PARSEC: An Adaptive and Efficient Platform for Reducing Cold Start in Serverless Computing

Nicolas Buitrago, 1, Hector Camacho, 1, Miguel Jimeno, *Senior Member, IEEE,* 1, Cesar Viloria-Nuñez, *Senior Member, IEEE,* 3, Jairo Cardona, 2, Augusto Salazar 1.

[1]Systems Engineering Department, Universidad del Norte, Barranquilla, Colombia (e-mail: nicolasbuitrago, trujilloh, majimeno, augustosalazar@uninorte.edu.co)

[2]Electrical Engineering Department, Universidad del Norte, Barranquilla, Colombia (e-mail: jacardona@uninorte.edu.co)

[3]School of Digital Transformation, Universidad Tecnológica de Bolívar, Colombia (e-mail: cesarvilorian@utb.edu.co)

***Abstract*—Serverless computing has revolutionized application development but faces a significant challenge: cold starts, which introduce delays when a function is called after a period of inactivity. Addressing these delays is crucial because they affect efficiency, performance, cost, and scalability. Existing mitigation strategies come with trade-offs, such as increased resource overhead and the need for precise resource management predictions. Also, optimizing the function startup process requires detailed knowledge of the runtime characteristics and the isolation technique used, such as using a container-based or a micro virtual machine setup. This work presents PARSEC, a comprehensive solution for cold start issues in serverless computing. By focusing on reducing initialization latency in idle containers, this research seeks to preserve scalability and ease of deployment features of serverless computing while overcoming cold start limitations. The proposed architecture improved cold start by streamlining the initialization of containers to reduce overhead. This involves minimizing unnecessary operations and customizing launches for serverless needs, aiming for a faster and more efficient setup. It also enhances the provisioning of Zygotes to speed up sandbox launches. The results show better performance for PARSEC when compared with other architectures, particularly at shorter wait times, suggesting effective cold start management. The strategic management of Zygotes and their provisioning scaling plays a critical role in managing large numbers of packages and instances, thereby enhancing the performance of package management. The cache system also evolves to become more selective, reducing overhead by focusing on essential packages.**

***Index Terms*—Article submission, IEEE, IEEEtran, journal, LaTeX, paper, template, typesetting.**

I. Introduction

THE serverless cloud model, introduced by Amazon Lambda in 2014 and later adopted by other major companies, encompasses Backend as a Service (BaaS) and Function as a Service (FaaS) [1] [2]. BaaS provides services such as user management and storage, while FaaS allows developers to run code on a provider-managed infrastructure, focusing on application logic [3]. Despite its benefits, serverless computing faces challenges such as cold start delays, debugging difficulties, security concerns, and execution time limits [4] [5].

Serverless computing is valued for its scalability and cost-effectiveness, transforming application development. However, cold starts—delays when a function is invoked after being idle—remain a critical issue. These delays impact efficiency, performance, cost, and scalability, making it essential to mitigate them to meet the response time requirements of serverless applications [6]. The dynamic resource scaling in serverless architectures lowers costs and lets developers focus on code implementation [7]. However, cold starts introduce delays that can degrade user experience, as the first call triggers function execution in a new environment, impacting response times.

Since its inception, serverless computing has grown rapidly; however, production applications still struggle with cold start delays, affecting real-world performance and often requiring overprovisioning to minimize these delays [8]. Improving cold start mitigation techniques is essential for maximizing the potential of serverless computing. Scalability in serverless computing involves dynamic resource allocation, which can cause cold starts, creating a trade-off between responsiveness and resource efficiency. Quantifying and understanding this trade-off is vital for optimizing serverless architectures.

Innovative solutions for cold starts include container preheating to reduce initial delays and optimization of function startup to enhance performance. Optimizing language interpreter startup and managing package dependencies are key strategies, with lightweight containers speeding up interpreter load times. Efficient dependency management ensures fast function operation, reducing cold start times. Although these methods improve serverless performance, they also present challenges. For example, container preheating increases resource overhead, and optimizing resource management requires accurate predictions. Optimizing the function startup process requires detailed knowledge of runtime characteristics and the underlying isolation technique, such as container-based or microVM-based isolation mechanisms.

This work aims to address cold start issues in serverless computing, ensuring it remains a practical, scalable option for diverse applications. The research focuses on overcoming latency during function initialization in idle containers, preserv-

This paper was produced by the IEEE Publication Technology Group. They are in Piscataway, NJ.

Manuscript received April 19, 2021; revised August 16, 2021.

0000–0000/00$00.00 © 2021 IEEE

ing the advantages of serverless architectures like scalability and low maintenance.

In this work, we propose PARSEC, a serverless platform that mitigates cold start latency through a novel intra-function reuse mechanism based on specialized Zygote containers. Unlike existing approaches, PARSEC avoids inter-function sharing and instead builds dynamic Zygote trees to enable fast and secure container cloning. By combining lean containers, multi-level caching, and adaptive provisioning, PARSEC significantly reduces initialization times while preserving isolation and scalability. Our results show that PARSEC achieves superior performance compared to state-of-the-art solutions, particularly in scenarios with frequent short-lived invocations.

II. Background

A. Motivation and Research Gap

Serverless computing platforms have introduced mechanisms to reduce cold start latency, including container prewarming [9], [10], package caching [11]–[13], predictive invocation [14]–[16], and sandboxing optimizations [17], [18]. However, these approaches often face trade-offs between memory overhead, orchestration complexity, and runtime isolation.

An interesting solution is Pagurus [19], [20], which leverages Zygote-based container reuse across functions. Although effective at reducing startup latency, this strategy requires inter-function sharing, encrypted repackaging, and privilege management. This increases system complexity and introduces potential security vulnerabilities. More recent works propose alternate approaches. FlashCube [21] reduces startup time via streamlined container runtimes, while Rainbowcake [22] implements layer-wise caching and reuse across container boundaries. However, both rely on the concept of shared container state. Similarly, AsyFunc [23] and Opportunistic Pre-Loading [24] target inference-specific workloads using asymmetric function design and anticipatory loading. The proposal called Catalyzer [25] achieves low startup latency using checkpoint-based booting, but depends on a tightly integrated execution model. These methods demonstrate strong advances but are either domain-specific, rely on shared runtime artifacts, or require non-trivial runtime customization. This leaves an important gap: the need for a scalable and general-purpose solution that avoids inter-function sharing while maintaining predictable performance and secure isolation.

PARSEC addresses this gap through a novel intra-function reuse strategy based on dynamically generated Zygote trees. Each function is served by a dedicated set of pre-initialized containers tailored to its own dependencies, eliminating the need for cross-function reuse. Combined with a multi-tier caching mechanism for both install- and import-time overhead, PARSEC reduces cold start latency while maintaining simplicity, scalability, and strong isolation guarantees.

B. The Cold Start Problem in Serverless Computing

Function-as-a-Service (FaaS) enables developers to focus on application logic while offloading operational concerns to cloud providers. Its scalability and cost-effectiveness have made it widely adopted by major platforms such as AWS Lambda, Azure Functions, and Google Cloud Functions. However, a key performance bottleneck in FaaS is the cold start problem, which occurs when a function is invoked after a period of inactivity and must be initialized from scratch. These cold starts introduce significant delays due to the overhead of container startup, runtime initialization, dependency loading, and network configuration.

Various mitigation strategies have been explored to address this issue. These strategies include: 1) container prewarming, where functions are kept alive in memory to serve future requests, 2) package caching, which reduces import overhead by storing common libraries in memory or on disk, 3) predictive provisioning using machine learning to anticipate invocations, and 4) Zygote-based approaches that fork from pre-initialized interpreter states to avoid full runtime startup. While effective in certain contexts, these methods each introduce trade-offs in terms of memory consumption, orchestration complexity, or security risk.

Zygote-based systems like Pagurus [19], [20] attempt to reuse containers across functions to mitigate startup latency. However, this introduces privacy and isolation concerns due to inter-function sharing, requiring complex privilege domains and sophisticated dependency resolution.

Beyond open-source platforms such as Apache OpenWhisk, several studies have characterized cold start behaviors in commercial FaaS providers. For example, AWS Lambda exhibits startup latencies that vary based on the runtime language and memory allocation, ranging from a few hundred milliseconds to over a second in some cases [26]. Azure Functions and Google Cloud Functions show similar variability, with latency influenced by factors such as function size, concurrency levels, and autoscaling events. These findings reinforce that cold start delays are not limited to academic platforms but are a widespread and unresolved issue across the FaaS ecosystem. This motivates the need for generalizable, platform-agnostic solutions such as PARSEC.

C. Existing Mitigation Strategies

Mitigating cold starts has been the focus of a wide range of research, with approaches spanning container management, caching techniques, predictive algorithms, and sandbox optimization. In this section, we summarize key directions in the literature and the basis they provide for the design of PARSEC.

1) Container Prewarming and Pooling: One of the most common strategies is container prewarming, where idle containers are kept alive in memory to reduce the cost of starting new ones. Techniques in this category include preheating function environments [9], as well as maintaining pools of warm containers to serve incoming requests, as stated by authors of [10], who proposed a framework called HotC. It uses adaptive live container control and request prediction to optimize runtime performance. Similarly, the Knative Serving platform addresses cold starts through container pool management [27], while hysteretic policies based on Markov Decision Processes have been proposed to maintain optimal pool sizes [28]. In this same line, FlashCube [21] proposes streamlined

container runtimes for rapid provisioning, and Rainbowcake [22] introduces layer-wise container caching and sharing to minimize reuse overhead. However, both approaches depend on various forms of cross-function or layered reuse, requiring coordination that can impact isolation and flexibility.

2) Package and Function Caching: To avoid redundant setup during function startup, cache-based approaches attempt to retain and reuse imported libraries or installed packages. In [11], Faa$T introduces a distributed cache that unloads after each invocation but is pre-warmed with expected future objects. FaasCache [12] draws parallels between function lifecycles and object caching to enhance reuse. Additionally, SOCK [13] demonstrates the benefits of refining Linux container primitives to improve caching and provisioning.

Complementary techniques like Freshen [29] allow pre-execution steps to populate caches or trigger relevant setups before function invocation. These approaches reduce startup costs but still face challenges in dependency tracking and memory overhead when generalized.

3) Predictive Scheduling and Invocation Forecasting: Prediction-based strategies leverage workload patterns to pre-initialize containers before they're needed. Choreography-based approaches, such as the one proposed in [14], use function composition knowledge to trigger early container provisioning. IceBreaker [30] exploits heterogeneous nodes to optimize warm-up under variable cost constraints. Reinforcement learning models are used in [15], [31] to determine when and how long to keep containers warm, while LSTM models anticipate future invocations.

TppFaaS [16] uses Temporal Point Processes to model and forecast invocation probability distributions, improving allocation. AsyFunc [23] introduces an asymmetric handler strategy for inference workloads, optimizing cold path latency by pairing fast and slow function paths. Opportunistic Pre-Loading [24] further builds on this idea by warming not only functions but also relevant data paths, showing performance gains in serverless inference systems.

4) Sandbox and Runtime Optimization: Another line of research focuses on optimizing the function environment itself. SAND [17] enables efficient resource allocation via fine-grained, process-level sandboxing. Firecracker [18], [32] introduces MicroVMs, enabling AWS Lambda and Fargate to launch isolated instances in under 125 milliseconds while maintaining strong boundaries. Faasm [33] leverages WebAssembly and Software Fault Isolation to create lightweight execution contexts, optimizing both memory usage and launch latency. Catalyzer [25] takes a different route by using a checkpoint-based booting process that does not need initialization, achieving sub-millisecond startup times at the cost of increased complexity in system design.

5) Mixed or Hybrid Techniques: Some solutions combine multiple strategies. The Zygote mechanism used in Android systems [34] has been adapted in systems like Pagurus [19], [20], which reuse containers across functions. However, Pagurus requires complex repackaging, conflict resolution, and isolation mechanisms that complicate orchestration and security.

Other hybrid solutions include NUka [35], which identifies image retrieval as a cold start bottleneck, and AWU [36], which applies time series models for warm-up triggers. WLEC [37] and ENSURE [38] blend scheduling, container reuse, and resource management to create more adaptive environments, albeit with varying trade-offs.

D. Comparative Analysis of Key Solutions

Several recent solutions offer meaningful contributions to the cold start problem. Among Zygote-based systems, Pagurus [19], [20] is most closely related to our work. Pagurus reduces latency through container reuse across functions, but its design introduces architectural complexities, including encrypted repackaging, cross-function scheduling, and package conflict resolution. These mechanisms increase orchestration overhead and raise isolation concerns, particularly in multi-tenant environments.

In contrast, PARSEC avoids inter-function sharing altogether. Each function is served by a dedicated tree of pre-initialized Zygotes containing only the dependencies it requires. This intra-function reuse model eliminates privilege management domains and simplifies container lifecycle operations while preserving cold start benefits.

FlashCube [21] and Rainbowcake [22] pursue a low-latency startup through efficient container runtime design and layer-wise caching, respectively. However, both approaches still depend on shared container layers across functions, introducing coupling and coordination costs. FlashCube is not so efficient, especially when there is an opportunity to improve package and dependency handling, where PARSEC shines. In the case of Rainbowcare, their proposal has a weaker isolation because there is coupling across functions, which does not favor scaling and coordination. AsyFunc [23] and Opportunistic Pre-Loading [24] improve inference-specific startup performance using paired functions or anticipatory data loading, yet remain domain-focused and lack general-purpose applicability. In the case of AsyFunc, it is focused on machine learning models, thus being domain-specific. Catalyzer [25] takes a radical approach by eliminating initialization entirely through checkpoint-based booting. While it achieves sub-millisecond startup times, it depends on specialized runtime infrastructure and snapshot consistency, which can be difficult to generalize or integrate into existing ecosystems.

Compared to these solutions, PARSEC offers a unique balance: general-purpose applicability, lightweight intra-function reuse, and a dynamic, cache-aware container provisioning strategy that avoids the complexity and security risks associated with shared runtimes. Its design builds upon ideas introduced in SOCK [13] and OpenLambda [39], while extending them with deeper Zygote management, fine-grained cache control, and deterministic cold start reduction.

E. Summary and Design Keys for PARSEC

The landscape of cold start mitigation in serverless computing is broad and evolving. Existing solutions have approached the problem from various angles—container reuse, caching, predictive scheduling, and runtime optimization—but each introduces trade-offs in system complexity, memory overhead,

or isolation guarantees. In particular, solutions relying on inter-function sharing, such as Pagurus [19], [20], offer performance benefits but come at the cost of security, orchestration complexity, and reduced modularity.

Techniques like FlashCube [21] and Rainbowcake [22] reduce startup latency through container-level or layer-wise optimizations but still depend on shared infrastructure across functions. Others, including AsyFunc [23], Opportunistic Pre-Loading [24], and Catalyzer [25], focus on specialized use cases such as inference or checkpoint-based initialization, which may not generalize across diverse FaaS workloads.

These insights inform the design of PARSEC. Our platform adopts an intra-function reuse model, in which each function is served by a dedicated pool of Zygotes—pre-initialized with only the packages that function requires. This avoids cross-function sharing, simplifying isolation while preserving startup gains. The Zygote trees are dynamically constructed and updated based on invocation patterns, enabling adaptation to evolving workloads.

Zygotes in PARSEC are not created for all possible combinations of packages. Instead, they are specialized for combinations that are frequently used. For this reason, package usage patterns are monitored, and those that most often appear together are grouped into candidate sets. Preloading all possible combinations would be inefficient. A threshold must be established: if a package or a combination of packages is imported with a frequency that exceeds this threshold, it is then considered a candidate to be preloaded as a Zygote.

In addition, PARSEC integrates a multi-tier caching architecture that targets both install-time and import-time overhead. Combined with lean container management inspired by SOCK [13] and efficient sandbox provisioning, this design provides a balanced solution that reduces latency, simplifies orchestration, and maintains secure, function-specific boundaries.

Taken together, these choices position PARSEC as a general-purpose, efficient, and secure solution to cold start mitigation—one that addresses the limitations of prior work while maintaining the modularity and scalability expected from modern serverless platforms.

III. The SOCK and OpenLambda architecture

OpenLambda, an open-source platform for building web services, was presented in [40], and its architecture is shown in Figure 1. This paper builds on OpenLambda's new approaches to address the cold start issue. The OpenLambda architecture provides sandboxing mechanisms for isolation, with the specialized SOCK container engine replacing Docker to facilitate optimizations in sandbox provisioning, which are listed here:

- Omitting network namespaces, enhancing scalability.
- Efficient file system management, utilizing bind mounts.
- Pooling and reusing pre-initialized cgroups, which improves resource utilization.

SOCK employs simple read-only bind mounts for container file systems, enhancing containerization speed and reducing queuing delays before function logic execution. Beyond Docker replacement, SOCK adoption enables optimizations in other critical components of the OpenLambda workflow:

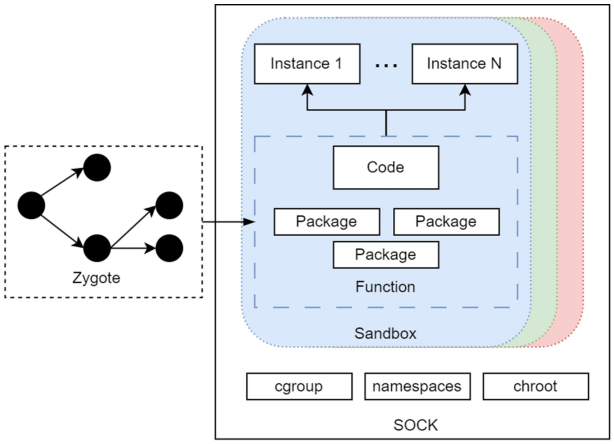


Fig. 1. OpenLambda base architecture

- Language Interpreter Bootstrapping: Streamlines the process to minimize startup delays for functions in diverse programming languages.
- Package Dependencies Installation: Emphasizes optimization of the installation process.
- Function Loading and Linking: Expedites loading and linking, minimizing the time between invocation and function logic initiation.

The following section presents an improved architecture.

IV. Serverless platform architecture

This section outlines the design and implementation of an optimized serverless platform aimed at reducing cold starts for serverless functions, building upon the OpenLambda platform.

A. Design Challenges and Insights

The design of PARSEC is driven by the following insights and challenges identified in previous sections. First, cold-start latency in our target environment is dominated by repeated dependency installation and import, even when using lean containers. Second, container-level initialization and sandbox setup incur non-trivial overheads that cannot be fully hidden by scaling policies. Third, preserving per-function isolation while reusing initialized state is non-trivial in a multi-tenant setting. Finally, any caching or reuse mechanism must be resource-aware to avoid memory exhaustion in the presence of workload churn. The key insight from the proposed architecture is that a small number of frequently co-imported package sets account for the majority of function invocations. By pre-installing and pre-importing these sets into function-specific Zygote templates, and by reusing these templates via the import cache and handler pool, we can remove most dependency-related initialization from the critical path without violating isolation.

The enhancements focus then on mitigating container launch overheads and minimizing language runtime initialization costs to address delays associated with new instance spin-ups. The platform enhances the balance between isolation and performance by avoiding specific Linux resource namespaces and optimizing cgroups creation. Each of these features is explained in detail now.

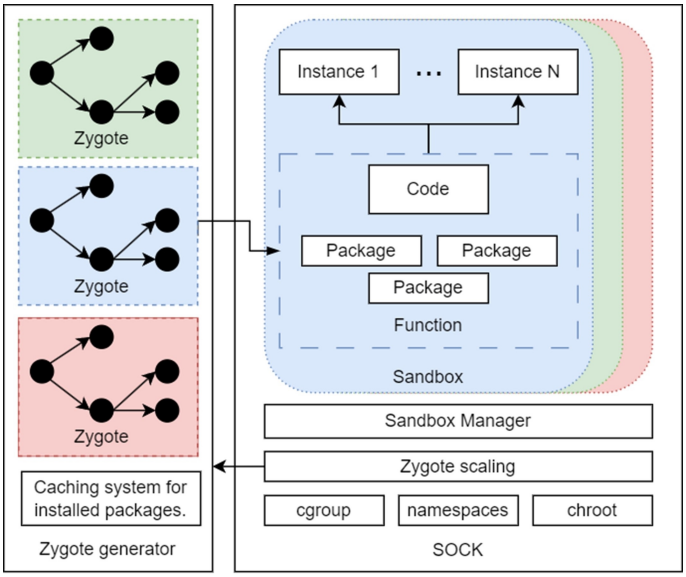


Fig. 2. Overview of the platform architecture

Each design element in Figure 2 directly targets one or more of the above challenges: the cache removes repeated package installation, and combined with Zygote pools, it removes repeated imports. Container optimizations from SOCK reduce remaining provisioning costs.

B. Platform Design

The serverless platform design, depicted in Figure 2, showcases optimizations seamlessly integrated into the invocation workflow. The interaction unfolds as follows:

1) The pool manager checks for an available paused container in the handler cache upon initiation of a new function invocation, loaded with the function's metadata and dependencies.
2) If no suitable cached container is found, the manager consults the Zygote cache, which maintains pools of pre-warmed Python interpreters and selective packages based on recent invocation history.
3) To expedite the launch of a secure sandbox along with language runtime and imports, the manager clones a container from the closest matching Zygote.
4) The cloned helper process handles the loading and linking tasks after fetching the function code, returning to the cache pool.
5) Both the handler and Zygote caches adhere to the Least-Recently-Used (LRU) replacement policies, effectively managing resource consumption.

The platform's optimization strategy targets various phases to achieve reduced end-to-end cold start latencies for Python serverless functions. The meticulous resource pooling and reuse approach aims to maximize steady-state throughput following the initial overhead. As depicted in Figure 2, the optimized platform strategically aims for lower end-to-end cold start latencies, with enhancements spanning specialized containers, interpreter reuse, and caching architectures, collectively contributing to improved efficiency and performance.

Notably, PARSEC differs from prior Zygote-based systems such as Pagurus in several key aspects. While Pagurus aims to reduce cold starts through inter-function container sharing, PARSEC deliberately avoids any cross-function reuse. Instead, it introduces a modular and deterministic Zygote mechanism, in which each function is served by a dedicated tree of pre-initialized interpreter instances based on its specific package dependencies. This eliminates the need for privilege management domains and reduces both the risk of version conflicts and the memory footprint. By leveraging intra-function cloning with copy-on-write semantics and a cache-aware scheduler, PARSEC achieves greater predictability and performance consistency across diverse workloads.

C. Functional Workflow

This section elucidates the end-to-end flow within the serverless platform, as depicted in Figure 3. The functional workflow unfolds as follows:

1) Upon a client's initiation of a function invocation, a request is sent to the gateway. The gateway, dispatcher, and metadata store components collaborate to route the request based on the function name.
2) The dispatcher serves as the initial point of contact, querying metadata associated with the function from a persistent catalog. This metadata encompasses runtime specifications, resource profiles, packages, and other declarative configurations.
3) Before initializing a new instance, the availability of ready sandboxed containers in a handler cache pool is checked. Containers from previous invocations may reside paused in memory, awaiting potential reuse.
4) If no suitable handler exists in the cache, the dispatcher consults the import cache, which tracks interpreter pools with certain packages pre-imported. An appropriate zygote is selected based on the function's package dependencies.
5) The chosen zygote template facilitates the cloning of a new sandboxed container via fork to safely initialize the environment. The child process inherits the zygote's pre-initialized interpreter, pre-imported packages, namespaces, cgroups, and other relevant state.
6) Any additional packages required by the function are loaded within the forked sandbox. Other housekeeping tasks are executed before loading the function handler logic.
7) Finally, the business logic, defining the function implementation, executes upon the event payload. Outputs are then returned to the waiting client.
8) Post-execution, the sandbox may reset or persist in a paused state in the handler cache for subsequent reuse. Garbage collection ensures resource leakage is bounded across activations.

The integrated caching, pooling, and cloning mechanisms collaborate to avoid repeatedly installing expensive tasks across function invocations. Tailored replication limits overheads while accelerating deployments through efficient reuse.

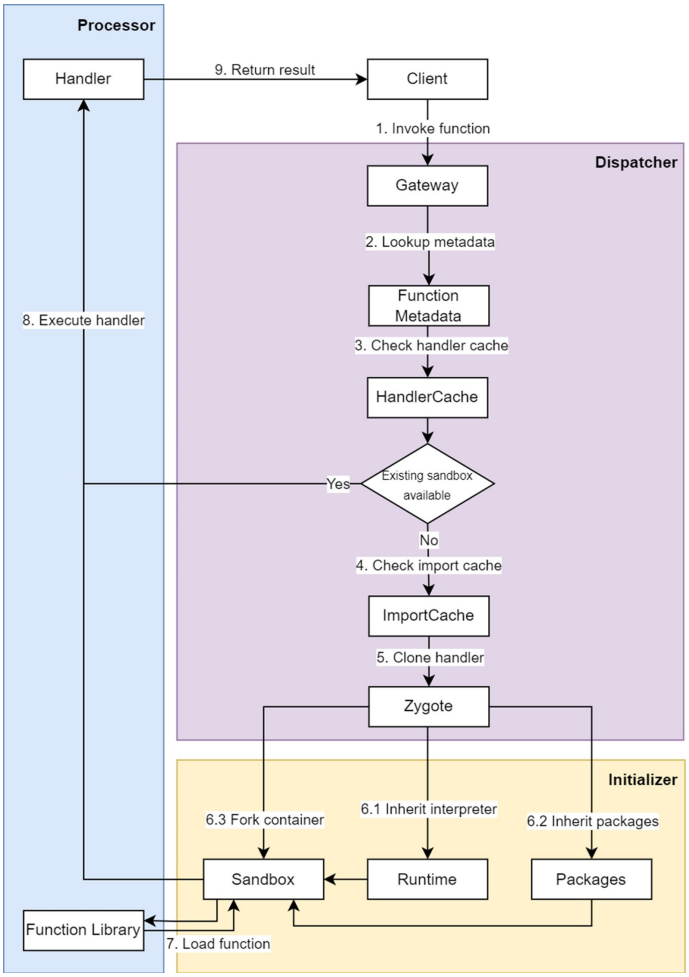


Fig. 3. Platform workflow

D. Lean Containers

To minimize container launch latency and resource overhead, our platform avoids unnecessary Linux kernel features commonly used for general-purpose containerization. These features, while flexible, are not essential for FaaS workloads and may hinder scalability. The SOCK container engine, developed in earlier work [13], serves as the foundational isolation layer. SOCK strategically avoids several expensive and unnecessary Linux primitives, which, while offering flexibility, are not essential in serverless contexts. Network namespaces introduce a global lock, serializing access to shared kernel data structures, limiting scalability [41]. However, in serverless platforms where functions are typically fronted by a NAT gateway, network isolation becomes largely superfluous. By avoiding network namespaces, a significant throughput bottleneck is eliminated. Secondly, bind mounts emerge as a lighter-weight alternative to union file systems for assembling container root directories. The proposed platform leverages bind mounts to multiplex read-only access to common libraries across functions, simultaneously allowing for per-function scratch space for writes. This approach minimizes overhead by cross-mounting existing paths instead of maintaining layered copies.

E. Interpreter Reuse

Language runtime initialization and package imports account for a large portion of cold start latency. To address this, we adopt the Zygote process model—successfully used in other domains like Android OS—to enable fast interpreter cloning and avoid repeated imports across function invocations. To reduce the initialization costs of interpreters and libraries across function instances, the system adopts and generalizes the Zygote process model from Android systems. Instead of initializing interpreters from scratch, new Python runtimes are forked from pre-warmed Zygote processes that have loaded specific packages beforehand. Child processes inherit the parent's initialized executable memory, thanks to copy-on-write semantics [34]. The system maintains pools of Zygotes with varying application packages imported based on workload patterns, enabling containers to efficiently clone from the Zygote containing those imports. Policies govern parent selection and eviction, employing a tree representation for Zygotes. The Lambda model implemented in the system enables interpreter reuse.

F. Caching Architecture

Installing and importing dependencies is a recurring cost in serverless platforms. Most existing systems either duplicate package installations or cache indiscriminately, leading to memory waste or contention. To mitigate this, we design a multi-tier caching strategy to target both install-time and import-time overheads.

While Zygotes contribute to the acceleration of launching new sandboxes by caching interpreter state, two additional caching tiers specifically target package management overheads associated with cold starts. Firstly, an install cache contains a substantial subset of the Python Package Index (PyPI) repository pre-installed into a shared read-only partition. Containers mount this partition to avoid the need for repeatedly installing the same packages. Analysis reveals that over 97% of PyPI packages can co-install safely without conflicts that would require isolation [13].

Secondly, the Zygote pool acts as an import cache by selectively pre-importing hot packages likely to appear in future invocations. The import cache monitors package usage patterns by tracking the frequency and co-occurrence of imported packages across recent function invocations. A sliding window over a configurable number of recent executions is used to maintain statistics. Packages that appear frequently together are grouped into candidate sets for new Zygote templates. The system uses a simple frequency threshold and co-import correlation to decide which combinations merit preloading. This heuristic allows the platform to dynamically adapt Zygote provisioning to workload patterns. In the case that the fork is not necessary, LRU is implemented. If the idle time of a Zygote child has not expired, and if it is possible to reuse it, then it will be used. Cache hits result in the cloning of an existing Zygote, thereby skipping imports. Pooling Zygotes based on package sets avoids contention on single templates. Least Recently Used (LRU) eviction ensures resource efficiency and bounds resource stranding.

Zygotes are not created for all possible package combinations; instead, they are specialized for combinations that are frequently used. To this end, we monitor package usage patterns and group packages that most often appear together into candidate sets. Preloading every possible combination would be inefficient. We therefore define a promotion threshold: if a package—or a combination of packages—is imported with a frequency that exceeds this threshold, it is considered a candidate to be preloaded as a Zygote. With this approach, frequent packages are loaded, thus removing the necessity to load all possible combinations, because the majority of them would not be necessary.

Together, the platform strategically caches packages at multiple levels—as on-disk installs, resident memory in interpreters, and paused container states. This coordinated reuse minimizes duplication across various initialization routes, providing a balanced approach to eviction and replication to manage concurrency and resource consumption.

G. Zygote Mechanics

In addition to minimizing cold starts, the system must efficiently manage growing package diversity across serverless functions. We introduce a tree-structured Zygote pool to maximize reuse while avoiding package conflicts and redundant memory usage.

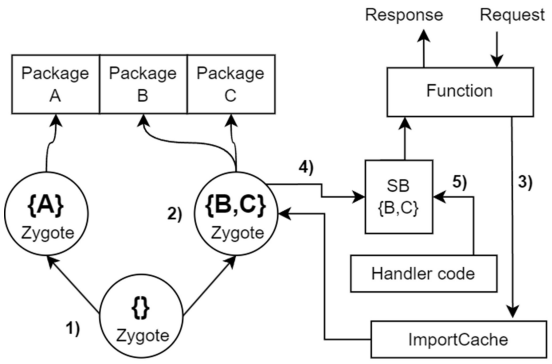


Fig. 4. Zygote Workflow

Zygotes are pre-initialized interpreter sandboxes that serve as templates for cloning new handler containers on-demand. By pre-loading common packages in advance, zygotes enable child processes to inherit ready-to-execute application contexts, avoiding redundant initialization. The mechanics of Zygotes unfold as follows (refer to Figure 4):

1) Base zygotes are spawned with only the Python runtime and no imports. The platform pre-provisions certain zygotes by importing select packages.
2) Zygotes can fork specialized child zygotes that pre-import additional packages, creating a tree structure of templates.
3) When a function is invoked and requires specific packages, its handler container clones from the zygote highest up the tree that has those imports pre-loaded.
4) The child container process inherits the parent zygote's interpreter, packages, cgroups, namespaces, and other states via copy-on-write, avoiding redundant costs.
5) Further customization, such as loading the handler code, occurs in the cloned sandbox before handling the request.

In this way, zygotes act as forkable caches of initialized runtimes, allowing function containers to layer over them. This amortizes substantial initialization costs through reuse. Tailored templates play a crucial role in accelerating various cold start paths.

H. Zygote Selection Process

When running multiple concurrent containers on the same host, resource contention can degrade performance. Existing isolation methods introduce overhead when applied per-container at runtime. We implement a lightweight cgroup pooling strategy to preconfigure and reuse resource boundaries efficiently.

The selection of an appropriate Zygote template is a critical aspect of the serverless platform, ensuring optimal performance while mitigating potential security risks. In the context illustrated in Figure 5, consider a scenario in which a handler is invoked, requiring the presence of packages A and B.

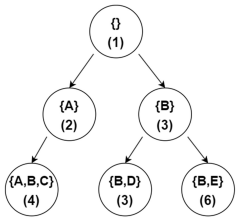


Fig. 5. Tree package cache [13]

Entry 4 appears optimal for a new interpreter due to pre-imported packages. However, using it poses a security risk if it includes potentially malicious packages, such as package C. To mitigate this, it is recommended not to use cache entries with pre-imported packages not explicitly requested by a handler. Thus, Entries 2 and 3 are better candidates for Zygote selection. The import cache selects the entry with the most matching packages, and breaks ties randomly. If no exact match exists, the platform replicates Entry X to create Entry Y, imports the remaining packages into Y, and then provisions the handler from Y.

This method ensures Zygote selection aligns with handler package requirements, balancing performance and security. The random tie-breaking mechanism enhances security by minimizing exposure to unvetted code during Zygote selection [13].

V. Optimization Methods

This work introduces an adaptive serverless platform extending OpenLambda to minimize cold start times through targeted enhancements in container launch and Python environment initialization. These optimizations will be evaluated using a comprehensive test suite to measure cold start performance under various workloads. The aim is to demonstrate significantly faster and more consistent cold starts compared

to current open-source platforms. These adaptive techniques could expand the applicability of serverless computing to latency-critical services such as serverless databases, IoT applications, and reactive systems.

A. Container Initialization Optimization

The optimization of container initialization focuses on streamlining and enhancing the process for launching the container proxy. The new implementation introduces a cleaner interface, reducing redundancy and improving code readability through concise error handling and specific error messages. A dedicated function for reading the proxy PID centralizes logic and promotes code modularity.

A notable feature is sandbox checkpoint restoration, which saves snapshots of container states for quick restoration. This allows faster recovery compared to fresh launches, especially when reusing the same configuration. Checkpointed images can be split into child instances via fork, offering rapid scaling with minimal resource overcommitment. Sibling forks use copy-on-write memory, efficiently sharing read-only cache state.

Additionally, a timer mechanism for periodic checks of the proxy's readiness replaces a lengthier loop structure, providing a more efficient and responsive method for verifying proxy availability. This enhances the responsiveness of the initialization process, contributing to overall optimization.

B. Zygote Management Enhancements

To optimize package management costs, the platform incorporates two additional caching layers: an install cache and an import cache. The install cache includes a substantial portion of the Python Package Index (PyPI) pre-installed in a read-only file system partition shared across containers. This avoids the need to repeatedly install popular packages in each environment. Containers mount this partition at runtime, providing immediate access to required packages.

The import cache acts as a reservoir of pre-initialized interpreter instances, referred to as Zygotes, with frequently used packages already imported. The platform monitors package usage trends to inform Zygote provisioning. Forking a container from an existing Zygote with relevant imports avoids redundant library loading.

This multi-tier caching architecture mitigates package management overheads by streamlining both installation and import processes. Strict containment limits and eviction mechanisms protect against resource exhaustion attacks, ensuring robust security. These enhancements reduce the overhead associated with library imports and code loading, balancing currency and resource usage.

1) Zygote Scaling: The newly implemented Zygote scaling enhances the system's package management capabilities by dynamically managing the creation and cleanup of Zygotes based on system requirements. This Zygote scaling provider includes parameters for flexibility and adaptability, such as specifying the maximum and minimum number of Zygotes, an idle Zygote timeout, and a pool of sandboxes. The provider intelligently selects Zygotes for sandbox creation based on the current workload.

A key feature is its cleanup mechanism, which identifies and removes Zygotes eligible for cleanup based on specific criteria. This approach includes maintaining multiple independent Zygote trees, ensuring that concurrent requests can choose from this group without delays. This dynamic selection process identifies the Zygote with the least workload, preventing contention for a single Zygote and addressing potential bottlenecks associated with concurrent calls.

2) Zygote Generator: The introduction of the Zygote generator represents a significant advancement in the system's package management capabilities, addressing a previously absent feature. This enhancement introduces a dynamic mechanism to generate Zygote trees based on specified package dependencies. The Zygote generator operates through three main functions (refer to Figure 6):

1) **Node Generation**: Creates structured nodes representing packages and their dependencies.
2) **Tree Building**: Organizes package dependencies into a hierarchical tree structure using a dependency map, accurately identifying root dependencies.
3) **JSON Configuration**: Converts the generated tree into a structured JSON configuration, clearly representing the Zygote tree.

Additionally, the generator reads package requirements from a specified file, efficiently filtering out unnecessary information. The execution section demonstrates the practical application of these functions, dynamically generating Zygote trees based on specified dependencies. Collectively, the Zygote generator significantly enhances the system's package management capabilities, providing a dynamic and automated mechanism for creating Zygote trees [13].

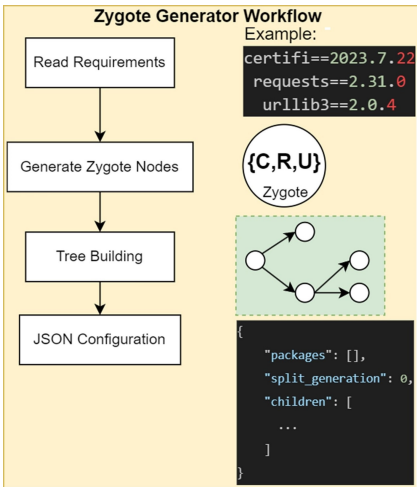


Fig. 6. Zygote generator workflow

C. Cache system improvement

The enhancement in the cache system introduces a structured mechanism for storing successfully installed packages. In the absence of such a system initially, the improved approach

adopts the singleton pattern to ensure a single and thread-safe instance of the cache. This streamlined implementation simplifies the process of adding packages to the cache and retrieving them, promoting code modularity. A pivotal aspect of this improvement is the introduction of a method for efficiently clearing the cache, providing a controlled and organized approach to cache management. The inclusion of a method to determine the number of packages in the cache offers valuable insights into cache usage. Moreover, the implementation introduces configurable flags to dynamically control the behavior of the caching system. These flags allow adaptability based on system requirements, providing a flexible approach to prioritize local resources and enabling caching. The forward-looking feature of fetching packages from external sources, if not available locally, adds an additional layer of resilience to the system.

D. Resource Contention Isolation

While specialized containers and caching accelerate the deployment of individual serverless functions, the potential for resource contention arises when multiple functions execute concurrently on the same host. Linux processes are isolated with a combination of cgroup (for performance isolation) and namespace primitives (for logical isolation). To mitigate interference, the platform enhances performance isolation using Linux control groups (cgroups). As discussed before, configuring new cgroups incurs non-trivial overheads. Thus, the proposed approach involves pooling pre-initialized cgroups that sandboxes can be assigned to at runtime. Specifically, a cluster-wide coordinator tracks available cgroups on each host that enforce preset quotas on CPU, memory, IO, and other resources. When launching new containers, the manager selects the cgroup that best fits the function's declared resource profile, embedding containers within assigned limits for the duration of execution.

Moreover, the platform explores dynamically adjusting cgroup parameters and process-to-cgroup mappings based on monitored contention and changing requirements. This ability to smoothly redistribute spare capacity or restrict interference promotes performance stability amidst load variability. Together with the other improvements, robust resource isolation allows high steady-state throughput, even with language runtimes and packages cached in memory.

The refined cgroup configuration and management, marked by systematic retry mechanisms and improved error handling during cgroup assignment and destruction, significantly contribute to heightened performance isolation within the serverless platform. The initial lack of a structured retry mechanism could potentially lead to inconsistencies in cgroup removal, impacting the reliability of resource contention isolation. The introduction of a controlled retry strategy, coupled with an adjustable limit and exponential backoff, ensures a more robust cgroup destruction process.

VI. Case Study

A. Scenario Description

1) Dataset Overview: Our case study utilizes the Azure Functions Dataset, covering function invocations across Azure's infrastructure from July 15 to July 28, 2019. This dataset offers a comprehensive view of serverless computing workloads, reflecting their diversity and complexity [42].

The Azure Functions Dataset is significant due to its broad scope, ensuring our study's relevance to various serverless scenarios. It captures diverse workloads, mirroring the dynamic nature of serverless applications in real-world cloud environments. This dataset is particularly relevant to our objective of reducing cold start times in serverless platforms, providing insights into enterprise serverless deployments.

2) Dataset Description: Sourced from Microsoft's Azure Functions service, the Azure Functions Trace 2019 dataset offers a portion of the workload gathered in July 2019 [42]. It includes a random sample of Applications, each containing various functions. The dataset consists of files detailing function invocations, triggers, execution times, and memory allocation patterns.

3) Dataset Processing: The dataset underwent processing to facilitate experimentation. Rows were organized based on days, with each row representing function invocations for a specific minute. This restructuring increased the dataset size significantly. From this processed dataset, functions with diverse behaviors were selected for analysis [42].

Table I provides a statistical summary of each function's invocations, guiding our analysis of serverless performance and cold start optimization. These steps ensure our case study's robustness and relevance to real-world serverless scenarios.

TABLE I
SUMMARY OF DATASET STATISTICS FOR EACH FUNCTION

Func.	Total	Avg. /Min	Std. Dev.	Min	Max	P25	P50	P75
1	2887	2.004	0.629	0	6	2.0	2.0	2.0
2	1446	1.004	0.533	0	4	1.0	1.0	1.0
3	2256	1.566	0.799	0	3	1.0	2.0	2.0
4	3681	2.556	3.769	0	21	0.0	0.0	4.0
5	805	0.559	0.959	0	5	0.0	0.0	1.0
6	676	0.469	0.995	0	7	0.0	0.0	0.0
7	376	0.261	0.640	0	4	0.0	0.0	0.0
8	141	0.097	0.299	0	2	0.0	0.0	0.0

B. Experimental Setup

For our experimental setup, we employed a Standard_B4ms virtual machine (VM) in the Azure cloud environment. This VM was chosen for its suitability in handling burstable performance workloads, aligning well with our focus on reducing cold start times.

The Standard_B4ms VM provides versatility in hardware compatibility and processor support, ensuring consistent performance across different configurations. Key specifications of this VM are outlined in Table II.

B-series VMs, such as the B4ms, are optimized for fluctuating workloads common in serverless computing scenarios. Their ability to accumulate CPU credits during low-usage periods enables them to deliver enhanced performance when needed, making them ideal for assessing our serverless platform's performance improvements.

TABLE II
TECHNICAL SPECIFICATIONS OF STANDARD_B4MS VM

Property	Value
vCPU	4
Memory: GiB	16
Temp storage (SSD) GiB	32
Base CPU Performance of VM (%)	45
Initial Credits	120
Credits banked/hour	54
Max Banked Credits	1296
Max data disks	8
Max uncached disk throughput: IOPS/MBps	2880/35
Max burst uncached disk throughput: IOPS/MBps	8000/200
Max NICs	4

C. Implementation Details

The experimental execution was methodically structured, employing Python for intricate dataset handling and analysis. The dataset, encompassing detailed records of function executions across an entire day, was meticulously processed using Python. This thorough preparation provided a solid foundation for the subsequent stages of our evaluation. Each function represented in the dataset was subjected to a specific number of invocations per minute, mirroring real-world usage patterns. The following details outline the extended methodology and processes:

1) **Function Setup:** Serverless functions were configured in OpenLambda to match dataset characteristics, including runtime environments, function code, permissions, and dependencies.
2) **Data Processing:** Python scripts parsed the dataset to extract function names, invocation frequencies, and timing, and scheduled function invocations to simulate workload patterns.
3) **Invocation and Monitoring:** Functions were invoked according to the schedule, and response times were recorded to measure execution efficiency.
4) **Metrics Collection:** Key metrics, including cold start latency and function execution time, were collected to assess performance characteristics.
5) **Data Aggregation and Analysis:** Collected data was aggregated and analyzed using statistical methods and data visualization to identify patterns and insights.

By utilizing Python for both dataset processing and orchestration of function invocations, the experiment ensured a robust and replicable evaluation framework. The detailed measurement of response times and concurrent resource utilization monitoring provided a nuanced understanding of the platform's performance under various conditions. This comprehensive approach was instrumental in assessing the effectiveness of the proposed optimizations in real-world serverless computing environments.

VII. RESULTS

A. Impact of Pre-Imported Packages

Our research aimed at optimizing cold start times in serverless architectures by measuring library loading latency using PARSEC. The systematic methodology ensured reliable and valid results. We tested the loading times of five widely-used Python libraries, chosen for their popularity in serverless applications. The primary objective was to compare the loading latency under different methods. Latency, recorded in milliseconds, was the key metric. Three methods were tested:

- **Installed+Imported:** Traditional method involving installation and importation.
- **Zygote:** Preloading method to reduce loading times.
- **Cache:** Utilizing cached data for faster loading.

Each method was tested under identical conditions to ensure accuracy. The latency for each library and method was recorded, resulting in a comprehensive performance dataset.

The data collected indicated:

- **Cache Method:** Consistently showed the best results, significantly reducing loading times for all libraries. This method leverages pre-stored data to expedite the process.
- **Zygote Method:** Displayed considerable latency reduction compared to the Installed+Imported method by preloading libraries, though not as rapidly as the Cache method.
- **Installed+Imported Method:** Resulted in the longest loading times, highlighting the need for innovative approaches in serverless architectures.

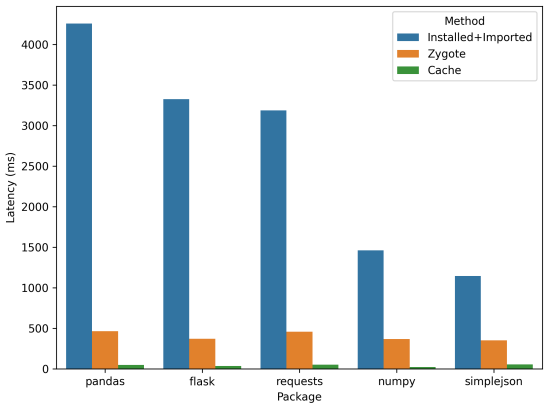


Fig. 7. Comparison of latency by library and method in PARSEC

The results in Figure 7 indicate that both Cache and Zygote methods significantly outperform the traditional Installed+Imported method in reducing library loading times. The Cache method, in particular, stands out as the most effective strategy for minimizing latency, potentially reducing cold start times in serverless computing. These findings strongly recommend adopting these advanced methods to enhance the performance and responsiveness of serverless applications.

B. Comparison of Parsec, Sock, and Docker

To improve serverless architecture performance and reduce cold start times, we compared three sandbox types: PARSEC, SOCK, and Docker. Each sandbox was deployed in a standardized environment and subjected to tests simulating real-world scenarios, focusing on latency as the primary metric. Latency, measured in milliseconds (ms), indicates cold start performance, with lower values signifying faster responses.

Each sandbox was tested with varying levels of concurrent operations to assess scalability and performance under load. Latency was recorded multiple times for accuracy and reliability. This comprehensive approach ensured that performance data was representative and fair. The collected data was analyzed to compare each sandbox's performance and scalability. The focus was on absolute latency and how performance scaled with increasing load, offering insights into the suitability of each sandbox for serverless applications.

The results in Figure 8 showed distinct performance characteristics. PARSEC consistently had the lowest latency across all levels of concurrent operations, indicating it as the most efficient option. As concurrent operations increased, PARSEC maintained a lower increase in response time, suggesting robust architecture for higher loads.

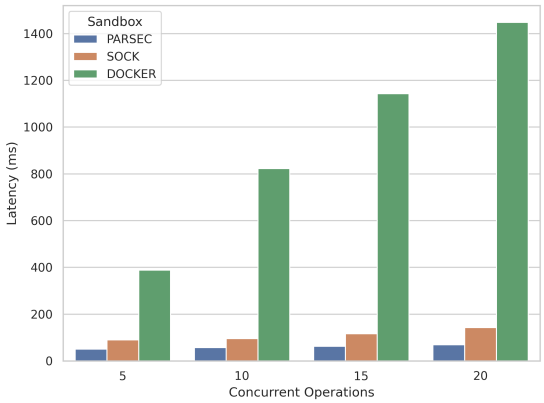


Fig. 8. Comparison of Latency by Sandbox Type as a Function of Concurrent Operations

SOCK and Docker showed higher latency than PARSEC. SOCK performed better than Docker but did not match PARSEC's efficiency. Latency increases were more pronounced for SOCK and Docker under heavier loads, indicating decreased performance.

PARSEC significantly outperformed its counterparts in percentage improvement. The exact figures varied with concurrent operations, but on average, PARSEC improved response times notably compared to SOCK and Docker. This highlights PARSEC's design effectiveness in optimizing for quicker response times and managing concurrent requests efficiently.

C. Container Standby Time

This section compares the cold start times of two serverless platforms, PARSEC and SOCK. The methodology involved measuring response times of containers on each platform under varied wait times. The steps included:

- Sending initial requests to containers on both platforms.
- Allowing containers to remain idle for varying wait times to simulate cold starts.
- Sending subsequent requests post-wait time and recording the elapsed time and status code.

This process was repeated for different wait times to assess the impact of inactivity on each platform's performance.

Figure 9 compares the response times of PARSEC and SOCK as a function of wait time, illustrating each platform's cold start behavior and efficiency. The data indicates that PARSEC generally offers better performance, particularly at shorter wait times, suggesting effective cold start management. Both platforms show increased response times with longer waits, but PARSEC consistently performs better.

Quantitatively, PARSEC demonstrates variable improvement over SOCK, with a more significant advantage at shorter wait times. Although this margin decreases as wait time increases, PARSEC remains more efficient throughout the tested range.

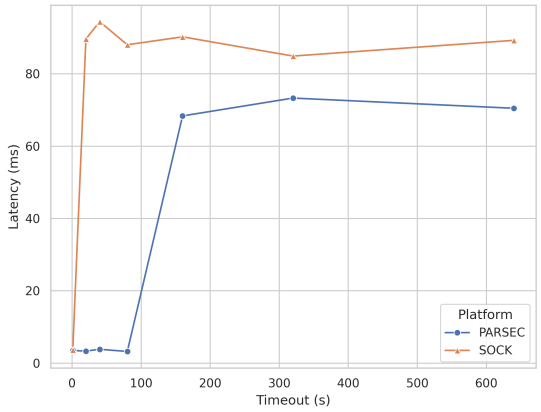


Fig. 9. Comparison of Response Time vs. Waiting Time between PARSEC and SOCK

D. Cold Start Analysis with Azure Dataset

This analysis evaluates cold start times using the Azure Functions Dataset to compare the performance of SOCK and PARSEC environments. The objective is to assess response time and cold start frequency accurately.

The environment setup is:

- SOCK Environment: Conventional serverless architecture baseline.
- PARSEC Environment: Enhanced SOCK version with optimizations to reduce cold starts and improve response times.

Eight distinct functions, representing typical serverless applications, were deployed in both environments with identical codebases. A script simulated invocations over a set period, ensuring equal and realistic usage patterns.

For each invocation, the following data was collected:

- Minute of Invocation: To calculate invocations per minute.
- Response Time (ElapsedTime): Recorded in milliseconds.
- Status Code: To ensure successful execution.
- Test Identifier: Indicating SOCK or PARSEC environment.

To illustrate the results, we present the analysis of Function 1 as an example. Figure 10 compares response times for SOCK and PARSEC environments, focusing on latency patterns throughout the day.

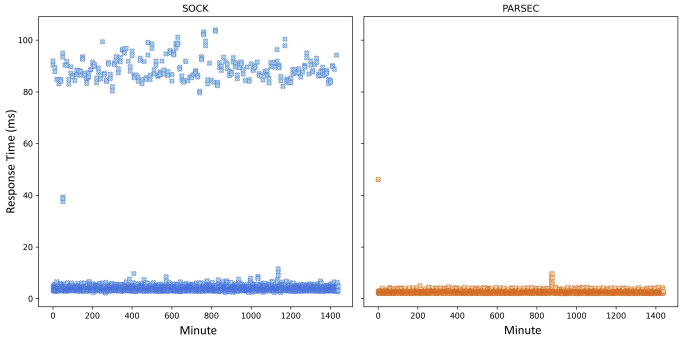


Fig. 10. Response Time Analysis for Function 1 with Azure Function Dataset

Function 1, invoked 2887 times in both environments, showed significant performance differences. In SOCK, the average response time was 12.42 ms, with 283 cold starts (response time greater than 30 ms). In PARSEC, the average response time dropped to 2.67 ms, with only 2 cold starts. This demonstrates PARSEC's superior optimization in reducing both response times and cold starts, highlighting its effectiveness in enhancing serverless architecture performance.

E. Discussion

Table III compares SOCK and PARSEC, focusing on average response times and cold starts. The data illustrates PARSEC's superior performance, with shorter response times and fewer cold starts across all functions, highlighting PARSEC's optimization in latency-sensitive areas. PARSEC's fewer cold starts indicate significant advancements in serverless computing, enhancing user experience and system reliability. These improvements are crucial for real-world applications where performance consistency is key. The empirical evidence supports PARSEC's architectural improvements. Compared to SOCK, PARSEC excels in reducing cold start times and improving overall performance.

TABLE III
SUMMARY OF DATASET STATISTICS FOR EACH FUNCTION

Func.	**SOCK**		**PARSEC**	
	Avg. Response Time (ms)	Cold Starts	Avg. Response Time (ms)	Cold
1	12.42	283	2.67	2
2	13.18	162	3.78	27
3	12.71	232	2.79	6
4	16.54	466	8.81	212
5	17.20	125	5.21	32
6	21.35	136	10.90	85
7	22.94	84	9.48	43
8	29.15	42	12.37	26

Table IV details the performance enhancements achieved by PARSEC, showing percentage improvements in response time and reductions in cold start occurrences for each function. This data provides concrete evidence of PARSEC's effectiveness, illustrating significant improvements in both response efficiency and cold start mitigation. These enhancements impact not only technical aspects but also user experience and operational efficiency. PARSEC's reductions in response times and cold starts reflect a shift towards more efficient, reliable, and responsive serverless computing systems.

TABLE IV
PERFORMANCE ENHANCEMENTS PER FUNCTION IN DURATION AND COLD STARTS WITH PARSEC

Function	Duration Improvement	Cold Start Improvement
1	78.51%	99.29%
2	71.35%	83.33%
3	78.07%	97.41%
4	46.76%	54.51%
5	69.73%	74.40%
6	48.94%	37.50%
7	58.65%	48.81%
8	57.55%	38.10%

VIII. CONCLUSION

This work focused on addressing cold start challenges by optimizing container initialization through tailored processes, efficient restoration of sandbox checkpoints, and aligning container setup with serverless requirements. The proposed architecture achieves cold start improvement by streamlining the initialization of containers to reduce overhead. This involves minimizing unnecessary operations and customizing launches for serverless needs, aiming for a faster and more efficient setup. It also enhances the provisioning of Zygotes to speed up sandbox launches. By dynamically managing pre-warmed Python interpreters and selectively using packages based on recent usage, this approach seeks to improve the responsiveness of the serverless platform. This proposal enhances the cache system by advancing a multi-tier cache system to reduce package installation and import times. By fine-tuning the caching strategy, the system aims to import only necessary packages, reducing unnecessary workload and improving efficiency.

Finally, we investigated methods to enhance performance isolation using cgroups. This involves exploring efficient ways to reassign processes to different cgroups, improving performance isolation in serverless environments. The strategic management of Zygotes and their provisioning scaling plays a critical role in managing large numbers of packages and instances, thereby enhancing the performance of package management. The cache system evolves to become more selective, reducing overhead by focusing on essential packages. Enhancements in performance isolation delve into cgroups, aiming to manage resources better and improve isolation. Alongside, strategies for managing resource contention are optimized, including intelligent retry limits and backoff strategies, to enhance resource cleanup. Together, these enhancements contribute to the serverless computing field by addressing key challenges and improving adaptability, responsiveness, and efficiency. This proposal marks a significant step forward, offering a refined approach to serverless computing challenges.

REFERENCES

[1] E. Ahumada-Tello and R. Evans, "Survey on serverless computing," *Journal of Cloud Computing*, vol. 10, no. 1, p. 39.
[2] E. Ahumada-Tello and R. Evans, "A complexity-based framework for social product development," *Procedia CIRP*, vol. 119, pp. 1204–1209, 2023.

[3] S. G. Kulkarni, G. Liu, K. K. Ramakrishnan, and T. Wood, "Living on the edge: Serverless computing and the cost of failure resiliency," in *2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pp. 1–6. ISSN: 1944-0375.

[4] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Computing Surveys*, vol. 54, no. 11, pp. 239:1–239:32.

[5] P. Sanmartin, K. Avila, S. Valle, J. Gomez, and D. Jabba, "Sbr: A novel architecture of software defined network using the rpl protocol for internet of things," *IEEE Access*, vol. 9, pp. 119977–119986, 2021.

[6] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless applications: Why, when, and how?," *IEEE Software*, vol. 38, no. 1, pp. 32–39. Conference Name: IEEE Software.

[7] D. Bardsley, L. Ryan, and J. Howard, "Serverless performance and optimization strategies," in *2018 IEEE International Conference on Smart Cloud (SmartCloud)*, pp. 19–26.

[8] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," pp. 133–146.

[9] P. Silva, D. Fireman, and T. E. Pereira, "Prebaking functions to warm the serverless cold start," in *Proceedings of the 21st International Middleware Conference*, Middleware '20, pp. 1–13, Association for Computing Machinery.

[10] K. Suo, J. Son, D. Cheng, W. Chen, and S. Baidya, "Tackling cold start of serverless applications by efficient and adaptive container runtime reusing," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 433–443. ISSN: 2168-9253.

[11] F. Romero, G. I. Chaudhry, Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$t: A transparent auto-scaling cache for serverless applications," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pp. 122–137, Association for Computing Machinery.

[12] A. Fuerst and P. Sharma, "FaasCache: keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pp. 386–400, Association for Computing Machinery.

[13] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," pp. 57–70.

[14] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using application knowledge to reduce cold starts in FaaS services," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, pp. 134–143, Association for Computing Machinery.

[15] P. Vahidinia, B. Farahani, and F. S. Aliee, "Mitigating cold start problem in serverless computing: A reinforcement learning approach," *IEEE Internet of Things Journal*, pp. 1–1. Conference Name: IEEE Internet of Things Journal.

[16] M. Steinbach, A. Jindal, M. Chadha, M. Gerndt, and S. Benedict, "TppFaaS: Modeling serverless functions invocations via temporal point processes," *IEEE Access*, vol. 10, pp. 9059–9084. Conference Name: IEEE Access.

[17] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards high-performance serverless computing," pp. 923–935.

[18] K. Mahajan, S. Mahajan, V. Misra, and D. Rubenstein, "Exploiting content similarity to address cold start in container deployments," in *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT '19 Companion, pp. 37–39, Association for Computing Machinery.

[19] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li, and M. Guo, "Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 69–84, USENIX Association.

[20] Z. Li, Q. Chen, and M. Guo, "Pagurus: Eliminating Cold Startup in Serverless Computing with Inter-Action Container Sharing," Aug. 2021. arXiv:2108.11240 [cs].

[21] A. Nayak, S. Ramaswamy, H. Kasture, D. Narayanan, and M. Sivathanu, "Flashcube: Fast provisioning of serverless functions with streamlined container runtimes," in *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC)*, pp. 135–149, 2022.

[22] C. Yu, J. Kang, J. Yoon, and B.-G. Lee, "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 (ASPLOS)*, 2024.

[23] Z. Wang, Q. Lin, Y. Zhang, *et al.*, "Asyfunc: A high-performance and resource-efficient serverless inference system via asymmetric functions," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2023.

[24] X. Xie, X. Ding, Z. Yang, *et al.*, "Pre-warming is not enough: Accelerating serverless inference with opportunistic pre-loading," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2024.

[25] W. Shen, C. Wang, Y. Zhang, *et al.*, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[26] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 181–188.

[27] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach."

[28] Y.-H. Chiang, C. Zhu, H. Lin, and Y. Ji, "Hysteretic optimality of container warming control in serverless computing systems," *IEEE Networking Letters*, vol. 3, no. 3, pp. 138–141. Conference Name: IEEE Networking Letters.

[29] E. Hunhoff, S. Irshad, V. Thurimella, A. Tariq, and E. Rozner, "Proactive serverless function resource management," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, pp. 61–66, Association for Computing Machinery.

[30] R. B. Roy, T. Patel, and D. Tiwari, "IceBreaker: warming serverless functions better with heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pp. 753–767, Association for Computing Machinery.

[31] S. Agarwal, M. A. Rodriguez, and R. Buyya, "A reinforcement learning approach to reduce serverless function cold start frequency," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 797–803.

[32] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," pp. 419–434.

[33] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," pp. 419–433.

[34] "Overview of memory management | app quality."

[35] S. Qin, H. Wu, Y. Wu, B. Yan, Y. Xu, and W. Zhang, "Nuka: A generic engine with millisecond initialization for serverless computing," in *2020 IEEE International Conference on Joint Cloud Computing*, pp. 78–85.

[36] Z. Xu, H. Zhang, X. Geng, Q. Wu, and H. Ma, "Adaptive function launching acceleration in serverless computing platforms," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 9–16. ISSN: 1521-9097.

[37] K. Solaiman and M. A. Adnan, "WLEC: A not so cold architecture to mitigate cold start problem in serverless computing," in *2020 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 144–153.

[38] A. Suresh, G. Somashekar, A. Varadarajan, V. R. Kakarla, H. Upadhyay, and A. Gandhi, "ENSURE: Efficient scheduling and autonomous resource management in serverless environments," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 1–10.

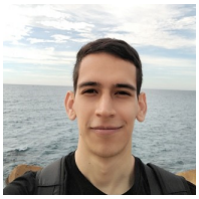
[39] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with OpenLambda,"

[40] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. 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.

[41] "cgroups - linux manual page."

[42] M. Shahrad, R. Fonseca, Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," pp. 205–218.

IX. Biography Section



Nicolas Buitrago received his Computer Science and Engineering degree at Universidad del Norte in Colombia in 2021. He then graduated with a Master of Science in Computer Science and Engineering from the same university in 2024. His research interests include database design, cloud computing, machine learning and serverless computing.

Hector Camacho received his Computer Science and Engineering degree at Universidad del Norte in Colombia in 2021. He then graduated with a Master of Science in Computer Science and Engineering from the same university in 2024. His research interests include cloud computing, software engineering and machine learning.

Miguel Jimeno (M' 05), received his Computer Science and Engineering degree at Universidad del Norte in Colombia in 2002. He then graduated with a Master of Science in Computer Science and Engineering and a PhD in Computer Science from the University of South Florida in 2007 and 2010, respectively. His research interests include cloud computing, deep learning, and cloud deployments of e-learning systems and healthcare information systems. Miguel's experience started as a software engineer, and now as an associate professor at Universidad del Norte. He was awarded several national and international research grants.

Cesar Viloria-Nuñez (IEEE Senior Member) was born in Barranquilla, Colombia, in 1985. He received his degree in Electronics Engineering from Universidad del Norte, Colombia, in 2008, his M.Sc. in Computer Science and Engineering from the same institution in 2010, and is currently pursuing a PhD in Economics and Innovation Management from Universidad Autónoma de Madrid. He currently serves as the Dean of the Digital Transformation School at Universidad Tecnológica de Bolívar. He has directed projects in areas such as Digital Transformation, Digital Platform Development, and Artificial Intelligence. His research interests focus on disruptive technologies and technology management.

Jairo A. Cardona received the B.S. degree in Telecommunications and Electronics Engineering and the M.Sc. degree in Telematics from the Universidad del Cauca, Colombia, in 1995 and 2005 respectively. He is a Ph.D. in Computer Science student and an Assistant Professor of the Electrical and Electronics Department at Universidad del Norte. His research and teaching interests include the development of computational intelligence approaches applied to several application domains and technology. His research expertise is related to Computer Networks, Software-Defined Networks and Quality of Experience

Augusto Salazar received his Computer Science and Engineering degree at Universidad del Norte in Colombia in 2003. He then graduated with a Master of Science in Computer Science and Engineering from National Chiao Tung University in 2011. Starting from 2012 he has held a position as assistant professor professor at Universidad del Norte. His research interests include edge computing, gamification, privacy and IoT.