109/TSC.2022.3166553.

[223] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. "The Serverless Computing Survey: A Technical Primer for Design Architecture". In: *ACM Computing Surveys* 54 (10s Jan. 31, 2022), pp. 1–34. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3508360.

[224] Changyuan Lin and Hamzeh Khazaei. "Modeling and Optimization of Performance and Cost of Serverless Applications". In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (Mar. 1, 2021), pp. 615–632. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2020.3028841.

[225] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. 2. ed., 3. printing. The Java Series ... from the Source. Boston Munich: Addison-Wesley, 2003. 473 pp. ISBN: 978-0-201-43294-7.

[226] Fangming Liu and Yipei Niu. "Demystifying the Cost of Serverless Computing: Towards a Win-Win Deal". In: *IEEE Transactions on Parallel and Distributed Systems* 35.1 (Jan. 2024), pp. 59–72. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2023.3330849.

[227] Feng Liu, Weiping Guo, Zhi Qiang Zhao, and Wu Chou. "SaaS Integration for Software Cloud". In: *2010 IEEE 3rd International Conference on Cloud Computing*. Miami, FL, USA: IEEE, July 2010, pp. 402–409. ISBN: 978-1-4244-8207-8. DOI: 10.1109/CLOUD.2010.67.

[228] Guozhi Liu, Bi Huang, Zhihong Liang, Minmin Qin, Hua Zhou, and Zhang Li. "Microservices: Architecture, Container, and Challenges". In: *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. Macau, China: IEEE, Dec. 2020, pp. 629–635. ISBN: 978-1-7281-8915-4. DOI: 10.1109/QRS-C51114.2020.00107.

[229] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. "Serverless Computing: An Investigation of Factors Influencing Microservice Performance". In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. Orlando, FL: IEEE, Apr. 2018, pp. 159–169. ISBN: 978-1-5386-5008-0. DOI: 10.1109/IC2E.2018.00039.

[230] Josh Lockhart. *Modern PHP: New Features and Good Practices*. First edition. Beijing Boston; Farnham Sebastopol Tokyo: O'Reilly, 2015. 244 pp. ISBN: 978-1-4919-0501-2.

[231] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. "Serialization/Deserialization-free State Transfer in Serverless Workflows". In: *Proceedings of the Nineteenth European Conference on Computer Systems*. Athens Greece: ACM, Apr. 22, 2024, pp. 132–147. ISBN: 979-8-4007-0437-6. DOI: 10.1145/3627703.3629568.

[232] Lukas Makor, Sebastian Kloibhofer, Peter Hofer, David Leopoldseder, and Hanspeter Mössenböck. "Automated Profile-Guided Replacement of Data Structures to Reduce Memory Allocation". In: *The Art, Science, and Engineering of Programming* 10.1 (Feb. 15, 2025), p. 3. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2025/10/3.

[233] Ilias Mavridis and Helen Karatza. "Performance Evaluation of Cloud-Based Log File Analysis with Apache Hadoop and Apache Spark". In: *Journal of Systems and Software* 125 (Mar. 2017), pp. 133–151. ISSN: 01641212. DOI: 10.1016/j.jss.2016.11.037.

[234] Garrett McGrath and Paul R. Brenner. "Serverless Computing: Design, Implementation, and Performance". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. Atlanta, GA, USA: IEEE, June 2017, pp. 405–410. ISBN: 978-1-5386-3292-5. DOI: 10.1109/ICDCSW.2017.36.

[235] Rohit K. Mehta and John Chandy. "Leveraging Checkpoint/Restore to Optimize Utilization of Cloud Compute Resources". In: *2015 IEEE 40th Local Computer Networks Conference Workshops (LCN Workshops)*. Clearwater Beach, FL: IEEE, Oct. 2015, pp. 714–721. ISBN: 978-1-4673-6773-8. DOI: 10.1109/LCNW.2015.7365919.

[236] P M Mell and T Grance. *The NIST Definition of Cloud Computing*. NIST SP 800-145. Gaithersburg, MD: National Institute of Standards and Technology, 2011, NIST SP 800–145. DOI: 10.6028/NIST.SP.800-145.

[237] *Memory Protection Keys — The Linux Kernel Documentation*. URL: https://docs.kernel.org/core-api/protection-keys.html (visited on Dec. 3, 2025).

[238] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux j* 239.2 (2014), p. 2.

[239] Bakhta Meroufel and Ghalem Belalem. "Optimization of Checkpointing/Recovery Strategy in Cloud Computing with Adaptive Storage Management". In: *Concurrency and Computation: Practice and Experience* 30.24 (Dec. 25, 2018), e4906. ISSN: 1532-0626, 1532-0634. DOI: 10.1002/cpe.4906.

[240] *MessagePack: It's like JSON. but Fast and Small.* URL: https://msgpack.org/ (visited on Nov. 21, 2025).

[241] Meta. *Facebook*. Facebook. URL: https://www.facebook.com/ (visited on Dec. 27, 2025).

[242] *Micronaut*. URL: https://micronaut.io/ (visited on Nov. 30, 2025).

[243] Microsoft. *Microsoft Azure*. URL: https://azure.microsoft.com/en-us (visited on Nov. 26, 2025).

[244] Microsoft. *Microsoft OneDrive*. URL: https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage (visited on Nov. 28, 2025).

[245] Microsoft. *Microsoft Teams: Video Conferencing, Meetings, Calling*. URL: https://www.microsoft.com/en-us/microsoft-teams/group-chat-software (visited on Nov. 28, 2025).

[246] Microsoft. *Office 365*. URL: https://www.office.com/ (visited on Dec. 27, 2025).

[247] Microsoft. *Outlook*. URL: https://www.outlook.com/ (visited on Dec. 27, 2025).

[248] Lazar Milikic, Milan Cugurovic, and Vojin Jovanovic. "GraalNN: Context-Sensitive Static Profiling with Graph Neural Networks". In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. Las Vegas NV USA: ACM, Mar. 2025, pp. 123–136. ISBN: 979-8-4007-1275-3. DOI: 10.1145/3696443.3708958.

[249] Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. "Instant Pickles: Generating Object-Oriented Pickler Combinators for Fast and Extensible Serialization". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. Indianapolis Indiana USA: ACM, Oct. 29, 2013, pp. 183–202. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509547.

[250] Chnar Mustafa Mohammed and Subhi R.M Zeebaree. "Sufficient Comparison among Cloud Computing Services: IaaS, PaaS, and SaaS: A Review". In: *International Journal of Science and Business* 5.2 (2021), pp. 17–30. DOI: None.

[251] Brij Mohan. *X-AD: Explainable Anomaly Detection in Serverless Systems*. Oct. 24, 2025. DOI: 10.36227/techrxiv.176127299.98238158/v1. Pre-published.

[252] *MongoDB: The World's Leading Modern Database*. MongoDB. URL: https://www.mongodb.com/ (visited on Nov. 21, 2025).

[253] Tokito Murata and Kenichi Kourai. "Parallel and Consistent Live Checkpointing and Restoration of Split-Memory VMs". In: *Future Generation Computer Systems* 159 (Oct. 2024), pp. 432–443. ISSN: 0167739X. DOI: 10.1016/j.future.2024.05.024.

[254] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Michael Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture*. First edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly Media, 2016. 1 p. ISBN: 978-1-4919-5625-0.

[255] *National Institute of Standards and Technology*. URL: https://www.nist.gov/ (visited on Nov. 23, 2025).

[256] Netflix. *Conductor*. Netflix, Inc., Nov. 26, 2025. URL: https://github.com/Netflix/conductor (visited on Nov. 26, 2025).

[257] *Ngs-Doo/Dsl-Json*. New Generation Software Ltd, Nov. 17, 2025. URL: https://github.com/ngs-doo/dsl-json (visited on Nov. 20, 2025).

[258] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. "Skyway: Connecting Managed Heaps in Distributed Big Data Systems". In: *ACM SIGPLAN Notices* 53.2 (Nov. 30, 2018), pp. 56–69. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/3296957.3173200.

[259] Muhammad Niswar, Reza Arisandy Safruddin, Anugrayani Bustamin, and Iqra Aswad. "Performance Evaluation of Microservices Communication with REST, GraphQL, and gRPC". In: *International Journal of Electronics and Telecommunication* 70.2 (2024), pp. 429–436.

[260] *Notion*. URL: https://www.notion.com/ (visited on Nov. 28, 2025).

[261] James O'Reilly. *Network Storage: Tools and Technologies for Storing Your Company's Data*. Cambridge, MA: Morgan Kaufmann, 2017. 1 p. ISBN: 978-0-12-803865-9.

[262] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 57–70. ISBN: 978-1-931971-44-7. URL: https://www.usenix.org/conference/atc18/presentation/oakes.

[263] Tiago Oliveira, Ricardo Martins, Saonee Sarker, Manoj Thomas, and Aleš Popovič. "Understanding SaaS Adoption: The Moderating Impact of the Environment Context". In: *International Journal of Information Management* 49 (Dec. 2019), pp. 1–12. ISSN: 02684012. DOI: 10.1016/j.ijinfomgt.2019.02.009.

[264] *Open Data Description Language*. URL: https://opendd1.org/ (visited on Nov. 20, 2025).

[265] *Open Liberty Docs*. URL: https://openliberty.io/docs/latest/instanton.html (visited on Nov. 15, 2025).

[266] OpenFaaS Ltd. *OpenFaaS*. OpenFaaS - Serverless Functions Made Simple. URL: https://www.openfaas.com/ (visited on Nov. 26, 2025).

[267] *OpenJDK*. URL: https://openjdk.org/ (visited on Nov. 19, 2025).

[268] OpenJDK. *Java Microbenchmark Harness*. OpenJDK, Dec. 11, 2025. URL: https://github.com/openjdk/jmh (visited on Dec. 11, 2025).

[269] *OpenJDK: CRaC*. URL: https://openjdk.org/projects/crac/ (visited on Nov. 15, 2025).

[270] Oracle. *GraalOS*. URL: https://www.graal.cloud/graalos/ (visited on Nov. 30, 2025).

[271] Oracle. *GraalVM*. URL: https://www.graalvm.org/downloads/ (visited on Nov. 19, 2025).

[272] Oracle. *Java Object Serialization Specification*. online. URL: https://docs.oracle.com/en/java/javase/17/docs/specs/serialization/serial-arch.html.

[273] Oracle. *Oracle Cloud Infrastructure*. URL: https://www.oracle.com/cloud/ (visited on Nov. 26, 2025).

[274] OVH. *OVHcloud Global Cloud Service Provider*. URL: https://us.ovhcloud.com/ (visited on Nov. 26, 2025).

[275] Claus Pahl. "Containerization and the PaaS Cloud". In: *IEEE Cloud Computing* 2.3 (May 2015), pp. 24–31. ISSN: 2325-6095. DOI: 10.1109/MCC.2015.51.

[276] Michael Paleczny, Christopher Vick, and Cliff Click. "The Java HotSpot™ Server Compiler". In: *Java (TM) Virtual Machine Research and Technology Symposium (JVM 01)*. Monterey, CA: USENIX Association, Apr. 2001. URL: https://www.usenix.org/conference/jvm-01/java-hotspot%7Btexttrademark%7D-server-compiler.

[277] Pu Pang, Gang Deng, Kaihao Bai, Quan Chen, Shixuan Sun, Bo Liu, Yu Xu, Hongbo Yao, Zhengheng Wang, Xiyu Wang, Zheng Liu, Zhuo Song, Yong Yang, Tao Ma, and Minyi Guo. *Async-Fork: Mitigating Query Latency Spikes Incurred by the Fork-based Snapshot Mechanism from the OS Level*. Jan. 18, 2023. DOI: 10.48550/arXiv.2301.05861. arXiv: 2301.05861 [cs]. Pre-published.

[278] Fawaz Paraiso, Nicolas Haderer, Philippe Merle, Romain Rouvoy, and Lionel Seinturier. "A Federated Multi-cloud PaaS Infrastructure". In: *2012 IEEE Fifth International Conference on Cloud Computing*. Honolulu, HI, USA: IEEE, June 2012, pp. 392–399. ISBN: 978-1-4673-2892-0. DOI: 10.1109/CLOUD.2012.79.

[279] Parallels International GmbH. *Parallels RAS: Application and Desktop Delivery*. URL: https://www.parallels.com/ (visited on Nov. 11, 2025).

[280] *Parquet*. Parquet. URL: https://parquet.apache.org/ (visited on Nov. 21, 2025).

[281] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs". In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. Toronto Ontario Canada: ACM, Apr. 24, 2010, pp. 2–11. ISBN: 978-1-60558-635-9. DOI: 10.1145/1772954.1772958.

[282] Wayne Pauley. "Cloud Provider Transparency: An Empirical Evaluation". In: *IEEE Security & Privacy Magazine* 8.6 (Nov. 2010), pp. 32–39. ISSN: 1540-7993. DOI: 10.1109/MSP.2010.140.

[283] *PEP 3154 – Pickle Protocol Version 4*. Python Enhancement Proposals (PEPs). URL: https://peps.python.org/pep-3154/ (visited on Nov. 22, 2025).

[284] Cristiano Pereira, Harish Patil, and Brad Calder. "Reproducible Simulation of Multi-Threaded Workloads for Architecture Design Exploration". In: *2008 IEEE International Symposium on Workload Characterization*. Seattle, WA, USA: IEEE, Oct. 2008, pp. 173–182. ISBN: 978-1-4244-2777-2. DOI: 10.1109/IISWC.2008.4636102.

[285] T. Pflanzner and A. Kertesz. "A Survey of IoT Cloud Providers". In: *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. Opatija, Croatia: IEEE, May 2016, pp. 730–735. ISBN: 978-953-233-086-1. DOI: 10.1109/MIPRO.2016.7522237.

[286] Maksym Planeta, Jan Bierbaum, Leo Sahaya Daphne Antony, Torsten Hoefler, and Hermann Härtig. *MigrOS: Transparent Operating Systems Live Migration Support for Containerised RDMA-applications*. Version 2. 2020. DOI: 10.48550/ARXIV.2009.06988. Pre-published.

[287] Matthew Portnoy. *Virtualization Essentials*. Third edition. Erscheinungsort nicht ermittelbar: Sybex, 2023. 1 p. ISBN: 978-1-394-18157-5.

[288] Nigel Poulton. *Docker Deep Dive: Zero to Docker in a Single Book*. 2020 edition. Germany: Nigel Poulton, 2020. ISBN: 978-1-5218-2280-7.

[289] Nigel Poulton and Pushkar Joglekar. *The Kubernetes Book*. Version 4, March 2019. United Kingdom: Nigel Poulton, 2019.

[290] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. "Optimus Prime: Accelerating Data Transformation in Servers". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Lausanne Switzerland: ACM, Mar. 9, 2020, pp. 1203–1216. ISBN: 978-1-4503-7102-5. DOI: 10.1145/3373376.3378501.

[291] Chandra Prakash, Debadatta Mishra, Purushottam Kulkarni, and Umesh Bellur. "Portkey: Hypervisor-Assisted Container Migration in Nested Cloud Environments". In: *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Virtual Switzerland: ACM, Feb. 25, 2022, pp. 3–17. ISBN: 978-1-4503-9251-8. DOI: 10.1145/3516807.3516817.

[292] Mark J. Price. *C# 12 and .NET 8 - Modern Cross-Platform Development Fundamentals: Start Building Websites and Services with ASP.NET Core 8, Blazor, and EF Core 8*. 1st ed. Birmingham: Packt Publishing Limited, 2023. 1 p. ISBN: 978-1-83763-587-0.

[293] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. "An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers". In: *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. Cgo 2019. Washington, DC, USA: IEEE Press, 2019, pp. 164–179. ISBN: 978-1-7281-1436-1.

[294] Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. *Stanza: A Python Natural Language Processing Toolkit for Many Human Languages*. Version 2. 2020. DOI: 10.48550/ARXIV.2003.07082. Pre-published.

[295] *Rackspace Technology | Multicloud Solutions Provider*. URL: https://www.rackspace.com/ (visited on Nov. 26, 2025).

[296] Ray Rafaels. *Cloud Computing: From Beginning to End: Cloud Technology, Design, and Migration Methodologies Explained*. 2. Aufl. Rafaels, 2018. 12 pp. ISBN: 978-1-9867-2628-3.

[297] Md Shahidur Rahaman, Sadia Nasrin Tisha, Eunjee Song, and Tomas Cerny. "Access Control Design Practice and Solutions in Cloud-Native Architecture: A Systematic Mapping Study". In: *Sensors* 23.7 (Mar. 24, 2023), p. 3413. ISSN: 1424-8220. DOI: 10.3390/s23073413.

[298] Arokia Paul Rajan. "A Review on Serverless Architectures - Function as a Service (FaaS) in Cloud Computing". In: *TELKOMNIKA (Telecommunication Computing Electronics and Control)* 18.1 (Feb. 1, 2020), p. 530. ISSN: 2302-9293, 1693-6930. DOI: 10.12928/telkomnika.v18i1.12169.

[299] R. Arokia Paul Rajan. "Serverless Architecture - A Revolution in Cloud Computing". In: *2018 Tenth International Conference on Advanced Computing (ICoAC)*. Chennai, India: IEEE, Dec. 2018, pp. 88–93. ISBN: 978-1-7281-0353-2. DOI: 10.1109/ICoAC44903.2018.8939081.

[300] Dimpi Rani and Rajiv Ranjan. "A Comparative Study of SaaS, PaaS and IaaS in Cloud Computing". In: *International Journals of Advanced Research in Computer Science and Software Engineering* 4.6 (June 2014), pp. 458–461.

[301] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. *A Remote Direct Memory Access Protocol Specification*. RFC5040. RFC Editor, Oct. 2007, RFC5040. DOI: 10.17487/rfc5040.

[302] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. "PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education". In: *Proceedings of the 2004 Workshop on Computer Architecture Education Held in Conjunction with the 31st International Symposium on Computer Architecture - WCAE '04*. Munich, Germany: ACM Press, 2004, 22–es. ISBN: 978-1-4503-4733-4. DOI: 10.1145/1275571.1275600.

[303] Fabien Renaud. *Fabienrenaud/Java-Json-Benchmark*. Oct. 30, 2025. URL: https://github.com/fabienrenaud/java-json-benchmark (visited on Nov. 20, 2025).

[304] Leonard Richardson and Michael Amundsen. *RESTful Web APIs*. First edition, second release. Beijing Cambridge Farnham Köln Sebastopol Tokyo: O'Reilly, 2015. 373 pp. ISBN: 978-1-4493-5806-8.

[305] Felix Richter. *Infographic: AWS Stays Ahead as Cloud Market Accelerates*. Statista Daily Data. Nov. 4, 2025. URL: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers (visited on Nov. 26, 2025).

[306] Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. "Sulong - Execution of LLVM-based Languages on the JVM: Position Paper". In: *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. Rome Italy: ACM, July 17, 2016, pp. 1–4. ISBN: 978-1-4503-4837-9. DOI: 10.1145/3012408.3012416.

[307] Ivan Ristović. *Java Native Serialization/Deserialization Benchmarks*. Zenodo, July 28, 2025. DOI: 10.5281/ZENODO.16535566.

[308] Ivan Ristović. "LINVAST Infrastructure for Language-Invariant Abstract Syntax Trees". In: *Journal of Information Technology and Multimedia Systems — Info M* 25.81/82 (2025), pp. 47–54.

[309] Ivan Ristović. "Pre-Initialization of the Java Virtual Machine Context". In: *Proceedings of the 13th Symposium "Mathematics and Applications"*. Dec. 2023.

[310] Ivan Ristović. *Sd-Benchmarks: A Set of Micro and Macro S/D Benchmarks*. URL: https://github.com/ivan-ristovic/sd-benchmarks (visited on Dec. 11, 2025).

[311] Ivan Ristović, Milan Čugurović, Strahinja Stanojević, Marko Spasić, Vesna Marinković, and Milena Vujosević Janicić. "Analyzing Control Flow Graph Traversals". In: *Proceedings of the 13th Symposium "Mathematics and Applications"*. Dec. 2023.

[312] Ivan Ristović, Milan Čugurović, Strahinja Stanojević, Marko Spasić, Vesna Marinković, and Milena Vujosević Janicić. "Efficient Control-Flow Graph Traversal". In: *Proceedings of the 15th YUINFO Conference*. Mar. 2024.

[313] Ivan Ristović, Milan Čugurović, Strahinja Stanojević, Marko Spasić, Vesna Marinković, and Milena Vujosević Janicić. "Machine Learning-Driven Prediction of Optimal Control Flow Graph Traversal Strategy". In: *Serbian Journal of Electrical Engineering* 22.3 (2025). DOI: 10.2298/SJEE250828003R.

[314] Ivan Ristović, Milena Vujosević Janicić, Peter Hofer, and Vojin Jovanović. "Object Snapshotting and Sharing across Application Instances". Pat. ORC25139810-US-NPR (United States). Mar. 2025.

[315] Ivan Ristović, Vojin Jovanović, Peter Hofer, and Milena Vujošević Janičić. "GraalDoss: Direct Object Snapshotting and Sharing for Cloud-Native Applications". In: *Future Generation Computer Systems* (2026), p. 108375. ISSN: 0167-739X. DOI: 10.1016/j.future.2026.108375.

[316] Joanna Rutkowska and Rafal Wojtczuk. "Qubes OS Architecture". In: *Invisible Things Lab Tech Rep* 54 (2010), p. 65.

[317] Salesforce. *Heroku*. Heroku. URL: https://www.heroku.com/ (visited on Nov. 26, 2025).

[318] Salesforce. *Salesforce Automation Software by Sales Cloud*. Salesforce. URL: https://www.salesforce.com/sales/cloud/ (visited on Nov. 26, 2025).

[319] Pierangela Samarati and Sabrina De Capitani Di Vimercati. "Cloud Security: Issues and Concerns". In: *Encyclopedia of Cloud Computing*. Ed. by San Murugesan and Irena Bojanova. 1st ed. Wiley, June 9, 2016, pp. 205–219. ISBN: 978-1-118-82197-8. DOI: 10.1002/9781118821930.ch17.

[320] Manish Saraswat and R.C. Tripathi. "Cloud Computing: Analysis of Top 5 CSPs in SaaS, PaaS and IaaS Platforms". In: *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*. Moradabad, India: IEEE, Dec. 4, 2020, pp. 300–305. ISBN: 978-1-7281-8908-6. DOI: 10.1109/SMART50582.2020.9337157.

[321] Manish Saraswat and R.C. Tripathi. "Cloud Computing: Comparison and Analysis of Cloud Service Providers-AWs, Microsoft and Google". In: *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*. Moradabad, India: IEEE, Dec. 4, 2020, pp. 281–285. ISBN: 978-1-7281-8908-6. DOI: 10.1109/SMART50582.2020.9337100.

[322] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. "Memory Deduplication for Serverless Computing with Medes". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. Rennes France: ACM, Mar. 28, 2022, pp. 714–729. ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3524272.

[323] Gabriel N. Schenker, Hideto Saito, Hui-Chuan Chloe Lee, and Ke-Jou Carol Hsu. *Getting Started with Containerization: Reduce the Operational Burden on Your System by Automating and Managing Your Containers*. 1st ed. Birmingham: Packt Publishing, 2019. 1 p. ISBN: 978-1-83864-570-0.

[324] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. "What Serverless Computing Is and Should Become: The next Phase of Cloud Computing". In: *Communications of the ACM* 64.5 (May 2021), pp. 76–84. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3406011.

[325] Werner Schuster. *The Essence of Google Dart: Building Applications, Snapshots, Isolates*. online. 2011. URL: https://www.infoq.com/articles/google-dart/.

[326] Berat Can Şenel, Maxime Mouchet, Justin Cappos, Timur Friedman, Olivier Fourmaux, and Rick McGeer. "Multitenant Containers as a Service (CaaS) for Clouds and Edge Clouds". In: *IEEE Access* 11 (2023), pp. 144574–144601. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3344486.

[327] Mohit Sewak and Sachchidanand Singh. "Winning in the Era of Serverless Computing and Function as a Service". In: *2018 3rd International Conference for Convergence in Technology (I2CT)*. Pune, India: IEEE, Apr. 2018, pp. 1–5. ISBN: 978-1-5386-4273-3. DOI: 10.1109/I2CT.2018.8529465.

[328] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. "Serverless Computing: A Survey of Opportunities, Challenges, and Applications". In: *ACM Computing Surveys* 54 (11s Jan. 31, 2022), pp. 1–32. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3510611.

[329] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Request for Comments RFC 4180. Internet Engineering Task Force, Oct. 2005. 8 pp. DOI: 10.17487/RFC4180.

[330] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. "Serverless Linear Algebra". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. Virtual Event USA: ACM, Oct. 12, 2020, pp. 281–295. ISBN: 978-1-4503-8137-6. DOI: 10.1145/3419111.3421287.

[331] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. "Fireworks: A Fast, Efficient, and Safe Serverless Framework Using VM-level Post-JIT Snapshot". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. Rennes France: ACM, Mar. 28, 2022, pp. 663–677. ISBN: 978-1-4503-9162-7. DOI: 10.1145/3492321.3519581.

[332] João M. Silva, José Simão, and Luís Veiga. "Ditto – Deterministic Execution Replayability-as-a-Service for Java VM on Multiprocessors". In: *Middleware 2013*. Ed. by David Eyers and Karsten Schwan. Red. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, and Gerhard Weikum. Vol. 8275. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 405–424. ISBN: 978-3-642-45064-8. DOI: 10.1007/978-3-642-45065-5_21.

[333] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. "Prebaking Functions to Warm the Serverless Cold Start". In: *Proceedings of the 21st International Middleware Conference*. Delft Netherlands: ACM, Dec. 7, 2020, pp. 1–13. ISBN: 978-1-4503-8153-6. DOI: 10.1145/3423211.3425682.

[334] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. "Prebaking Functions to Warm the Serverless Cold Start". In: *Proceedings of the 21st International Middleware Conference*. Delft Netherlands: ACM, Dec. 7, 2020, pp. 1–13. ISBN: 978-1-4503-8153-6. DOI: 10.1145/3423211.3425682.

[335] José Simão, Tiago Garrochinho, and Luís Veiga. "A Checkpointing-enabled and Resource-aware Java Virtual Machine for Efficient and Robust e-Science Applications in Grid Environments". In: *Concurrency and Computation: Practice and Experience* 24.13 (Sept. 10, 2012), pp. 1421–1442. ISSN: 1532-0626, 1532-0634. DOI: 10.1002/cpe.1879.

[336] Chris Simmonds. *Mastering Embedded Linux Programming, Second Edition*. 2nd ed. Erscheinungsort nicht ermittelbar: PACKT Publishing, 2017. 468 pp. ISBN: 978-1-78728-328-2.

[337] Akanksha Singh, Smita Sharma, Shipra Ravi Kumar, and Suman Avdesh Yadav. "Overview of PaaS and SaaS and Its Application in Cloud Computing". In: *2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH)*. Greater Noida, India: IEEE, Feb. 2016, pp. 172–176. ISBN: 978-1-5090-2084-3. DOI: 10.1109/ICICCS.2016.7542322.

[338] Ashish Singh and Kakali Chatterjee. "Cloud Security Issues and Challenges: A Survey". In: *Journal of Network and Computer Applications* 79 (Feb. 2017), pp. 88–115. ISSN: 10848045. DOI: 10.1016/j.jnca.2016.11.027.

[339] Slack. *Slack*. URL: https://slack.com (visited on Nov. 28, 2025).

[340] Eishay Smith. *Eishay/Jvm-Serializers*. Nov. 6, 2025. URL: https://github.com/eishay/jvm-serializers (visited on Nov. 20, 2025).

[341] Richard T. Snodgrass and Karen P. Shannon. *The Interface Description Language: Definition and Use*. Principles of Computer Science Series. Rockville, MD: Computer Science Pr. [u.a.], 1989. 615 pp. ISBN: 978-0-7167-8198-1.

[342] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. "The Pains and Gains of Microservices: A Systematic Grey Literature Review". In: *Journal of Systems and Software* 146 (Dec. 2018), pp. 215–232. ISSN: 01641212. DOI: 10.1016/j.jss.2018.09.082.

[343] Jiri Soukup. *Serialization and Persistent Objects: Turning Data Structures into Efficient Databases*. 1st ed. Berlin, Heidelberg: Springer Berlin / Heidelberg, 2014. 1 p. ISBN: 978-3-642-39323-5.

[344] Marko Spasić, Ivan Ristović, Strahinja Stanojević, Milan Čugurović, Milica Karličić, and Milena Vujosević Janicić. "Evaluating GraalVM Compiler Performance Using a Distributed Computing Cluster". In: *Proceedings of the 15th Serbian Mathematical Congress*. June 2024.

[345] Codruţ Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. "Comparing Points-to Static Analysis with Runtime Recorded Profiling Data". In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. Cracow Poland: ACM, Sept. 23, 2014, pp. 157–168. ISBN: 978-1-4503-2926-2. DOI: 10.1145/2647508.2647524.

[346] Aleksandar Stefanović, Ivan Ristović, and Milena Vujošević Janičić. "Constant Folding of Reflective Calls via Static Analysis of Java Bytecode". In: *Proceedings of the 14th Symposium "Mathematics and Applications"*. Dec. 2024.

[347] Aleksandar Stefanović, Ivan Ristović, and Milena Vujošević Janičić. "Extensible Java Bytecode Data-Flow Analysis JVMCI Framework". In: *Proceedings of the 14th Symposium "Mathematics and Applications"*. Dec. 2025.

[348] Radostin Stoyanov, Adrian Reber, Daiki Ueno, Michał Clapiński, Andrei Vagin, and Rodrigo Bruno. "Towards Efficient End-to-End Encryption for Container Checkpointing Systems". In: *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*. Kyoto Japan: ACM, Sept. 4, 2024, pp. 60–66. ISBN: 979-8-4007-1105-3. DOI: 10.1145/3678015.3680477.

[349] Radostin Stoyanov, Viktória Spišaková, Jesus Ramos, Steven Gurfinkel, Andrei Vagin, Adrian Reber, Wesley Armour, and Rodrigo Bruno. *CRIUgpu: Transparent Checkpointing of GPU-Accelerated Workloads*. Version 1. 2025. DOI: 10.48550/ARXIV.2502.16631. Pre-published.

[350] Masato Suetake, Hazuki Kizu, and Kenichi Kourai. "Split Migration of Large Memory Virtual Machines". In: *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*. Hong Kong Hong Kong: ACM, Aug. 4, 2016, pp. 1–8. ISBN: 978-1-4503-4265-0. DOI: 10.1145/2967360.2967368.

[351] Audie Sumaray and S. Kami Makki. "A Comparison of Data Serialization Formats for Optimal Efficiency on a Mobile Platform". In: *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. Kuala Lumpur Malaysia: ACM, Feb. 20, 2012, pp. 1–6. ISBN: 978-1-4503-1172-4. DOI: 10.1145/2184751.2184810.

[352] Alexander Supalov. *Inside the Message Passing Interface: Creating Fast Communication Libraries*. Boston Berlin: DeG Press, 2018. 1 p. ISBN: 978-1-5015-1554-5.

[353] Lakshmisri Surya. "Software as a Service in Cloud Computing". In: *SSRN Electronic Journal* 7 (Dec. 2019), pp. 182–186.

[354] Andy Syrewicze. *Pro Microsoft Hyper-V 2019: Practical Guidance and Hands-On Labs*. Berkeley, CA: Apress L. P, 2018. 1 p. ISBN: 978-1-4842-4115-8.

[355] Márk Szalay, Péter Mátray, and László Toka. "Real-Time FaaS: Towards a Latency Bounded Serverless Cloud". In: *IEEE Transactions on Cloud Computing* 11.2 (Apr. 1, 2023), pp. 1636–1650. ISSN: 2168-7161, 2372-0018. DOI: 10.1109/TCC.2022.3151469.

[356] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. "Naos: Serialization-free RDMA Networking in Java". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 1–14. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/taranov.

[357] Tencent. *Tencent Cloud*. URL: https://www.tencentcloud.com/ (visited on Nov. 26, 2025).

[358] The Apache Software Foundation. *Apache Arrow*. Apache Arrow. URL: https://arrow.apache.org/ (visited on Nov. 21, 2025).

[359] The Apache Software Foundation. *Apache Avro*. Apache Avro. URL: https://avro.apache.org/ (visited on Nov. 21, 2025).

[360] The Apache Software Foundation. *Apache Fory™*. URL: https://fory.apache.org/ (visited on Nov. 21, 2025).

[361] The Apache Software Foundation. *Apache Thrift*. URL: https://thrift.apache.org/ (visited on Nov. 21, 2025).

[362] *The JavaScript Object Notation (JSON) Data Interchange Format*. URL: https://www.rfc-editor.org/info/std90 (visited on Nov. 20, 2025).

[363] Aleksandar Tošić. "Run-Time Application Migration Using Checkpoint/Restore in Userspace". Version 1. In: (2023). DOI: 10.48550/ARXIV.2307.12113.

[364] *Travis CI/CD*. URL: https://www.travis-ci.com/ (visited on Nov. 28, 2025).

[365] *Trello*. URL: https://trello.com/ (visited on Nov. 28, 2025).

[366] WeiTek Tsai, XiaoYing Bai, and Yu Huang. "Software-as-a-Service (SaaS): Perspectives and Challenges". In: *Science China Information Sciences* 57.5 (May 2014), pp. 1–15. ISSN: 1674-733X, 1869-1919. DOI: 10.1007/s11432-013-5050-z.

[367] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. "Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing". In: *ACM SIGARCH Computer Architecture News* 44.3 (Oct. 12, 2016), pp. 53–65. ISSN: 0163-5964. DOI: 10.1145/3007787.3001143.

[368] Nnaemeka Kingsley Ugwumba and Peter Sunday Jaja. *Ahead-of-Time vs. Just-in-Time Compilation Trade-offs: Empirical Performance Studies*. Oct. 22, 2025. DOI: 10.21203/rs.3.rs-7915532/v1. Pre-published.

[369] David Ungar. "Annotating Objects for Transport to Other Worlds". In: *ACM SIGPLAN Notices* 30.10 (Oct. 17, 1995), pp. 73–87. ISSN: 0362-1340, 1558-1160. DOI: 10.1145/217839.217845.

[370] *Universal Binary JSON Specification – The Universally Compatible Format Specification for Binary JSON*. URL: https://ubjson.org/ (visited on Nov. 21, 2025).

[371] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. "Benchmarking, Analysis, and Optimization of Serverless Function Snapshots". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Virtual USA: ACM, Apr. 19, 2021, pp. 559–572. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446714.

[372] *V8 JavaScript Engine*. URL: https://v8.dev/ (visited on Nov. 19, 2025).

[373] Mohammad Hadi Valipour, Bavar Amirzafari, Khashayar Niki Maleki, and Negin Daneshpour. "A Brief Survey of Software Architecture Concepts and Service Oriented Architecture". In: *2009 2nd IEEE International Conference on Computer Science and Information Technology*. Beijing: IEEE, Aug. 2009, pp. 34–38. ISBN: 978-1-4244-4519-6. DOI: 10.1109/ICCSIT.2009.5235004.

[374] Guido VanRossum and Fred L Drake. *The Python Language Reference*. Vol. 561. Python Software Foundation Amsterdam, The Netherlands, 2010.

[375] Luis M. Vaquero. "EduCloud: PaaS versus IaaS Cloud Usage for an Advanced Computer Science Course". In: *IEEE Transactions on Education* 54.4 (Nov. 2011), pp. 590–598. ISSN: 0018-9359, 1557-9638. DOI: 10.1109/TE.2010.2100097.

[376] Bruno Venditti. *The World's Largest Cloud Providers, Ranked by Market Share*. Visual Capitalist. Sept. 10, 2025. URL: https://www.visualcapitalist.com/the-worlds-largest-cloud-providers-ranked-by-market-share/ (visited on Nov. 26, 2025).

[377] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S. Milojicic, and Ada Gavrilovska. "Fast In-Memory CRIU for Docker Containers". In: *Proceedings of the International Symposium on Memory Systems*. Washington District of Columbia USA: ACM, Sept. 30, 2019, pp. 53–65. ISBN: 978-1-4503-7206-0. DOI: 10.1145/3357526.3357542.

[378] Juan Cruz Viotti and Mital Kinderkhedia. *A Survey of JSON-compatible Binary Serialization Specifications*. Version 2. 2022. DOI: 10.48550/ARXIV.2201.02089. Pre-published.

[379] William Von Hagen. *Professional Xen Virtualization*. Wrox Professional Guides. Indianapolis, Ind: Wiley Pub, 2008. 1 p. ISBN: 978-0-470-13811-3.

[380] Maja Vukasovic and Aleksandar Prokopec. "Exploiting Partially Context-sensitive Profiles to Improve Performance of Hot Code". In: *ACM Transactions on Programming Languages and Systems* 45.4 (Dec. 31, 2023), pp. 1–64. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/3612937.

[381] Stefan Walraven, Eddy Truyen, and Wouter Joosen. "Comparing PaaS Offerings in Light of SaaS Development: A Comparison of PaaS Platforms Based on a Practical Case Study". In: *Computing* 96.8 (Aug. 2014), pp. 669–724. ISSN: 0010-485X, 1436-5057. DOI: 10.1007/s00607-013-0346-9.

[382] Hao Wang, Di Niu, and Baochun Li. "Distributed Machine Learning with a Serverless Architecture". In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. Paris, France: IEEE, Apr. 2019, pp. 1288–1296. ISBN: 978-1-7281-0515-4. DOI: 10.1109/INFOCOM.2019.8737391.

[383] Muhammad Waseem, Peng Liang, and Mojtaba Shahin. "A Systematic Mapping Study on Microservices Architecture in DevOps". In: *Journal of Systems and Software* 170 (Dec. 2020), p. 110798. ISSN: 01641212. DOI: 10.1016/j.jss.2020.110798.

[384] Wang Wei, Li Na, Zhang Lei, Liu Fang, Chen Hao, Yang Xiuying, Huang Lei, Zhao Min, Wu Gang, Zhou Jie, Xu Jing, Sun Tao, Ma Li, Zhu Qiang, Hu Jun, Guo Wei, He Yong, Gao Yuan, Lin Dan, Zheng Yi, and Shi Li. *An Extensive Study on Text Serialization Formats and Methods*. Version 1. 2025. DOI: 10.48550/ARXIV.2505.13478. Pre-published.

[385] Sebastian Werner, Jorn Kuhlenkamp, Markus Klems, Johannes Muller, and Stefan Tai. "Serverless Big Data Processing Using Matrix Multiplication as Example". In: *2018 IEEE International Conference on Big Data (Big Data)*. Seattle, WA, USA: IEEE, Dec. 2018, pp. 358–365. ISBN: 978-1-5386-5035-6. DOI: 10.1109/BigData.2018.8622362.

[386] Max Wildgrube. *Structured Data Exchange Format (SDXF)*. Request for Comments RFC 3072. Internet Engineering Task Force, Mar. 2001. 26 pp. DOI: 10.17487/RFC3072.

[387] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. "Initialize Once, Start Fast: Application Initialization at Build Time". In: *Proceedings of the ACM on Programming Languages* 3 (OOPSLA Oct. 10, 2019), pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3360610.

[388] Christian Wimmer, Codrut Stancu, David Kozak, and Thomas Würthinger. "Scaling Type-Based Points-to Analysis with Saturation". In: *Proceedings of the ACM on Programming Languages* 8 (PLDI June 20, 2024), pp. 990–1013. ISSN: 2475-1421. DOI: 10.1145/3656417.

[389] Christian Wimmer and Thomas Würthinger. "Truffle: A Self-Optimizing Runtime System". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. Tucson Arizona USA: ACM, Oct. 19, 2012, pp. 13–14. ISBN: 978-1-4503-1563-0. DOI: 10.1145/2384716.2384723.

[390] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. 2. Second Edition 2024. Berlin, Heidelberg: Springer Berlin Heidelberg, 2024. 1 p. ISBN: 978-3-662-69306-3. DOI: 10.1007/978-3-662-69306-3.

[391] Mingyu Wu, Shuaiwei Wang, Haibo Chen, and Binyu Zang. "Zero-Change Object Transmission for Distributed Big Data Analytics". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 137–150. URL: https://www.usenix.org/conference/atc22/presentation/wu.

[392] Wei-Wen Wu. "Developing an Explorative Model for SaaS Adoption". In: *Expert Systems with Applications* 38.12 (Nov. 2011), pp. 15057–15064. ISSN: 09574174. DOI: 10.1016/j.eswa.2011.05.039.

[393] Wei-Wen Wu, Lawrence W. Lan, and Yu-Ting Lee. "Exploring Decisive Factors Affecting an Organization's SaaS Adoption: A Case Study". In: *International Journal of Information Management* 31.6 (Dec. 2011), pp. 556–563. ISSN: 02684012. DOI: 10.1016/j.ijinfomgt.2011.02.007.

[394] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. "Self-Optimizing AST Interpreters". In: *Proceedings of the 8th Symposium on Dynamic Languages*. Tucson Arizona USA: ACM, Oct. 22, 2012, pp. 73–82. ISBN: 978-1-4503-1564-7. DOI: 10.1145/2384577.2384587.

[395] X Corp. *X*. X (formerly Twitter). URL: https://x.com/ (visited on Dec. 27, 2025).

[396] *Xen Project*. URL: https://xenproject.org/ (visited on Nov. 12, 2025).

[397] Weijun Xiao, Yinan Liu, Qing Yang, Jin Ren, and Changsheng Xie. "Implementation and Performance Evaluation of Two Snapshot Methods on iSCSI Target Storages". In: *Proc. Of NASA/IEEE Conference on Mass Storage Systems and Technologies*. 2006, pp. 101–110.

[398] Tong Xing, Antonio Barbalace, Pierre Olivier, Mohamed L. Karaoui, Wei Wang, and Binoy Ravindran. "H-Container: Enabling Heterogeneous-ISA Container Migration in Edge Computing". In: *ACM Transactions on Computer Systems* 39.1–4 (Nov. 30, 2021), pp. 1–36. ISSN: 0734-2071, 1557-7333. DOI: 10.1145/3524452.

[399] *YAML™ Specification Index*. URL: https://yaml.org/spec/ (visited on Nov. 19, 2025).

[400] Albert Mingkun Yang and Tobias Wrigstad. "Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK". In: *ACM Transactions on Programming Languages and Systems* 44.4 (Dec. 31, 2022), pp. 1–34. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/3538532.

[401] Guang Yang, Jie Liu, Muzi Qu, Shuai Wang, Dan Ye, and Hua Zhong. "FaasRS: Remote Sensing Image Processing System on Serverless Platform". In: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. Madrid, Spain: IEEE, July 2021, pp. 258–267. ISBN: 978-1-6654-2463-9. DOI: 10.1109/COMPSAC51774.2021.00044.

[402] Yanning Yang, Dong Du, Haitao Song, and Yubin Xia. "On-Demand and Parallel Checkpoint/Restore for GPU Applications". In: *Proceedings of the ACM Symposium on Cloud Computing*. Redmond WA USA: ACM, Nov. 20, 2024, pp. 415–433. ISBN: 979-8-4007-1286-9. DOI: 10.1145/3698038.3698510.

[403] Zhaojun Yang, Jun Sun, Yali Zhang, and Ying Wang. "Understanding SaaS Adoption from the Perspective of Organizational Users: A Tripod Readiness Model". In: *Computers in Human Behavior* 45 (Apr. 2015), pp. 254–264. ISSN: 07475632. DOI: 10.1016/j.chb.2014.12.022.

[404] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. Boston, MA: USENIX Association, 2010, p. 10.

[405] Ilya Zaslavsky. "Service-Oriented Architecture". In: *Encyclopedia of GIS*. Ed. by Shashi Shekhar, Hui Xiong, and Xun Zhou. Cham: Springer International Publishing, 2017, pp. 1895–1896. ISBN: 978-3-319-17884-4. DOI: 10.1007/978-3-319-17885-1_1199.

[406] *Zend Enterprise PHP Development Platform*. URL: https://www.zend.com/ (visited on Nov. 22, 2025).

[407] Irene Zhang, Alex Garthwaite, Yury Baskakov, and Kenneth C. Barr. "Fast Restore of Checkpointed Memory Using Working Set Estimation". In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Newport Beach California USA: ACM, Mar. 9, 2011, pp. 87–98. ISBN: 978-1-4503-0687-4. DOI: 10.1145/1952682.1952695.

[408] Mingxue Zhang and Wei Meng. "JSISOLATE: Lightweight in-Browser JavaScript Isolation". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens Greece: ACM, Aug. 20, 2021, pp. 193–204. ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3468577.

[409] Sizhuo Zhang, Hari Angepat, and Derek Chiou. "HGum: Messaging Framework for Hardware Accelerators". In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. Cancun: IEEE, Dec. 2017, pp. 1–8. ISBN: 978-1-5386-3797-5. DOI: 10.1109/RECONFIG.2017.8279799.

[410] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. "On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications". In: *Proceedings of the Sixteenth European Conference on Computer Systems*. Online Event United Kingdom: ACM, Apr. 21, 2021, pp. 540–555. ISBN: 978-1-4503-8334-9. DOI: 10.1145/3447786.3456258.

[411] Diyu Zhou and Yuval Tamir. *HyCoR: Fault-Tolerant Replicated Containers Based on Checkpoint and Replay*. Version 1. 2021. DOI: 10.48550/ARXIV.2101.09584. Prepublished.

[412] Diyu Zhou and Yuval Tamir. "RRC: Responsive Replicated Containers". In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 85–100. URL: https://www.usenix.org/conference/atc22/presentation/zhou-diyu.

[413] *Zoom*. URL: https://www.zoom.com (visited on Nov. 28, 2025).

Biography

Ivan Ristović was born in 1995 in Užice, Serbia. He graduated from the Užice Gymnasium in 2014, receiving *Vuk Karadžić* award for distinguished performance. He finished BSc studies in September 2017, as part of the Informatics group at the Faculty of Mathematics in Belgrade. He finished MSc studies in July 2020. His MSc thesis titled "Language-invariant comparison of structurally-similar imperative code segments" was awarded an Annual Honorary Mention of the Mathematical Institute of the Serbian Academy of Sciences and Arts in the field of computing for MSc students. He enrolled in the PhD program at the Faculty of Mathematics in October 2020.

Since 2018 he works at the Faculty of Mathematics as a Teaching Assistant. He held practice classes for the following undergraduate courses: Object-oriented programming, Programming paradigms, Computer networks, and Functional programming. He held practice classes for the following master studies courses: Software development II and Software verification. In the period from June 2021 to April 2023 he worked as a Researcher, as part of the research collaboration between the Faculty of Mathematics and Oracle Labs research center in Belgrade. Since April 2023, in addition to his employment at the Faculty, he works as a Senior Researcher at Oracle Labs.

Biografija autora

Ivan Ristović je rođen 1995. godine u Užicu. Užičku gimnaziju završio je 2014. godine kao nosilac Vukove diplome. Osnovne akademske studije na smeru Informatika na Matematičkom fakultetu završio je u septembru 2017. godine. Master studije na Matematičkom fakultetu završio je u julu 2020. godine. Njegova master teza "Jezički-invarijantna provera strukturno sličnih segmenata imperativnog koda" je dobila Pohvalu Matematičkog instituta SANU za studente master studija u oblasti računarstva. Doktorske studije na Matematičkom fakultetu upisao je u oktobru 2020. godine.

Od 2018. godine zaposlen je na Matematičkom fakultetu. Na osnovnim studijama držao je vežbe iz predmeta Objektno orijentisano programiranje, Programske paradigme, Računarske mreže, i Funkcionalno programiranje. Na master studijama držao je vežbe iz predmeta Razvoj softvera 2 i Verifikacija softvera. U periodu od juna 2021. godine do aprila 2023. radio je kao istraživač na projektu saradnje Matematičkog fakulteta i kompanije Oracle. Od aprila 2023. godine, pored saposlenja na Matematičkom fakultetu, angažovan je i kao istraživač-senior u kompaniji Oracle.

Attachment 1.

Statement of Authorship

I, Author Ivan Ristović

Identifier 2013/2020

Declare

That the doctoral dissertation titled

Direct Data-Snapshotting and Snapshot Sharing Across Cloud-Native Applications

- is the result of my own research work,
- that the proposed dissertation, in whole or in parts, has not been submitted for obtaining any degree at other higher education institutions,
- that the results are correctly stated, and
- that I have not violated copyright or used the intellectual property of others.

Signature

In Belgrade, 10.04.2026.

Прилог 1.

Изјава о ауторству

Потписани-а Иван Ристовић

број индекса 2013/2020

Изјављујем

да је докторска дисертација под насловом

Директно снимање података и дељење снимака између апликација у облаку

- резултат сопственог истраживачког рада,
- да предложена дисертација у целини ни у деловима није била предложена за добијање било које дипломе према студијским програмима других високошколских установа,
- да су резултати коректно наведени и
- да нисам кршио/ла ауторска права и користио интелектуалну својину других лица.

Потпис докторанда

У Београду, 10.04.2026.

Ivan Ristović

Attachment 2.

Statement of Identity of Printed and Electronic Version of Doctoral Dissertation

Name and surname of the author Ivan Ristović

Identifier 2013/2020

Study program Informatics

Thesis title Direct Data-Snapshotting and Snapshot Sharing Across Cloud-Native Applications

Mentor prof. dr Milena Vujošević Janičić

I, the undersigned Ivan Ristović

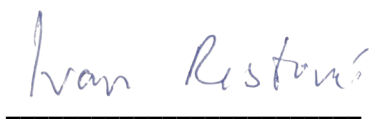
Declare that the printed version of my doctoral dissertation is identical to the electronic version submitted for publication in the **Digital Repository of the University of Belgrade**.

I allow the publication of my personal data related to obtaining the academic title of Doctor of Philosophy, such as name and surname, year and place of birth, and the date of thesis defense.

These personal data may be published on the web pages of the digital library, in the electronic catalog, and in publications of the University of Belgrade.

Signature

In Belgrade, 10.04.2026.



Прилог 2.

Изјава о истоветности штампане и електронске верзије докторског рада

Име и презиме аутора Иван Ристовић

Број индекса 2013/2020

Студијски програм Информатика

Наслов рада Директно снимање података и дељење снимака између апликација у облаку

Ментор проф. др Милена Вујошевић Јаничић

Потписани/а Иван Ристовић

Изјављујем да је штампана верзија мог докторског рада истоветна електронској верзији коју сам предао/ла за објављивање на порталу **Дигиталног репозиторијума Универзитета у Београду.**

Дозвољавам да се објаве моји лични подаци везани за добијање академског звања доктора наука, као што су име и презиме, година и место рођења и датум одбране рада.

Ови лични подаци могу се објавити на мрежним страницама дигиталне библиотеке, у електронском каталогу и у публикацијама Универзитета у Београду.

Потпис докторанда

У Београду, 10.04.2026.

Ivan Ristović

Appendix 3.

Statement of Use

I authorize the University Library "Svetozar Marković" to include my doctoral dissertation in the Digital Repository of the University of Belgrade, as my original work, titled:

Direct Data-Snapshotting and Snapshot Sharing Across Cloud-Native Applications

of which I am the sole author.

I have submitted the dissertation with all appendices in electronic format suitable for permanent archival.

My doctoral dissertation stored in the Digital Repository of the University of Belgrade may be used by anyone who respects the provisions of the selected Creative Commons license.

1. Attribution (CC BY)

2. Attribution – non-commercial (CC BY-NC)

3. Authorship – non-commercial – no adaptations (CC BY-NC-ND)

(4.) Authorship – non-commercial – shared under same terms (CC BY-NC-SA)

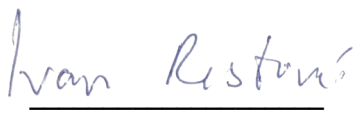
5. Authorship – no adaptations (CC BY-ND)

6. Authorship – shared under same terms (CC BY-SA)

(Select one option, details can be found on the Creative Commons website https://creativecommons.org)

Signature

In Belgrade, 10.04.2026.



Прилог 3.

Изјава о коришћењу

Овлашћујем Универзитетску библиотеку „Светозар Марковић“ да у Дигитални репозиторијум Универзитета у Београду унесе моју докторску дисертацију под насловом:

Директно снимање података и дељење снимака између апликација у облаку

која је моје ауторско дело.

Дисертацију са свим прилозима предао/ла сам у електронском формату погодном за трајно архивирање.

Моју докторску дисертацију похрањену у Дигитални репозиторијум Универзитета у Београду могу да користе сви који поштују одредбе садржане у одабраном типу лиценце Креативне заједнице (Creative Commons) за коју сам се одлучио/ла.

1. Ауторство

2. Ауторство - некомерцијално

3. Ауторство – некомерцијално – без прераде

4. Ауторство – некомерцијално – делити под истим условима

5. Ауторство – без прераде

6. Ауторство – делити под истим условима

(Молимо да заокружите само једну од шест понуђених лиценци, кратак опис лиценци дат је на полеђини листа).

Потпис докторанда

У Београду, 10.04.2026.

Ivan Ristović

1. Ауторство - Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце, чак и у комерцијалне сврхе. Ово је најслободнија од свих лиценци.

2. Ауторство – некомерцијално. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела.

3. Ауторство - некомерцијално – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца не дозвољава комерцијалну употребу дела. У односу на све остале лиценце, овом лиценцом се ограничава највећи обим права коришћења дела.

4. Ауторство - некомерцијално – делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца не дозвољава комерцијалну употребу дела и прерада.

5. Ауторство – без прераде. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, без промена, преобликовања или употребе дела у свом делу, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце. Ова лиценца дозвољава комерцијалну употребу дела.

6. Ауторство - делити под истим условима. Дозвољавате умножавање, дистрибуцију и јавно саопштавање дела, и прераде, ако се наведе име аутора на начин одређен од стране аутора или даваоца лиценце и ако се прерада дистрибуира под истом или сличном лиценцом. Ова лиценца дозвољава комерцијалну употребу дела и прерада. Слична је софтверским лиценцама, односно лиценцама отвореног кода.