Mosaic: Optimizing Cloud Resource Efficiency with Lazily-Packaged Application Modules

Serhii Ivanenko
serhii.ivanenko@tecnico.ulisboa.pt
INESC-ID, Instituto Superior Técnico, University of Lisbon
Lisbon, Portugal

Carlos Segarra
cs1620@ic.ac.uk
Imperial College London
London, United Kingdom

Rodrigo Bruno
rodrigo.bruno@tecnico.ulisboa.pt
INESC-ID, Instituto Superior Técnico, University of Lisbon
Lisbon, Portugal

Abstract

Modern cloud platforms require users to build their application code and package it with its run-time dependencies in a hardware-agnostic package like a container or VM image. Packaging applications *before* their deployment on their target hardware prevents users from leveraging optimized hardware or harnessing opportunities that appear at run-time, like application co-location for faster communication. This forces developers to preemptively package the same application for an ever-increasing universe of possible target architectures, bloating cloud storage and package registries.

We present Mosaic, our vision for a new modular architecture to build cloud applications that *delays* the packaging of applications until they are deployed on their target resources. Applications in Mosaic are composed of individual Cloud Modules: a language- and hardware-independent representation for application code and library dependencies, where each module offers a public API to communicate with other modules. We discuss a prototype implementation based on the WebAssembly instruction format and demonstrate its potential benefits to leverage hardware-optimized libraries to improve performance, bypass communication stacks for distributed applications that exhibit co-location, and reduce storage bloat through deduplication and sharing.

CCS Concepts

• **Software and its engineering** → **Runtime environments**; • **Computer systems organization** → **Cloud computing**.

Keywords

Cloud Computing, Microservices, Function-as-a-Service, Hardware Acceleration

ACM Reference Format:
Serhii Ivanenko, Carlos Segarra, and Rodrigo Bruno. 2025. Mosaic: Optimizing Cloud Resource Efficiency with Lazily-Packaged Application Modules. In *The 3rd Workshop on SErverless Systems, Applications and MEthodologies (SESAME' 25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3721465.3721864

SESAME' 25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1557-0/2025/03
https://doi.org/10.1145/3721465.3721864

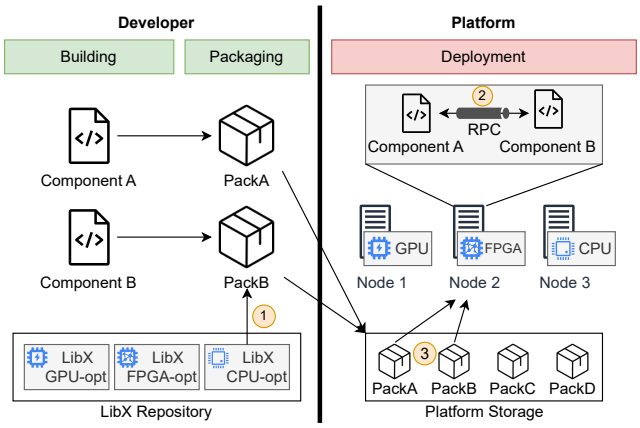

Figure 1: Inefficiencies in existing cloud platforms: (1) hardware-agnostic execution, (2) location-unaware communication, (3) dependency duplication.

1 Introduction

The development of modern applications can be separated into a build phase, where users prepare their application code, a packaging phase, where code gets bundled with its run-time dependencies, and a deployment phase (Fig. 1). When developing applications for execution in the cloud, users may not always know the hardware details of their target deployment platform either because they are using a managed service [32, 35] or an execution model that hides away these details like microservices [19] or serverless [26]. Cloud users are, as a consequence, forced to package applications in hardware-agnostic bundles like container [33] or VM images [39].

To reap the benefits of hardware specialization, users can package the same application for all supported target platforms, including combinations of CPU architectures [34], CPU extensions [38], compiler optimizations [2], hardware acceleration [36], etc. This task of specialization becomes increasingly harder as hardware heterogeneity increases, and applications become more complex and are packaged with other run-time dependencies [22, 28]. Users can either aim for the lowest common hardware features, giving up on performance (Fig. 1, 1), bloat package registries with many different versions of the same package (Fig. 1, 3), or target a very specific deployment, reducing the cloud's flexibility to manage the application and therefore increasing user's costs.

This problem is exacerbated when considering the deployment of a distributed application on cloud resources. Users do not know what is the best available communication fabric [1, 37] or if there

App Build Block	Building	Packaging	Deployment
Monoliths	Developer	Developer	Developer
Microservices	Developer	Developer	Platform
Functions	Developer	Developer	Platform
Cloud Modules	Developer	Platform	Platform

Table 1: Separation of responsibilities in cloud platforms.

are any kernel bypass mechanisms available (e.g., RDMA), neither do users know what will be the allocation of instances of the application to target resources, preventing co-located instances from using different communication channels like shared memory. Microservice applications, for example, communicate always using RPCs, including (de-)serialization and network stack traversals, even if co-located (Fig. 1, 2).

We observe that the source of these inefficiencies is that cloud platforms require developers to package their applications (such as monoliths, microservices, or serverless functions, deployed using the Infrastructure-as-a-Service, Container-as-a-Service, or Function-as-a-Service models, respectively) *before* they are deployed by the platform (Tab. 1). By instead letting the cloud platform decide how to package a set of application components, applications can harness the benefits of the best available resources at runtime. Existing solutions for run-time (re-)configuration, however, either require adopting a specific programming language [58] or programming framework [44], re-writing application code [63], or are specific to particular domains [3, 8, 9].

In this paper, we propose Mosaic, our vision for a new modular architecture for cloud applications. Applications in Mosaic are made up of independent Cloud Modules: a language- and hardware-agnostic intermediate representation with a narrow system interface. Each module has a public API to communicate with other modules. The target runtime environment can then *compose* modules into an application, optimizing for the available hardware resources, and provide an implementation for the system interface. Mosaic is not confined to the existing cloud service models and has the potential to improve them significantly or even pave the way to a new service offering.

Our Mosaic prototype uses WebAssembly (Wasm) [57] as an intermediate representation, and we plan on leveraging Wasm's system interface (WASI) [31], and Wasm's component model for the module's interface [29]. We believe Wasm's language- and hardware-independence, together with its mature runtime [40, 41] and compilation environment, provides the necessary tools to achieve our goals transparently to application code, but we also envision research challenges (§5).

Our early experiments to quantify the performance gap that could be covered by Mosaic demonstrate that application performance can vary significantly by selecting modules optimized for specific hardware extensions. In particular, running AVX-enabled TensorFlow versus off-the-shelf TensorFlow yields a score improvement of 2.40× and 2.67× for ML inference and training (respectively) when running the AI benchmark suite [4] (§2.1). Co-located microservices can improve throughput by 12-35% (§2.2), and dependency de-duplication can vastly reduce storage bloat (§2.3).

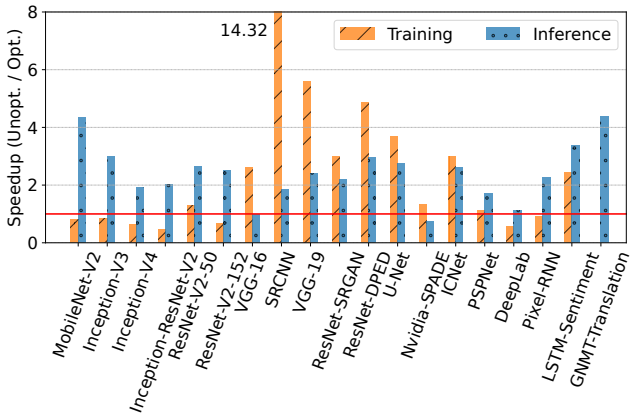

Figure 2: ML Training and Inference speedup comparing TensorFlow installed with `pip` and an optimized build. The red line delineates the improvement threshold.

2 Inefficiencies in Modern Cloud Platforms

As modern cloud platforms become more complex and heterogeneous, developers have less control and visibility over the underlying hardware and deployment location. While this tendency allows developers to focus on application logic and facilitates scalability, it also forces developers to build and package their applications conservatively, without being able to take advantage of local hardware accelerators or bypass network if application components are co-located. Moreover, since cloud platforms require developers to package applications before being uploaded to the platform, cloud storage easily becomes bloated with many copies of popular application libraries and frameworks (NumPy [22] and PyTorch [24] as examples of widely used Python libraries) included many different container and VM images. In this section, we experimentally quantify each of these inefficiencies.

2.1 Hardware-agnostic Execution

Some types of computationally intensive workloads, such as machine learning, data analytics, cryptography, and computer graphics, can benefit from special high-performance hardware units like AI and/or cryptographic co-processors, SIMD support such as AVX, or GPU and FPGA accelerators. As a result, most popular libraries and frameworks that support such workloads offer versions specialized to specific hardware accelerators, in addition to general-purpose builds that run on most commodity hardware. Examples include TensorFlow [28], PyTorch [25], CuPy [14], OpenCV [23].

In the presence of hardware acceleration, it is desirable to use versions of code that are specific to that hardware, as it may improve performance compared to versions targeting generic commodity hardware. To measure this performance impact, we run an AI benchmark suite [4] composed of a total of 19 Machine Learning and Computer Vision benchmarks. The benchmarks in the suite use TensorFlow [28] and report the inference and training times. The experiment uses a TensorFlow build installed via Python's package manager `pip` (*Unopt.*), and a TensorFlow build optimized for Intel

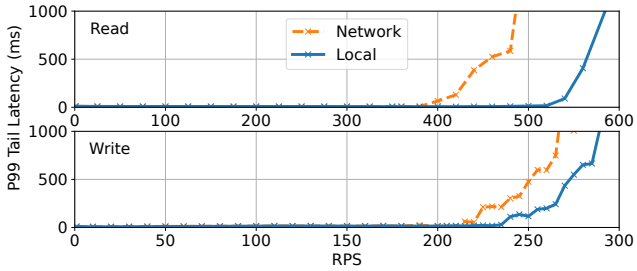

Figure 3: Social Network end-to-end latency using local and network communication between services.

CPUs with AVX support[1] (*Opt.*). The experiment was executed on a local cluster machine running Ubuntu 22.04.4 LTS (Linux kernel 5.15.0-97-generic) equipped with 2x Intel Xeon Gold 5320 CPU @ 2.2GHz and 128GB of DDR4 DRAM. Results in Fig. 2 reveal that an optimized build of TensorFlow improves inference by 2.40× and training by 2.67× compared to an unoptimized build on the same exact hardware. It is also interesting to note that while most benchmarks benefit from hardware acceleration, some do not, and some may even suffer performance degradation when accelerated. This experiment demonstrates that anticipating which type of hardware to use is nontrivial as it may depend on the specific workload.

Unawareness concerning the deployment hardware and the need for packaging distributed applications before uploading to the cloud storage prevent developers from taking full advantage of hardware accelerators, even if such specialized hardware is installed in the actual execution environment. Developers are forced to always target generic hardware so that their application would work regardless of the platform's deployment decision. This is a missed opportunity to optimize cloud applications that perform computationally intensive workloads. The opposite, allowing developers to target a specific type of hardware, is also non-optimal in terms of resource utilization as it would constrain the platform on which hardware it can use to deploy the application. In sum, statically bundling applications with a specific implementation that targets generic or specialized hardware fails to offer both application performance and resource efficiency.

2.2 Locality-unaware Communication

Distributed application architectures such as microservices have become popular for building and deploying distributed applications. With this approach, applications are built as sets of individual and loosely coupled logic components, called services, which communicate with each other through APIs. To facilitate communication, microservice frameworks employ various Remote Procedure Call (RPC) mechanisms, such as Apache Thrift [7] or gRPC [17].

However, developers cannot make any assumptions regarding the deployment location of each service since they build and package services before the platform takes deployment decisions. This affects the way services communicate with each other: all inter-service communication should take place over the network to accommodate all deployment scenarios. Consequently, even if two services happen to be deployed on a single compute instance, they still require serialization and go through the network stack.

[1]We used the following Docker image: intel/intel-optimized-tensorflow-avx512.

Benchmark	App (KBs)	Deps. (%)
dynamic-html (Python)	3.02	976.47 (99)
uploader (Python)	2.86	0.0 (0)
thumbnailer (Python)	4.08	7011.87 (99)
dynamic-html (JS)	3.07	143.66 (97)
uploader (JS)	2.63	3801.98 (99)
thumbnailer (JS)	2.58	23298.43 (99)
video-processing (Python)	17.38	86839.41 (99)
compression (Python)	3.60	0.00 (0)
image-recognition (Python)	40.22	247749.31 (99)
graph-pagerank (Python)	2.39	9291.69 (99)
graph-mst (Python)	2.41	9291.69 (99)
graph-bfs (Python)	2.39	9291.69 (99)
dna-visualisation (Python)	3.14	177362.42 (99)
SocialNetwork (C++)	1454.26	527.54 (26)
MediaMicroservices (C++)	1302.35	527.88 (28)
HotelReservation (Go)	6.73	4844.0 (99)

Table 2: Size comparison of applications and their dependencies from SeBS [50] and DeathStarBench [55].

Communicating over the network implies making system calls to open a socket, establishing a connection, and transferring data through this connection. If services use secure channels for communication, then data encryption is included in this sequence. Additionally, network communication can also entail the expensive process of data serialization. These steps contribute to the end-to-end latency of the request, which could be significantly reduced if the co-located components used local communication.

Fig. 3 shows the end-to-end latency (99th percentile) and throughput of read and write workloads in the Social Network microservice application from DeathStarBench [55] for two modes of communication between the services: network and local calls. For the network calls, services use Apache Thrift [7]; local calls are direct method invocations, representing the theoretical upper bound of performance by bypassing (de-)serialization and the TCP stack. For both modes, all components of Social Network run in the same machine, but in the network mode, each service executes in its container, whereas in the local mode, all services are merged in a single process running in a container. Results taken in the same evaluation environment as in the previous experiment demonstrate that for read and write workloads, the local communication mode can sustain 12% and 35% (respectively) higher request rate without degrading the tail latency compared to network communication.

2.3 Application Dependencies Duplication

Application dependencies (commonly referred to as libraries) are essential to abstract all sorts of tasks, such as media processing, cryptography, data analysis, etc. Such dependencies are commonly distributed at build-time through repositories or package managers, such as Apache Maven [6], NuGet [21], npm [20], etc.

In order to use external libraries in a cloud platform setting, applications are packaged together with libraries in a single application package after building the application. However, these dependencies can take up a significant portion of the overall package size. Tab. 2 compares the size of the application code with the size of the dependencies. The applications used in this experiment are the benchmarks from the SeBS [50] and DeathStarBench [55] benchmark suites. This experiment shows that dependencies constitute more than 99% of the overall application binary size for the majority of benchmarks. Dependencies dominate the application package size, thus imposing storage and network overhead. Besides, booting such an application involves loading and initializing all dependencies during instantiation, contributing to startup latency. Deduplicating common dependencies is possible; however, it comes at a cost of extra computational effort to run deduplication algorithms [47].

Takeaway. The current separation of concerns between developers and cloud platforms significantly limits application performance and resource efficiency. Developers are oblivious to the platform's hardware availability and deployment decisions, thus unable to avoid unnecessary inefficiencies by using hardware-specific code or optimal communication mechanisms. On the other hand, platforms that allow developers to target specific hardware have limited deployment options for already packaged applications and, therefore unable to optimize resource utilization.

3 Existing Cloud Platform Optimizations

Dynamically optimizing performance and resource consumption is a hot research topic in both academia and industry. Existing approaches can be divided into several categories:

Cloud Services such as Machine Learning-as-a-Service [78] or other Platform-as-a-Service offerings assume the role of packaging applications for resource efficiency. Various systems [3, 8, 9, 42, 51, 70] offer machine learning and artificial intelligence abstractions. However, these services are specific to their domains and require applications to be re-architected according to the service interface and execution model. In exchange, platforms have control over the code packaging of the applications. Similarly, data analytics platforms such as Dryad [59], Hadoop [52], and Spark [81] are solutions designed for parallel large-scale data-intensive distributed workloads which can perform various optimizations to improve data locality, thus reducing latency. In contrast, our proposal aims at enabling optimizations for general-purpose applications without requiring applications to be rewritten against a specific interface.

Application-level Programming Models and Frameworks have also received significant attention. Nu [71] proposes the notion of elastic logical processes and proclets to fully utilize fungible resources. Dandelion [63] leverages explicit distinction computation and I/O to optimize scheduling and employ lightweight secure execution sandboxes. Ghemawat et al. [56] proposes building applications as logical monoliths according to a specific programming model so that the runtime can dynamically make efficient deployments. Blueprint [44] also proposes its programming model to optimize communication between microservices. However, Blueprint does not automatically reconfigure communication protocols based on the application workload. Orleans [48] proposes an actor-based framework that abstracts deployment details from developers. Jolie [67] and Silvera [74] are full domain-specific programming languages for service-oriented or microservices applications. Compared to our vision, however, these works imply a complete or partial re-engineering of existing applications and their dependencies, which is not always possible, particularly if the application relies on external libraries such as ML frameworks.

Optimized Runtime Systems have also been used to take advantage of locality in distributed applications. CoFaaS [45] proposes consolidating serverless functions on the same compute node and avoiding the network layer by transparently transforming RPC invocations to local calls. Faasm [73] and Nightcore [60] facilitate the locality of related functions to benefit from local communication. Palette [43] offers a user-guided mechanism for promoting data locality in serverless. While these approaches focus on optimizing communication between components, they ignore other inefficient aspects of application execution in the cloud, such as the possibility of hardware-accelerated execution and dependency deduplication.

Just-In-Time (JIT) Compilation is a promising approach to dynamically optimize the application code for the locally available hardware. For example, TornadoVM [53] offers multiple compilation backends that target specific hardware accelerators, allowing Java programs to benefit from hardware-accelerated execution. However, JIT compilation also incurs significant costs in terms of profiling and compilation, which can become prohibitive for latency-sensitive applications. Besides, microservice and serverless workloads tend to be transient and short-lived [61, 72, 77], and their JIT-compiled code caches may not persist across invocations. This leads to repeated profiling and JIT compilation of the same code, leading to unpredictable latency overheads [49, 62].

Storage deduplication is a technique aimed at mitigating storage overhead by sharing common components, such as image layers or dependencies, across packaged applications. Brooker et al. [47], SOCK [68], Pagurus [65], and RainbowCake [80] feature strategies to deduplicate container image layers, allowing them to achieve reduced storage overhead and faster cold starts. However, existing strategies either apply coarse-grained deduplication [65, 68, 80] (container layer-level deduplication) or need to run expensive deduplication mechanisms to duplicate at the block level [47].

4 Mosaic Design and Architecture

We envision a new cloud environment where developers build applications into a set of Cloud Modules, which are then packaged and deployed by a cloud platform (see Fig. 4, notice that the cloud abstraction shifted to include packaging), offering transparent application acceleration to developers and the opportunity to optimize resource efficiency to platforms. Standalone modules export interfaces that can be used to interact with other modules. By exposing modules and their interaction directly to the cloud environment, Mosaic allows cloud platforms to intercept and offer modules that better suit the locally available hardware, use the most efficient communication medium for inter-module interaction, and deduplicate application storage. The rest of this section further elaborates on this vision.

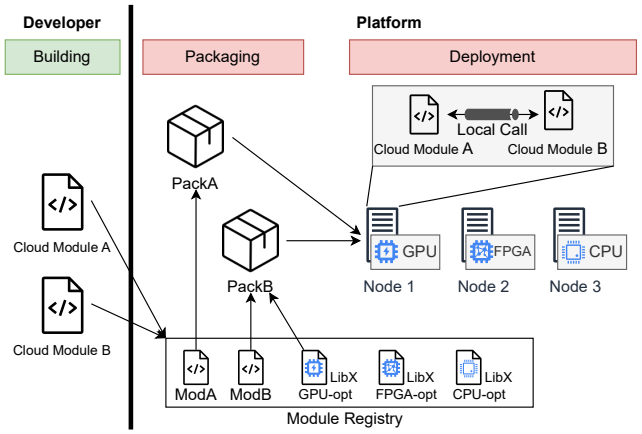

Figure 4: Application deployed on Mosaic. Developers push Cloud Modules to the Module Registry. The platform packages the application dynamically, selecting specialized versions of modules from the registry and connecting them with appropriate communication mechanisms based on the actual deployment scenario.

4.1 Cloud Modules

In Mosaic, applications are composed of a set of Cloud Modules. Modules are atomic deployable units that are loaded on demand from the Module Registry (§4.3) based on the available hardware. Deciding which hardware to use is an on-going research challenge discussed in §5. Module lazy loading also allows Mosaic to dynamically choose an appropriate communication mechanism (local or remote) between modules (remote communication can also be accelerated based on local network devices). Modules can contain a mix of code and data. Each module is given access to a private copy of loaded dependency modules, such as libraries. We envision that co-located dependencies could be deduplicated in memory through Copy-on-Write.

Mosaic does not require developers to re-architect existing application components, such as microservices, serverless functions, libraries, or even static files, using a new programming model or framework. Instead, modules can wrap the already existing artifacts that applications produce during compilation. For instance, application binaries and library dependencies are treated as independent modules in Mosaic. Moreover, any file that is part of the deployment package (for example, container image) that developers upload to a cloud platform is now also handled as an independent module.

4.2 Module System Interface

Cloud Modules may need to interact with each other or with the external environment to, for example, access cloud storage. We start from the insight that most cloud applications need a system interface to communicate or to download input data and upload output data. The execution environment offered in most cloud platforms (even in serverless platforms such as Amazon Lambda) is unnecessarily complex, allowing applications to access the entire POSIX interface and, as a consequence, too expensive (long initialization time and high memory footprint) as a fully isolated VM or container is necessary to support such a wide and powerful interface. Moreover, existing interfaces also expose system-level information directly to applications (e.g., applications can directly access network devices and file descriptors), which hampers the adoption of dynamic environments such as Mosaic.

We therefore adopt a new system interface (more details in §5) with two main design features. First, it is orthogonal to the sandboxing mechanism, i.e., it should support modules being deployed on different VMs, containers, or even in the same process. In-process sandboxing is particularly important to achieve high module density (low memory footprint), high elasticity (low initialization time), and to support fast inter-module communication. Second, it exposes an opaque interface, which does not disclose low-level system information, thus promoting state-of-the-art optimizations such as application migration and checkpoint/restore.

Mosaic's system interface also mediates inter-module communication. Similarly to how shared libraries expose functions, modules export a public module interface that can be used to interact with other modules. Application code calling into a library is now executes as a module calling another module through a handle obtained from Mosaic's system interface. Modules can also communicate over sockets (e.g., two microservices). The callee module can either be local or remote, and the handle hides the communication protocol. For data-only modules (for example, files), Mosaic exposes a file-based module interface that supports normal file operations (reading, writing, etc). Interactions with external components, such as communication with a persistent blob storage or a database, are handled as usual — through conventional network connections. Optimizing module communication is an active research challenge further detailed in §5.

On top of that, we envision the concept of streams for Cloud Modules to communicate with each other. Modules use the system interface to open a stream to an external service. After the stream is opened by the runtime, a handle is returned to the application code and can be used to read/write data from/to it. Streams completely hide the underlying communication mechanism and do not expose system-level information about the origin and destination. Such design, however, requires existing applications to be adapted to streams. While not being part of the current Mosaic's design, streams can become a new communication abstraction specifically designed for highly dynamic cloud environments.

4.3 Module Registry

The registry operates similarly to VM disk or container image registries in the sense that developers can upload artifacts that are later deployed by the cloud platform. The main difference compared to VM disks and container images — that are opaque to the platform — is that Cloud Modules export a public interface that platforms can identify. In Mosaic, platforms can deduplicate modules that expose the same interface by design. For example, the registry may have a single version of NumPy [22] (a popular dependency in Python applications), avoiding many duplicated versions packaged for each application. Furthermore, platforms can also offer specialized versions of modules optimized for locally available hardware. For example, platforms can offer Machine Learning modules (for

example, PyTorch [24] or TensorFlow [28]) built specifically for locally available accelerators or for specialized CPU versions available on the data center.

This separation of concerns between developers and platforms also improves resource utilization as platforms can dynamically decide which hardware to use not only based on performance but also on hardware availability. We further envision that this separation could open the door to new optimizations on popular packages (for example, Python's NumPy) and foster the adoption of new accelerators as platforms do not need to rely on developers to package applications against recent software optimized for recent hardware.

Finally, lazily loading and deduplicating modules in the registry also reduces the infrastructure burden to deploy applications, which often entails the repeated distribution of the same file over and over for different applications or instances of the same application.

5 Early Prototype and Research Challenges

We are developing an early prototype of the platform and runtime that implements our vision of Mosaic. The runtime is written in the Rust programming language and uses WebAssembly [57] (Wasm for short) as a binary code format for application modules. Wasm is a popular binary code format designed around three main principles: portability, security, and performance. Moreover, the Wasm community has been standardizing inter-module interfaces and system interfaces, which greatly fit the goals of this project. In particular, the WebAssembly System Interface [31] (WASI) has been recently proposed as a secure standard system interface for Wasm modules. The Wasm Component Model [29] has also recently been proposed to offer a standardized format to describe module dependencies and interfaces. Both the component model and WASI are being used as building blocks for designing cloud modules interface and the runtime engine.

Wasm is also a suitable compilation target for cloud applications thanks to its language- and platform-independence. Wasm is a narrow software stack waste, allowing applications written in multiple languages to use it as a common compilation target, thus requiring no developer intervention to support Mosaic. Besides, Wasm is a community-driven ecosystem that is gaining increasing popularity. There is great momentum in both academia and industry to improve Wasm support not only for Rust, C/C++, and Go [45, 54] but also for managed languages such as Java, Python, and JavaScript [5, 10–12, 18, 27, 73]. Companies such as Cloudflare [13], Fastly [15], and Fermyon [16] already support uploading Wasm code for execution in their cloud.

While Wasm offers unique support for Mosaic's portability and module sandboxing, it does not address several system aspects related to infrastructure management. We dedicate this section to discussing the open challenges we uncovered while building our early prototype.

Quantifying Module Efficiency on Different Hardware. Deciding where a module should be deployed requires the platform to automatically estimate which hardware will lead to higher resource efficiency. Traditionally, developers either rent specific cloud hardware to run an application or leave this decision to the platform, which in turn may deploy user code on any available commodity hardware. In sum, existing platforms focus primarily on optimizing resource usage. In Mosaic, however, platforms may be able to use optimized modules for specific hardware. This new dimension turns the resource usage optimization problem into a resource efficiency optimization one, a more complex optimization problem that requires estimating the efficiency of a particular module on specific hardware. Optimizing resource efficiency requires capturing the trade-off between performance and resource usage, as the goal of the platform is to improve application performance using the least amount of resources. This is an interesting but challenging research question as the efficiency of a module depends not only on the hardware where it runs but also on the workload intensity. For example, multiplying small matrices may yield very low resource efficiency when deployed on a large GPU. Existing work on resource provisioning and scheduling has focused mostly on vertical and horizontal scaling and has not investigated the impact of different hardware [46, 76].

Minimizing Module Communication Overheads. Modules may communicate with each other through existing networking primitives (e.g., sockets). Even if two modules are co-located, the application code still serializes data and ships it to the network stack. Bypassing the network stack is possible by modifying Mosaic's runtime WASI implementation to intercept the data-path between two co-located modules. For example, Mosaic may intercept a write to a socket and cache the data in memory until a read is requested. Bypassing serialization is, however, more challenging as it happens within the application code before it calls into networking primitives. Previous works describe how to minimize the overhead of serialization, particularly in managed languages [66, 75] or proposed frameworks and Domain-Specific Languages that developers would have to use to bypass serialization [44, 56, 58, 63]. In this work, we are investigating an alternative that does not require developer effort to completely bypass the serialization overhead. Building on the fact that Mosaic's applications are compiled into Wasm, the goal is to statically analyze the application code around the data communication sinks. If communication is local and serialization is to be avoided, the runtime should provide a specialized communication handle that directly invokes another module by copying data directly from the sending module and bypassing the application code that performs serialization. This optimization will be conducted at module loading time and will involve the transformation of the application code to bypass serialization. This optimization is made possible by Just-In-Time instrumenting Wasm code to fuse a data source to a data sink.

WebAssembly Acceleration Support. Mosaic is designed so that developers submit Wasm modules into the platform registry. Wasm modules are portable and can be executed on any hardware that supports a Wasm runtime. Platforms may also provide specialized module implementations that optimize for specific hardware but that still respect the module interface (for example, platforms may have special implementations of math libraries). Wasm modules can be Ahead-of-Time (AOT) compiled to a specific hardware architecture or Just-In-Time (JIT) compiled at run-time. This decision is left to the cloud platform and is invisible to users. We are currently investigating a non-trivial trade-off between keeping a large pool of AOT compiled modules for each specific hardware platform or paying the performance penalty of JIT compiling modules on the critical path. Finally, it is important to note that developer

code that relies on very specific hardware features and cannot be compiled into Wasm will naturally not be supported in Mosaic. Such applications will not be able to benefit from delayed packaging since they only target a specific hardware architecture.

Handling Security and Trust. Wasm offers a robust Software-Fault Isolation (SFI) security model combining a linear memory with control-flow integrity (CFI) enforced through checks at build and run-time [30]. However, Wasm's isolation guarantees can fail due to software bugs in the compiler and runtime [64]. We intend to combine Wasm's SFI with hardware-based memory isolation techniques such as Intel Memory Protection Keys [69] and hardware-base control flow integrity such as Intel CET [79].

Besides inter-module isolation, trust is also a crucial issue as developers now rely on the platform to select modules to be deployed together with developer code. Furthermore, modules may be packaged with specialized modules whose source code may not even be available. In these scenarios, developers need a mechanism to ensure that their data is secure and that the computation is correct. We plan to investigate ways to attestate modules and to allow developers to opt out of modules that cannot be attested.

6 Conclusion

This paper proposes Mosaic, an execution environment in which developers delegate packaging and deployment to the platform, allowing it to optimize application performance and hardware utilization. We elaborate on this vision and discuss a number of active research challenges that we are pursuing to implement Mosaic.

Acknowledgments

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UIDB/50021/2020 and through the FCT scholarship 2024.01902.BD, and partially funded by the European Union through the Horizon Europe projects CloudStars (101086248) and CloudSkin (101092646). Serhii's research was also supported in part by grants from Oracle.

References

[1] 1981. RFC 793 - Transmission Control Protocol. https://datatracker.ietf.org/doc/html/rfc793. Accessed: 2025-02-07.
[2] 2021. Intel march. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/march.html. Accessed: 2025-02-07.
[3] 2024. AI and Machine Learning at Google Cloud. https://cloud.google.com/solutions/ai. Accessed: 2025-02-07.
[4] 2024. AI Benchmark. https://ai-benchmark.com. Accessed: 2025-02-07.
[5] 2024. Announcing py2wasm: A Python to Wasm compiler. https://wasmer.io/posts/py2wasm-a-python-to-wasm-compiler. Accessed: 2025-02-07.
[6] 2024. Apache Maven Project. https://maven.apache.org. Accessed: 2025-02-07.
[7] 2024. Apache Thrift. https://thrift.apache.org. Accessed: 2025-02-07.
[8] 2024. AWS Machine Learning. https://aws.amazon.com/ai/machine-learning. Accessed: 2025-02-07.
[9] 2024. Azure Machine Learning. https://learn.microsoft.com/en-us/azure/machine-learning. Accessed: 2025-02-07.
[10] 2024. Bringing Python to Workers using Pyodide and WebAssembly. https://blog.cloudflare.com/python-workers. Accessed: 2025-02-07.
[11] 2024. Bytecoder. https://mirkosertic.github.io/Bytecoder. Accessed: 2025-02-07.
[12] 2024. CheerpJ. https://cheerpj.com. Accessed: 2025-02-07.
[13] 2024. Cloudflare: Connect, protect and build everywhere. https://www.cloudflare.com. Accessed: 2025-02-07.
[14] 2024. CuPy. https://cupy.dev. Accessed: 2025-02-07.
[15] 2024. Fastly: Powering the best of the internet. https://www.fastly.com. Accessed: 2025-02-07.
[16] 2024. Fermyon. https://www.fermyon.com. Accessed: 2025-02-07.
[17] 2024. gRPC. https://grpc.io. Accessed: 2025-02-07.
[18] 2024. Making JavaScript run fast on WebAssembly. https://bytecodealliance.org/articles/making-javascript-run-fast-on-webassembly. Accessed: 2025-02-07.
[19] 2024. Microservice Architecture. https://microservices.io. Accessed: 2025-02-07.
[20] 2024. npm. https://www.npmjs.com. Accessed: 2025-02-07.
[21] 2024. NuGet Gallery. https://www.nuget.org. Accessed: 2025-02-07.
[22] 2024. NumPy. https://numpy.org. Accessed: 2025-02-07.
[23] 2024. OpenCV CUDA. https://opencv.org/platforms/cuda. Accessed: 2025-02-07.
[24] 2024. PyTorch. https://pytorch.org. Accessed: 2025-02-07.
[25] 2024. PyTorch Hardware-Accelerated Video Decoding and Encoding. https://pytorch.org/audio/0.13.1/hw_acceleration_tutorial.html. Accessed: 2025-02-07.
[26] 2024. Serverless Architecture Overview. https://www.datadoghq.com/knowledge-center/serverless-architecture. Accessed: 2025-02-07.
[27] 2024. TeaVM. https://teavm.org. Accessed: 2025-02-07.
[28] 2024. TensorFlow. https://www.tensorflow.org. Accessed: 2025-02-07.
[29] 2024. WebAssembly Component Model design and specification. https://github.com/WebAssembly/component-model. Accessed: 2025-02-07.
[30] 2024. WebAssembly Security. https://webassembly.org/docs/security. Accessed: 2025-02-07.
[31] 2024. WebAssembly System Interface. https://wasi.dev. Accessed: 2025-02-07.
[32] 2025. Azure Kubernetes Service. https://azure.microsoft.com/en-us/products/kubernetes-service. Accessed: 2025-02-07.
[33] 2025. Docker. https://www.docker.com. Accessed: 2025-02-07.
[34] 2025. Docker Multi-arch build and images. https://www.docker.com/blog/multi-arch-build-and-images-the-simple-way. Accessed: 2025-02-07.
[35] 2025. Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine. Accessed: 2025-02-07.
[36] 2025. NVIDIA CUDA-X Libraries. https://developer.nvidia.com/gpu-accelerated-libraries. Accessed: 2025-02-07.
[37] 2025. The NVIDIA Quantum InfiniBand Platform. https://www.nvidia.com/en-us/networking/products/infiniband. Accessed: 2025-02-07.
[38] 2025. OpenSSL x86_64 processor capabilities vector. https://docs.openssl.org/3.4/man3/OPENSSL_ia32cap. Accessed: 2025-02-07.
[39] 2025. QEMU. https://www.qemu.org. Accessed: 2025-02-07.
[40] 2025. Wasmtime. https://wasmtime.dev. Accessed: 2025-02-07.
[41] 2025. WebAssembly Micro Runtime. https://bytecodealliance.github.io/wamr.dev. Accessed: 2025-02-07.
[42] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265—-283.
[43] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. 2023. Palette Load Balancing: Locality Hints for Serverless Functions. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 365–380. https://doi.org/10.1145/3552326.3567496
[44] Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. 2023. Blueprint: A Toolchain for Highly-Reconfigurable Microservice Applications. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 482–497. https://doi.org/10.1145/3600006.3613138
[45] Truls Asheim, Magnus Jahre, and Rakesh Kumar. 2024. CoFaaS: Automatic Transformation-based Consolidation of Serverless Functions. In *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies* (Athens, Greece) *(SESAME '24)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3642977.3652093
[46] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. 2023. With Great Freedom Comes Great Opportunity: Rethinking Resource Allocation for Serverless Functions. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 381–397. https://doi.org/10.1145/3552326.3567506
[47] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand Container Loading in AWS Lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 315–328. https://www.usenix.org/conference/atc23/presentation/brooker
[48] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2038916.2038932
[49] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. 2021. From warm to hot starts: leveraging runtimes for the serverless era. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS*

'21). Association for Computing Machinery, New York, NY, USA, 58–64. https://doi.org/10.1145/3458336.3465305

[50] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 64–78. https://doi.org/10.1145/3464298.3476133

[51] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[52] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (jan 2008), 107––113. https://doi.org/10.1145/1327452.1327492

[53] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. 2019. Dynamic application reconfiguration on heterogeneous hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Providence, RI, USA) *(VEE 2019)*. Association for Computing Machinery, New York, NY, USA, 165–178. https://doi.org/10.1145/3313808.3313819

[54] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 265––279. https://doi.org/10.1145/3423211.3425680

[55] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3––18. https://doi.org/10.1145/3297858.3304013

[56] Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, and Amin Vahdat. 2023. Towards Modern Development of Cloud Applications. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 110–117. https://doi.org/10.1145/3593856.3595909

[57] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185––200. https://doi.org/10.1145/3062341.3062363

[58] Cunchen Hu, Chenxi Wang, Sa Wang, Ninghui Sun, Yungang Bao, Jieru Zhao, Sanidhya Kashyap, Pengfei Zuo, Xusheng Chen, Liangliang Xu, Qin Zhang, Hao Feng, and Yizhou Shan. 2023. Skadi: Building a Distributed Runtime for Data Systems in Disaggregated Data Centers. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems* (Providence, RI, USA) *(HOTOS '23)*. Association for Computing Machinery, New York, NY, USA, 94–102. https://doi.org/10.1145/3593856.3595897

[59] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (Lisbon, Portugal) *(EuroSys '07)*. Association for Computing Machinery, New York, NY, USA, 59–72. https://doi.org/10.1145/1272996.1273005

[60] Zhipeng Jia and Emmett Witchel. 2021. 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) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[61] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, and Adam Barker. 2023. How Does It Function? Characterizing Long-term Trends in Production Serverless Workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 443–458. https://doi.org/10.1145/3620678.3624783

[62] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. 2024. Pronghorn: Effective Checkpoint Orchestration for Serverless Hot-Starts. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 298–316. https://doi.org/10.1145/3627703.3629556

[63] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/3620678.3624648

[64] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[65] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[66] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 132–147. https://doi.org/10.1145/3627703.3629568

[67] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. 2013. Service-oriented programming with Jolie. In *Web Services Foundations*. Springer New York, 81–107. https://doi.org/10.1007/978-1-4614-7518-7_4

[68] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 57–70. https://www.usenix.org/conference/atc18/presentation/oakes

[69] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 241––254.

[70] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[71] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. 2023. Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1409–1427. https://www.usenix.org/conference/nsdi23/presentation/ruan

[72] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[73] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 28, 15 pages. https://www.usenix.org/conference/atc20/presentation/shillaker

[74] Alen Suljkanović, Branko Milosavljević, Vladimir Inđić, and Igor Dejanović. 2022. Developing Microservice-Based Applications Using the Silvera Domain-Specific Language. *Applied Sciences* 12, 13 (2022), 6679. https://doi.org/10.3390/app12136679

[75] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. 2021. Naos: Serialization-free RDMA networking in Java. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 1–14. https://www.usenix.org/conference/atc21/presentation/taranov

[76] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 30, 14 pages. https://doi.org/10.1145/3342195.3387517

[77] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. 2021. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 443–457. https://www.usenix.org/conference/atc21/presentation/wang-ao

[78] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 945–960. https://www.usenix.org/conference/nsdi22/presentation/weng

[79] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. 2022. CETIS: Retrofitting Intel CET for Generic and Efficient Intra-process Memory Isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA,

2989––3002. https://doi.org/10.1145/3548606.3559344

[80] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container Caching and Sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 335–350. https://doi.org/10.1145/3617232.3624871

[81] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 15–28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia