



# AlloyStack: A Library Operating System for Serverless Workflow Applications

Jianing You[†], Kang Chen[‡], Laiping Zhao[†*], Yiming Li[†], Yichi Chen[†], Yuxuan Du[†], Yanjie Wang[†], Luhang Wen[†], Keyang Hu[‡], Keqiu Li[†]

[†] College of Intelligence & Computing, Tianjin University, Tianjin Key Lab. of Advanced Networking, China

[‡]Tsinghua University, China

## Abstract

Serverless workflow applications, composed of multiple serverless functions, are increasingly popular in production. However, inter-function communication and cold start latency remain key performance bottlenecks. This paper introduces AlloyStack, a library operating system (LibOS) tailored for serverless workflows. AlloyStack addresses two major challenges: (1) reducing cold start latency through on-demand OS component loading and (2) minimizing data transfer overhead by enabling functions within the same workflow to share a single address space, eliminating unnecessary data copying. To ensure secure isolation, AlloyStack uses Memory Protection Keys (MPK) to separate user functions from the LibOS while maintaining efficient data sharing. Our evaluation shows that AlloyStack reduces cold start times by 98.5% to just 1.3*ms*. Compared to SOTA systems, AlloyStack achieves a 7.3× to 38.7× speedup in Rust end-to-end latency and a 4.8× to 78.3× speedup in other languages for intermediate data-intensive workflows.

***CCS Concepts:*** • **Computer systems organization** → **Cloud computing**.

***Keywords:*** Serverless Computing, Cold Start, Data Transfer

**ACM Reference Format:**

Jianing You, Kang Chen, Laiping Zhao, Yiming Li, Yichi Chen, Yuxuan Du, Yanjie Wang, Luhang Wen, Keyang Hu and Keqiu. 2025. AlloyStack: A Library Operating System for Serverless Workflow Applications. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3717490

*Corresponding author: laiping@tju.edu.cn



*EuroSys '25, March 30-April 3, 2025, Rotterdam, Netherlands*



ACM ISBN 979-8-4007-1196-1/2025/03…$15.00
https://doi.org/10.1145/3689031.3717490

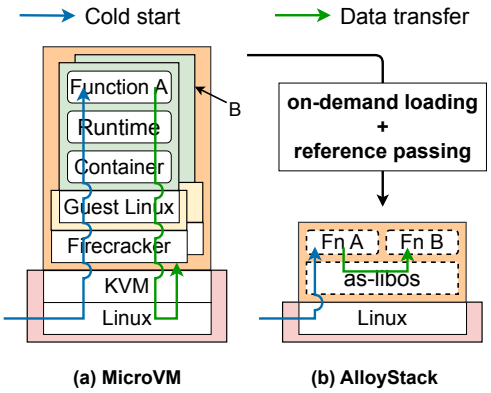


**Figure 1.** Overview of workflow deployment across different software stacks. "Cold start" indicates the path for triggering the workflow, while "Data transfer" denotes the path for moving intermediate data.

## 1 Introduction

A *serverless workflow* is a workflow design and execution model within a serverless architecture, enabling developers to orchestrate functions and their dependencies without managing servers [29, 48]. Today, *serverless workflows* are widely used in production, with major cloud providers like AWS, Azure, and Aliyun offering workflow orchestration services [14, 15, 19]. Studies show that 31% of serverless applications are built using workflows [29], and the top 5 most frequently invoked workflows in Azure Durable Functions account for 46% of total invocations [48].

The *serverless workflow* model exhibits two distinctive characteristics. *First, it relies on multiple small, single-purpose functions that collaborate to complete an execution path.* These functions are highly specialized, typically small in size, and have very short execution times—50% of them run for less than 1s on average [60]. However, to ensure isolation, these functions must be deployed in heavyweight sandboxes (e.g., MicroVM [20] and Kata container [56]), which introduces significant cold start latency (e.g., 200ms for a MicroVM [63]). As a result, the cold start problem is even more pronounced in *serverless workflows*, since each function may incur its own cold start delay. *Second, these functions are interdependent, often requiring substantial data exchanges.* Because function

instances are unaware of each other's locations, their communication relies on a third-party controller, typically external storage services such as Redis or S3 [59], resulting in inefficient communication [34]. This overhead, especially over the network, can account for up to 25% of total CPU cycles [36] and 77.6% of overall latency [45]. Therefore, accelerating workflow execution requires improvements in both *cold start latency* and *data transfer efficiency*.

To reduce the cold start latency introduced by heavyweight sandboxes, existing work mainly employs two approaches: *warm start* and *specialization*. *Warm start* approaches pre-starts the environment before a request arrives [60, 71], but they require accurate prediction of request times. Otherwise, they incur significant memory overhead (e.g., achieving a cold start rate below 1% may require an extra 40% of memory [71]). In contrast, *specialization* tailors the runtime by removing unnecessary components, thereby reducing startup time. However, current specialization methods lack the fine granularity needed for small functions. As shown in Figure 1, although a MicroVM accelerates cold starts by simplifying the device model [20], it still includes all major components of the guest Linux kernel. Our analysis reveals that many functions do not require the full kernel; for example, functions in ServerlessBench [72] typically need only 30–50% of the kernel components. Therefore, the first challenge we address is **(C1)** accelerating function startup through more fine-grained OS specialization.

To improve intermediate data transfer efficiency, existing approaches often rely on an orchestrator to coordinate functions, with external storage services (e.g., Redis, S3 [59]) facilitating data transfer between functions [60]. However, these methods introduce overhead due to high latency and multiple data copies. Previous work proposes *offloading control* (shifting control from the central orchestrator to local instances to reduce forwarding overhead [44]) or *keep-alive connections* (maintaining connections between functions to avoid the cost of repeatedly establishing connections [27, 45]). Thread-level function abstractions [37, 58, 61, 66] run functions inside different threads, avoiding redundant data copies by sharing memory address space. This reduces overhead, but introduces security concerns. A key challenge we aim to address is **(C2)** ensuring secure memory sharing between functions in the workflow.

We designed our library operating system (LibOS), AlloyStack, based on the observation that functions within the same *serverless workflow* typically belong to the same tenant. As a result, isolation between functions within the same workflow is less critical than between different workflows. Thus, AlloyStack allows multiple functions within the same workflow to share a single isolated environment [21, 34, 37, 67], as shown in Figure 1(b). This shared address space eliminates data copying and significantly reduces the overhead of intermediate data transfer between functions. To address the first challenge **(C1)**, components of AlloyStack are started on demand, drastically reducing cold start time and minimizing memory usage. Our tests show that function startup time is reduced to 1.3 ms. To address the second challenge **(C2)**, functions from different workflows use separate LibOS instances. AlloyStack employs Memory Protection Keys (MPK) [32] to isolate user functions from the LibOS.

AlloyStack is implemented from scratch using the Rust programming language. To support multiple programming languages, we introduced a WASM runtime, allowing C and Python applications to run without modifying their source code. To demonstrate the efficiency and practicality of AlloyStack, we ported applications in Rust, C, and Python and conducted comprehensive evaluations. AlloyStack has been open-sourced on GitHub [1].

In this work, we make the following contributions:

- We analyzed existing systems for serverless workflows and found that no current system effectively addresses both cold start delays and data transfer bottlenecks.
- We propose AlloyStack, a LibOS system tailored for serverless workflows. It supports on-demand loading of OS modules and enables reference passing in a single address space for efficient data sharing. AlloyStack is implemented in Rust and can support C and Python through WASM.
- Compared to SOTA systems, AlloyStack achieves a 7.3× to 38.7× speedup in Rust benchmarks and a 4.8× to 78.3× speedup in C and Python benchmarks.

## 2 Background & Motivation

### 2.1 Serverless Workflows

Workflows are typically represented by a DAG (Directed Acyclic Graph). In serverless workflows, the number of function instances is dynamically adjusted during execution to handle fluctuating loads, requiring real-time decisions to launch new instances and select which instance of a function to run.

Given a serverless workflow DAG, each function's execution is divided into three stages: (1) *instance cold start*, (2) *instance execution*, and (3) *data transfer* to downstream functions. To measure the cost of stages (1) and (3), we deploy the *ParallelSorting* benchmark application on OpenFaaS [4]. We use a MicroVM as the isolation sandbox by setting the container runtime to *Kata containers* [56] and configuring it to use *Firecracker* [20] as the hypervisor.

Our findings show that with a 50MB input, the *instance cold start* latency accounts for 45% of the end-to-end latency, while the *data transfer* stage accounts for 48%. Thus, cold start and data transfer are the primary bottlenecks affecting execution efficiency, indicating that current serverless software stacks are ill-suited for serverless workflows.

[1] https://github.com/tanksys/AlloyStack

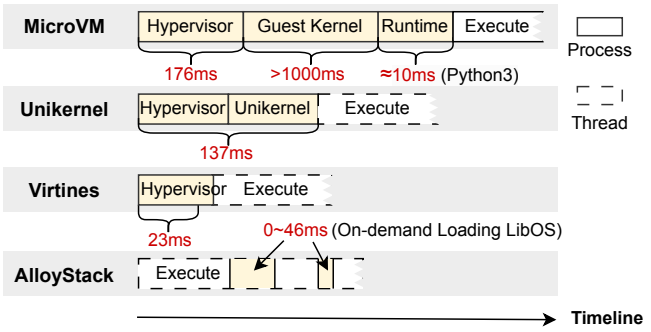


**Figure 2.** Time for each phase of cold start. MicroVM uses Firecracker, and Unikernel uses Unikraft.

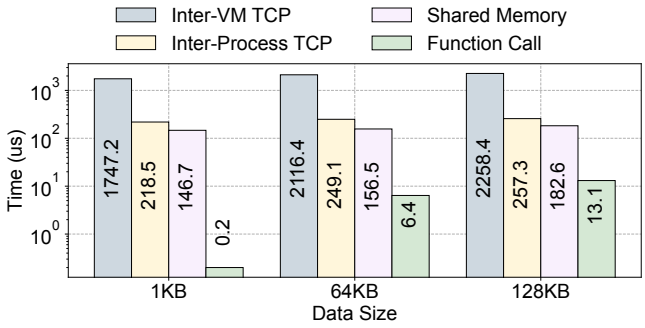


**Figure 3.** Performance of different communication primitives.

### 2.2 LibOS Specialization

To accelerate *cold start*, an effective approach is to reduce the amount of data loaded during startup. In *serverless workflows*, functions are highly specialized and typically small, so eliminating unnecessary runtime components can significantly shorten startup time. We analyzed the kernel components required by typical functions in ServerlessBench [72] (see Table 1). Most functions require only 3 to 5 components. For example, the *store-image-metadata* function in an image processing workflow updates the database metadata after processing. It uses *time* to retrieve the current date, *net* to connect to the database, and *mm* for dynamic memory allocation to store JSON objects. In fact, running the top 20 most downloaded applications from DockerHub [1] requires enabling only 302 out of 16,000 Linux compilation options [39].

**Table 1.** Kernel modules for different serverless functions.

| Function Name | Required kernel components |
|---|---|
| alu | mm |
| parallel-alu | time, irq, sched, locking, mm |
| long-chain | mm |
| extract-image-metadata | time, mm, block, fs, net |
| transform-metadata | time, mm |
| handler | time, mm, net |
| thumbnail | time, mm, block, fs, net |
| store-image-metadata | time, mm, net |
| online-compiling | time, irq, sched, locking, mm, ipc, block, fs, net |

Existing serverless systems leverage *trimming* to reduce the high startup overhead caused by heavy sandboxes. As shown in Figure 2, the MicroVM approach trims the device model by removing features that operate at a basic system level and are not required for serverless scenarios, such as BIOS, legacy devices, and PCI, while retaining essential operating system interfaces like the file system and networking. This optimization reduces startup latency from $1,817ms$ to approximately $1,186ms$ [20]. By completely removing the guest kernel and constructing the function execution environment solely through KVM, Virtines [66] further reduces the startup latency to $23ms$. Although Virtines is fast, removing the guest kernel can reduce isolation between functions. Since system calls are passed directly to the host kernel, the host becomes vulnerable to attacks via these calls, leaving serverless platforms without an extra layer of protection.

To remove unnecessary components in the guest kernel while keeping user functions separate from the host kernel, a practical solution is to use a Library Operating System (LibOS) [38, 46, 68]. This approach divides the monolithic operating system kernels into several small libraries and compiles each function with only the necessary components, greatly reducing the kernel size. For example, the LibOS image for Nginx can be reduced to $1.6MB$. Figure 2 shows that the startup latency of Unikernel, a LibOS for cloud use, is $137ms$, which is much shorter than that of a MicroVM.

In workflows with multiple functions, small libraries are often reused. Reducing LibOS redundancy remains a significant challenge for serverless workflow systems.

### 2.3 Address Space Sharing

To *measure data transfer* between functions, we compare four common data transfer methods used in serverless workflow: (1) *Inter-VM TCP*: data transfer between MicroVMs using TCP sockets, (2) *Inter-Process TCP*: data transfer using TCP sockets within a single MicroVM, (3) *Shared Memory*: data transfer within a MicroVM using shared memory mapping [34, 55], and (4) *Function Call*: direct memory access between threads [37, 61]. In method (1), two MicroVMs are launched in advance. The measurement for methods (1) and (2) spans the interval from establishing a TCP connection to receiving all data. In method (3), we create a file in ramfs in advance and map it into the address spaces of both the sender and receiver processes using *mmap()*. After data initialization, the sender writes one byte to a Linux pipe to notify the receiver. The measurement covers the interval from just before writing to the pipe until after the receiver has traversed the mapped

memory region. In method (4), we allocate the data buffer using an anonymous mapping. After data initialization, the sender immediately calls the receiver function.

Clearly, in methods (1) through (3), functions run in separate processes, while in method (4) the entire serverless workflow operates as a single process with functions running as threads. The **address space sharing** allows functions to transfer data using simple load/store instructions.

As shown in Figure 3, method (4) outperforms other data transfer mechanisms by 1 to 2 orders of magnitude. This is because threads can naturally share the address space without any syscalls, allowing the CPU to access data more efficiently compared to complex kernel functionalities such as sockets and memory mapping. This motivates the use of a single address space for the serverless workflow, improving intermediate data transfer efficiency.

### 2.4 Summary

The multiple small-sized functions and their dependencies in serverless workflows cause cold starts and data transfer bottlenecks in end-to-end latency. While current solutions partly address these challenges, no system can efficiently resolve both issues. Although using a LibOS can meet the specialized needs of individual functions, it may lead to redundant loading of LibOS components within a workflows deployment. Furthermore, while address space sharing greatly reduces data transfer latency, it compromises isolation between functions. Achieving address space sharing while maintaining secure isolation remains a major challenge.

## 3 AlloyStack System Design

Commercial serverless platforms (such as AWS Step Functions [14] and Azure Durable Functions [15]) typically allow only functions from the same tenant to be orchestrated into workflows. Because the isolation requirements within a single tenant are less strict than those across tenants, we propose a workflow domain (WFD) design. In this design, functions from the same tenant are grouped into a WFD, where weak isolation is applied, while strong isolation is maintained between different WFDs.

### 3.1 The Workflow Domain Abstraction

The WFD (**W**ork**F**low **D**omain) is an abstraction of a workflow execution environment that uses a single address space to associate all entities required for a workflow, including user functions, LibOS, heap memory, and other system resources. Within the WFD, the address space is divided into a system partition and a user partition using MPK, a lightweight memory partitioning mechanism [32]. Functions within a WFD can share system contexts, such as operating on the same file. In contrast, functions from different workflows cannot access each other's files across WFDs because system calls are processed through separate LibOS instances.

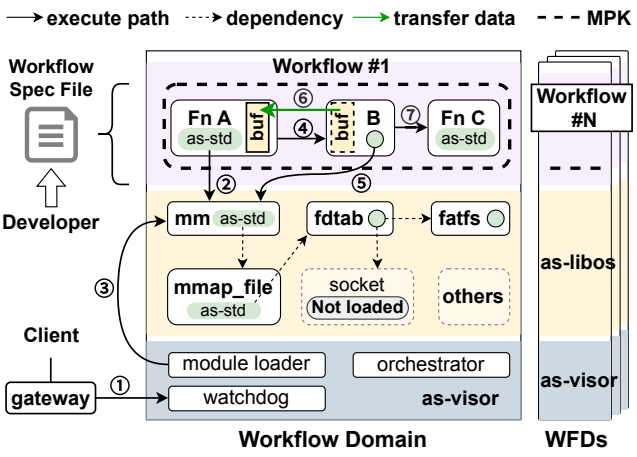


**Figure 4.** AlloyStack architecture.

WFD is the basic unit for workflow deployment. To reuse device drivers and other low-level primitives provided by the Linux kernel, each WFD is deployed within a user-space process without using hardware virtualization. Extremely large workflows that cannot be accommodated on a single node can be split into multiple WFDs based on the DAG and deployed across multiple nodes.

WFD provides isolation guarantees among different workflows, including memory, kernel resource, and fault isolation. Failures caused by data issues or bugs do not affect other WFDs. These guarantees support a retry-based fault tolerance mechanism. For workflows composed of idempotent functions, AlloyStack can recover from occasional bugs by restarting the workflow, though this approach may reduce efficiency. If a failure does not affect the `as-libos` (see below) and the workflow's intermediate data remains intact, AlloyStack can restart only the failed function. For stateful functions, which are not idempotent, fault tolerance cannot be achieved by restarts. Some works, such as Boki [33] and Halfmoon [54], address this issue. AlloyStack could mitigate this limitation by integrating these approaches.

### 3.2 System Architecture

Figure 4 shows the architecture and execution steps of AlloyStack. AlloyStack is deployed as processes across multiple nodes, receiving invocations that are load-balanced by a gateway. In each AlloyStack process, WFD provides a complete software stack for running workflow instances, consisting of three components: the standard library layer `as-std`, the kernel functionality layer `as-libos`, and the global runtime layer `as-visor`. Note that these layers differ from those in hypervisor, guest kernel, or container systems, as they are not separated into different virtual address spaces.

For a chained workflow with functions A, B, and C, AlloyStack executes as follows: `as-visor` binds the workflow to a specific HTTP endpoint and exposes it externally. When the

watchdog receives a workflow invocation event ①, as-visor instantiates the corresponding WFD (§ 3.1) for the workflow. The thread of function A (labeled as Fn A) begins execution at its entry point and runs until it calls an API from a LibOS module that is not yet initialized ②. as-std invokes the on-demand loading interface ③ (§ 4) provided by as-visor to retrieve the address of *alloc_buffer()* (Table 2), enabling the writing of intermediate data into the buffer (§ 5). In the subsequent stages ④, function B (Fn B) follows a similar process until it calls *acquire_buffer()* ⑤ to obtain a reference to the buffer, which allows zero-copy reading of intermediate data ⑥. After function C completes execution ⑦, as-visor destroys the WFD and reclaims the associated resources.

### 3.3 Global Runtime Layer: as-visor

AlloyStack uses as-visor to manage and trigger workflows. This module, which includes a watchdog, an orchestrator, and an MPK-enabled module loader, is independent of the specific user workload.

The watchdog is an HTTP server that listens for external invocation events. The orchestrator creates and runs parallel threads in stages, executing functions as specified by the workflow's configuration file. It also monitors essential host resources for workload execution, including user threads and memory segments (code segment, heap, stack, trampoline code, and the associated protection key), virtual network devices, and virtual disk images.

By default, AlloyStack assumes that functions within the same workflow belong to the same tenant and trust each other, allowing them to share MPK permissions. However, this is not always true. For example, the FINRA workflow processes highly sensitive trade data and requires stricter isolation between functions [37]. To achieve this, as-visor also supports the use of MPK to create separate memory partitions for each function instance, allowing tenants to enable isolation between functions in the same WFD.

The as-visor layer divides the process address space into two partitions: a system partition and a user partition. The system partition contains components such as as-visor and as-libos, while the user partition holds user functions and the trampoline pages for switching protection keys. as-visor assigns a unique protection key to each partition (or a unique protection key for each function if isolation between functions is enabled) and binds them at the memory page level. By organizing the MPK memory partitions and controlling the PKRU register values, as-visor ensures that, when operating in the user context, access to data in the system partition is prohibited.

### 3.4 Kernel Functionality Layer: as-libos

To control the execution, user functions cannot directly invoke system calls to the host kernel. Instead, AlloyStack allocates a separate LibOS, as-libos, for each WFD. The as-libos layer provides interfaces similar to syscalls, such as *open()*, *write()*, and *bind()*.

```
// on the standard Linux platform.
// use std::{io::{Read, Write},net::TcpStream};
// on the AlloyStack platform.
use as_std::{prelude::*, io::{Read, Write}, net::
    TcpStream};
let mut stream = TcpStream::connect("example.com");
stream.write_all(b"GET / HTTP/1.0\r\n\r\n");
stream.read(&mut [0; 4096]);
```

**Figure 5.** An example of a simple HTTP client function implemented using as-std.

Since serverless functions are small and designed for single-purpose tasks, AlloyStack does not reuse existing LibOS implementations because they are not optimized for serverless workflows. Instead, as-libos has been redesigned with a modular architecture so that modules can be loaded on demand. As shown in Figure 4, at the start of workflow execution, no as-libos modules are instantiated within the WFD until the user function indirectly invokes a syscall through as-std. Additionally, because there are dependencies among as-libos modules, as-std is compiled into them to enable inter-module communication using the same mechanism.

### 3.5 Standard Library Layer: as-std

Similar to other programming platforms, as-std provides a standard library designed for three main purposes. First, it intercepts user function syscall requests and routes them to the syscall handler of the as-libos. Ultimately, the user function's image must not contain blacklisted instructions such as syscall or sysenter. Second, it abstracts the differences between the underlying as-libos and other kernels at the API level. Within as-std, the syscall interface address of the as-libos in this WFD is tracked, enabling trampoline calls between user functions and as-libos modules, as well as among as-libos modules themselves once the relevant modules are loaded. For example, Figure 5 shows how to implement simple HTTP request sending and response reading using as-std. Users only need to replace the original "std" with "as_std" without concerning themselves with on-demand loading and interface binding details of as-libos. Finally, as-std is responsible for switching the MPK access permissions of the current thread through a trampoline code segment before transferring control to as-libos.

For function developers, as-std minimizes modifications to existing application logic code. They only need to adapt the code for intermediate data transmission to the dedicated AsBuffer interface provided by as-std. We have primarily defined the APIs for as-std and AsBuffer in Rust (as illustrated in Figure 6). To ensure compatibility with other languages, we have also designed an adaptation layer between

as-std and WASI, allowing developers to build functions without being restricted to Rust.

```
pub struct AsBuffer<T> {}
impl<T> AsBuffer<T> where T: FaasData {
    // Create a buffer.
    pub fn with_slot(slot: &str) -> Self {}
    // Reference the specified buffer through a slot.
    pub fn from_slot(slot: &str) -> Self {}
}
```

**Figure 6.** AsBuffer interface.

## 4 On-demand Loading for WFD Cold Start

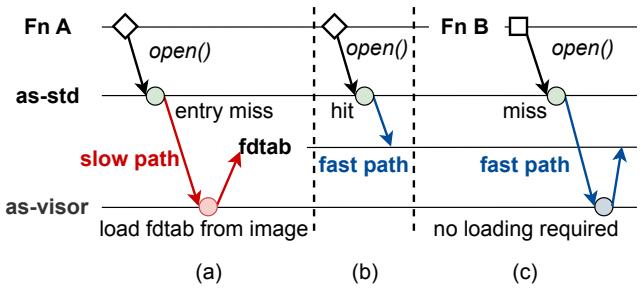


**Figure 7.** An illustration of the on-demand loading process using the fdtab module as an example.

AlloyStack ensures a fast cold start for each WFD. Due to the simplicity of components such as the watchdog and orchestrator, the cold start overhead of as-visor (approximately 78 µs on our machine) is negligible. Thus, in the AlloyStack architecture, the main source of cold start latency for a workflow is the initialization of as-libos.

as-libos addresses the cold start issue by introducing an on-demand loading mechanism. Figure 7 illustrates the loading process when Function *A* and Function *B* in a workflow invoke the *open()* interface multiple times. Initially, no as-libos modules are loaded in the WFD. When Function *A* needs to call *open()* (Figure 7(a)), it first passes through the interface provided by as-std under Rust's safety constraints. Since this is the first invocation and triggers an entry miss, as-std calls as-visor. as-visor, which manages all WFD resources, detects that the fdtab module is not loaded and instantiates it via the module loader. This process is referred to as the *slow path*.

After fdtab is loaded, as-std records the address entry for *open()*. Subsequent invocations of *open()* by Function *A* (Figure 7(b)) do not trigger an entry miss. This is the *fast path*, as no additional loading is required. Since AlloyStack runs multiple functions of a workflow within the same WFD, sharing a single as-libos instance, subsequent functions can reuse modules already loaded by earlier functions. For example, when Function *B* invokes *open()* (Figure 7(c)), it does not need to load the module again.

```
#[derive(FaasData, Default)]
pub struct MyFuncData {
    pub name: String,
    pub year: u64,
}
fn func_a() { // Data sender.
    let mut data: AsBuffer<MyFuncData>
        = AsBuffer::new_slot("
        Conference");
    data.name = "Euro";
    data.year = 2025;
}
fn func_b() { // Data receiver.
    let data: AsBuffer<MyFuncData> =
        AsBuffer::from_slot("
        Conference");
    println!("{}Sys, {}", data.name,
        data.year) // "EuroSys, 2025".
}
```

**Figure 8.** A simple demo to illustrate the transmission of intermediate data within a WFD in AlloyStack.

## 5 Reference Passing for Intermediate Data Transfer

AlloyStack allows multiple functions within a single WFD to share the same address space, enabling intermediate data transfer via references. AlloyStack provides a dedicated communication API, AsBuffer, for such transfers between functions in a workflow. The creation of an AsBuffer and the access to it are managed through two interfaces provided by as-libos: *alloc_buffer()* and *acquire_buffer()* (details in Table 2).

In real-world scenarios, many workflows are not simple chained invocations but complex DAGs [48]. To support such workflows, we introduce a *slot* parameter in the AsBuffer API. The slot uniquely identifies a buffer within the WFD, effectively creating a namespace for AsBuffer. as-libos maintains the mapping between each slot and its corresponding memory region. By using different slots, user functions can create multiple AsBuffer instances simultaneously, allowing different downstream functions to receive them (fan-out). Likewise, functions can receive intermediate data from multiple upstream functions (fan-in). Figure 8 illustrates a basic example of AsBuffer. Function *A* creates a AsBuffer with the slot "*Conference*" and writes the data into it. Function *B* then accesses the same buffer by specifying the identical slot, retrieves the data, and processes it for subsequent application logic.

## 6 Threat Model

To ensure that security policies (e.g., MPK and LibOS) are not bypassed, the cloud platform must guarantee that user function images do not include blacklisted instructions such as `wrpkru`, `syscall`, `sysenter`, and `int`. This requirement is achievable because detection can be implemented using tools like objdump [18], Dyninst [16], and E9Tool [17]. The identification of instructions can suffer from "false positives", where users may unintentionally include prohibited instructions. However, ERIM [64] proposes a solution based on binary rewriting. For instance, `nop` instructions can be inserted between adjacent instructions that might combine into a `wrpkru` and immediate value instructions can be transformed into variants that use registers. These techniques can be applied within AlloyStack. Since instruction identification is performed before the initiation of the workflow, dynamic inspection by LibOS is unnecessary. Additionally, the WASM runtime integrated into AlloyStack must also exclude any blacklisted instructions. We currently enforce this using the `no_std` feature of Wasmtime [23]. To accelerate WASM execution, AlloyStack employs AOT compilation to convert WASM images into the ELF format. This approach remains compatible with instruction detection mechanisms while preserving AlloyStack's isolation model.

The security policies in AlloyStack do not impose an intolerable burden on users during image construction. For images generated using Rust, this requirement can be easily met by using the `as-std` library we provide. For other languages, we require that WASM images interact with the underlying environment via WASI.

When the aforementioned condition is satisfied, AlloyStack can guarantee the following:

**(1) Functions within a WFD will not interfere with each other**. AlloyStack allocates exclusive heap, code, and data segments for each function, and assigns each thread an independent stack. Unless a user function incorrectly dereferences a pointer, the memory regions accessed by different functions within the same workflow do not overlap. Since `as-libos` and `as-visor` are both protected by MPK, functions cannot compromise them. Therefore, ensuring the correct implementation of the loader is sufficient to guarantee function separation. In addition, AlloyStack offers separate memory partitions for functions containing sensitive data through tenant-configurable options (§ 3.3).

**(2) Prevent `as-libos` from being bypassed**. MPK ensures that user functions can only access their own memory segments and cannot bypass `as-std` to read, write, or execute other memory regions. Since serverless platforms typically run arbitrary binary images uploaded by users, it is possible that users may compile function images using the default standard library instead of `as-std`, or inline instructions such as `wrpkru` and `syscall` within their functions. These cases are detectable, as discussed before.

**(3) Isolation between WFDs**. All interactions between the WFD and external environments are exclusively managed by `as-libos`. Since `as-libos` is written in Rust, most of the code benefits from the safety guarantees provided by safe Rust. For the small portion of code written in unsafe Rust, safety is ensured through careful auditing and inspection [25, 26, 35, 62]. As user functions cannot directly invoke syscalls, this design helps mitigate attacks from malicious functions. Additionally, the one-to-one deployment of WFDs with processes further enhances isolation.

Similar to other container platforms, AlloyStack does not implement specific defenses against side-channel attacks, such as Meltdown [43]. It lacks control over the underlying software and hardware behaviors, making it unable to address side-channel vulnerabilities. However, AlloyStack is fully compatible with existing mitigation measures, such as disabling SMT and KSM [49].

## 7 Implementation

AlloyStack is built with over 7,700 lines of Rust code. Its three core modules are `as-visor` (over 2,300 lines), `as-libos` (over 3,500 lines), and `as-std` (over 1,900 lines). To evaluate its performance, we developed benchmarks in Rust, C, and Python, consisting of 2,800 lines of code.

### 7.1 AlloyStack Components

For `as-visor`, we develop the *find_hostcall()* interface using the dynamic linker's *dlmopen()* function to enable on-demand loading. We use the *pkey_mprotect* system call to secure memory in system and user partitions. Additionally, we support multithreading, using Linux threads created via the *clone()* system call and managed by the kernel scheduler (CFS). Furthermore, we implement a gateway that triggers via CLI and HTTP and executes workflows from JSON configurations.

For `as-libos`, we reuse the open-source projects rust-fatfs [6] and ruxos [13] to implement file system interfaces. We also use smoltcp [2] and Linux's pseudo-device to create a TAP virtual device for each WFD, configured by the host OS to obtain independent IP addresses. This enables us to develop the socket interface—including *bind()*, *connect()*, and *accept()*—for implementing a TCP network stack in user space. Moreover, we customize the *alloc_buffer()* and *acquire_buffer()* interfaces to facilitate heap buffer allocation and allow functions to pass intermediate data. The *alloc_buffer()* function passes the required buffer size and alignment information to `as-libos`. The mm module in `as-libos` then allocates the buffer from the heap and records its slot and base address as a key-value pair. When the receiver invokes *acquire_buffer()*, `as-libos` queries and returns the buffer address based on the provided slot, and removes the slot entry to prevent multiple functions from owning the same buffer. Table 2 summarizes the interfaces supported by `as-libos`. To map files managed by `as-libos` into

**Table 2.** `as-libos` modules currently implemented in AlloyStack.

| as-libos Modules | Representative APIs | Description |
|---|---|---|
| mm | mmap(length, prot, fd) -> Result<usize> | Map a file to memory |
| | alloc_buffer(slot, layout, fingerprint) -> Result<usize> | Allocate buffers for intermediate data |
| | acquire_buffer(slot, fingerprint) -> Result<(usize)> | Access intermediate data |
| fdtab | open(path, flags, mode) -> Result<Fd> | Open/close/read/write file descriptors |
| fatfs | fatfs_open(path, flags) -> Result<Fd> | Manipulate files located within the file system image |
| | fatfs_write(path, buf) -> Result<Size> | |
| socket | smol_bind(addr) -> Result<SockFd> | TCP network communication provided by the LibOS network stack |
| | smol_connect(sockaddr) -> Result<SockFd> | |
| stdio | host_stdout(buf) -> Size | Write to the host console's stdout |
| mmap_file_backend | register_file_backend(mm_region, file_fd) -> Result<()> | Implement user-space page fault handling |
| time | gettimeofday() -> Result<u128> | Get the host's Unix timestamp |

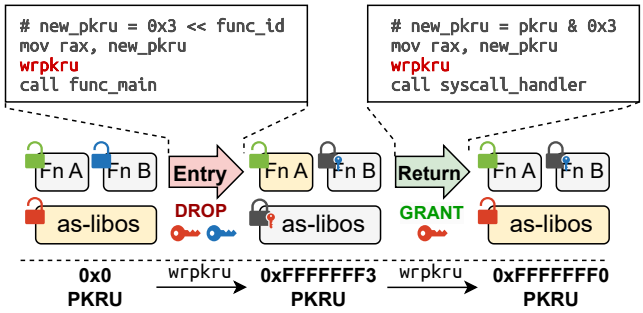


**Figure 9.** An example of adjusting MPK permissions via trampolines during function execution and syscall execution.

the address space, we implement the mmap_file_backend module. This module leverages Linux's Userfaultfd feature to receive notifications and handle page faults.

For `as-std`, we implement the control and switching of MPK access permissions using trampoline. As shown in Figure 9, the purpose of the trampoline is to elevate the current CPU's PKRU permissions before executing the system partition's code. This is achieved by writing assembly code to save the context, switch stacks, update the PKRU register, and perform the jump. When returning to user code, the reverse operations are performed to discard the PKRU permissions.

We use *linked_list_allocator* [3] as the default memory allocator, enabling dynamic memory allocation and easy recovery by heap units if functions crash. The `as-std` interface is similar to the Rust standard library. An adaptation layer bridges the WASI interface between the WASM runtime and `as-std`, connecting APIs with the same semantics via wasmtime's Linker.

### 7.2 Multi-language Support

To enhance system compatibility, AlloyStack supports applications in multiple programming languages. Specifically, to support applications written in C and Python, we utilize WASM to compile the code into WASM modules that can be executed on the wasmtime runtime [5]. We use a simple wrapper layer to import wasmtime's Rust SDK, disguising the WASM runtime as a regular Rust user function. To meet the requirement of avoiding blacklist instructions as discussed in § 6, we enable wasmtime's no_std feature [23]. This ensures that wasmtime can only perform OS operations through the APIs provided by AlloyStack.

We develop an adaptation layer in `as-std` to bridge the wasmtime and WASI interfaces, enabling the interactions between WASM functions and `as-libos`. All syscalls triggered by functions are forwarded to `as-libos` and processed through `as-std`. Currently, we have implemented 15 WASI interfaces and added two customized WASM interfaces, *buffer_register()* and *access_buffer()*, for intermediate data transfer. However, since Rust supports generics, the `AsBuffer` API (§ 5) provides richer semantics and greater ease of use compared to other languages. In contrast, C and Python currently only support data transfer in the form of strings.

We utilize the open-source CPython [7] and its officially provided WASM platform compilation scripts to run Python applications on AlloyStack without modifications. To provide the `AsBuffer` API to Python functions, we enhance the CPython runtime by adding a custom module called *pyasbuffer*. In the implementation of *pyasbuffer*, we use the compiler annotation *__attribute__()* to integrate it with `as-libos`. This does not break compatibility with existing Python functions, as they can ignore *pyasbuffer*. However, it also prevents leveraging AlloyStack's communication acceleration capabilities and restricts data transfer to traditional methods such as networks or files.

## 8 Evaluation

Our evaluation answers the following questions:

- Does the on-demand loading technique of AlloyStack have a significant impact on cold start optimization?
- Is the intermediate data transfer mechanism efficient?

- Does AlloyStack significantly accelerate the end-to-end latency of workflows composed of multiple functions?
- Can AlloyStack support workflow applications beyond Rust? Is the acceleration effect applicable to functions in other programming languages as well?

### 8.1 Methodology

We deploy AlloyStack on a machine as shown in Table 3.

**Table 3.** Experimental machine configurations.

| | Configuration |
|---|---|
| Hardware | CPU : Intel(R) Xeon(R) Gold 6338 CPU |
| | CPU Threads: 128 (64 physical cores) |
| | Processor Freq: 2.00GHz |
| | DRAM: 256GB DDR4, Disk: 1.8TB SSD |
| Software | Operating system: Ubuntu 22.04 |
| | Kernel: Linux 6.5.0-41-generic |
| | Compiler: rustc 1.76.0-nightly |

**Benchmarks.** We evaluate AlloyStack and the comparison systems using both synthetic benchmarks and real-world benchmarks. We implement three synthetic benchmarks: *no-ops*, *http-server*, and *pipe*. *no-ops* is an empty function that returns immediately. *http-server* is a server that returns a fixed response. *pipe* is an application consisting of two functions, where specific-sized intermediate data is transferred. We further choose three real-world applications for evaluation , including *FunctionChain*, *WordCount* and *ParallelSorting*. *FunctionChain* is a function chain application from ServerlessBench [72], where each function receives incoming data and forwards it to the next function. It represents workflows with longer lengths. *WordCount* is a word frequency counting application developed using the MapReduce model, originating from vSwarm benchmark [11]. It represents workflows with high parallelism but sparse intermediate data. We split the Map and Reduce phases of the *WordCount* application into functions that can be executed in parallel, and orchestrated within the workflow. *ParallelSorting* is a workflow application that implements a classic parallel sorting algorithm. It represents workflows with high parallelism and dense intermediate data. These three applications rely on the same set of `as-libos` modules, namely mm, fdtab, stdio, time, and fatfs (Table 2), except that *FunctionChain* does not require fatfs since it does not involve reading input data from files.

**Comparison systems.** We compare AlloyStack with state-of-the-art systems, including (1) single-function runtimes: *Unikraft* [38], *gVisor* [8] and *Wasmer* [9]. *Unikraft* [38] is a modular LibOS that operates in kernel mode. We deploy *Unikraft* instances within the *Firecracker* [20]. *gVisor* [8] is a LibOS-based secure container. gVisor uses the ptrace platform by default. We use the OCI runtime runsc to evaluate the performance of *gVisor*. *Wasmer* [9] is a WASM runtime. (2) Rust-supported workflow runtimes: *OpenFaaS* [4] and *Faastlane* [37]. *OpenFaaS* [4] is a container based, fundamental serverless software stack. We deploy all components of *OpenFaaS*, including the gateway and functions, on the same machine to eliminate latency caused by cross-machine communication. *Faastlane* [37] is a thread-level execution environment for functions that leverages MPK for memory isolation. To avoid performance bottlenecks caused by Python's GIL, Faastlane forks a separate subprocess for each function during parallel execution phases and uses IPC for data transfer. We implement the main execution process of *Faastlane* using Rust for a fair comparison with AlloyStack. Given that Rust does not encounter the GIL issue, we have additionally implemented a version of Faastlane that exclusively utilizes reference passing, denoted by the suffix "-refer". MicroVMs offer better isolation due to their guest kernel and have been adopted by many commercial platforms. To ensure a fair comparison, we have integrated Faastlane with Kata Container [56] for deployment, indicated by the suffix "-kata". In this case, each workflow instance is still deployed within a single process, but the process runs inside a Firecracker [20] MicroVM rather than directly on the host OS. (3) WASM-supported workflow runtime: *Faasm* [12]. *Faasm* [61] is also a thread-level FaaS runtime, but functions must be compiled to WASM to be executed by it. In experiments, we use suffixes like "-C" and "-Py" to differentiate between C and Python programming languages. For example, "AlloyStack-C" refers to deploying C-based benchmarks on the AlloyStack platform.

**Metrics.** We use three metrics to characterize the performance of AlloyStack: (1) *Cold start latency* refers to the time from when the system receives the event triggering the workflow, until the user's workload begins to run. (2) *Intermediate data transfer latency* refers to the period from when Function *A* writes data until Function *B* reads it. (3) *End-to-end latency* refers to the duration from triggering a workflow until its complete execution.

### 8.2 Cold Start Latency

We first evaluate the cold start latency using the *no-ops* benchmark. To demonstrate the effectiveness of the on-demand loading of AlloyStack, we introduce '*AS-load-all*', which disables the on-demand loading feature. As both *Wasmer* and *Faastlane* support deploying multiple functions as threads within a single address space, we also measure the thread startup latency and label them as *Wasmer-T* and *Faastlane-T*.

**Cold start latency: AlloyStack can reduce startup latency to 1.3ms while ensuring kernel isolation.** Figure 10 shows the cold start latency of AlloyStack and comparison systems. Among all the systems, AlloyStack's cold start latency is 1.3ms, slightly slower than *Faastlane-T*. This is

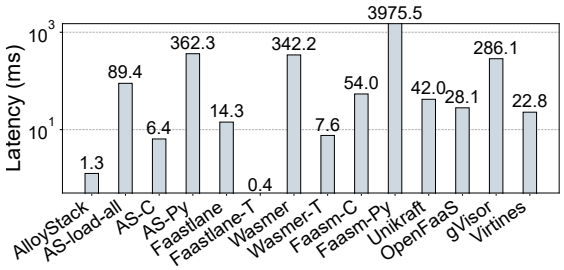


**Figure 10.** Comparison of cold start latency between AlloyStack (abbreviated as AS) and other platforms.

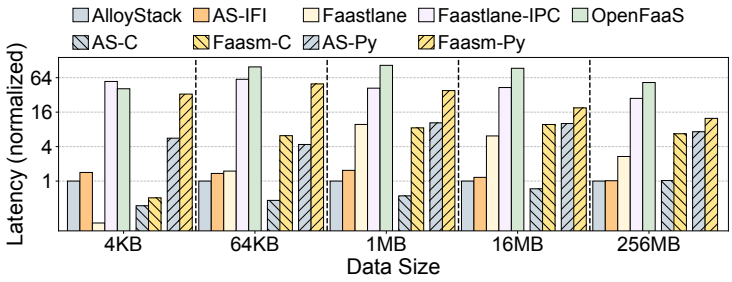


**Figure 11.** Comparison of intermediate data transfer latency. "AS-IFI" represents AlloyStack with inter-function isolation enabled.

because *Faastlane-T* does not support kernel isolation, eliminating the overhead of loading some dynamic libraries, resolving symbols, or initializing the isolation between user and system stacks. AS-load-all's cold start latency is 89.4 ms, indicating that loading LibOS components takes 88.1 ms. The experimental results from *Wasmer-T* and *Wasmer* are 7.6ms and 342ms, demonstrating that, although both are memory-safe languages, *Wasmer* incurs significant overhead compared to Rust due to its additional reliance on intermediate bytecode. *Virtines* also has a relatively low startup latency of 22.8ms, primarily because it does not require loading a guest kernel. However, it still uses KVM, incurring much overhead. Moreover, syscalls made by user functions deployed in these systems are processed directly by the host kernel, which may introduce security vulnerabilities [70]. The slow cold start of gVisor is due to two main factors: (1) Its initialization phase involves multiple syscalls interceptions and forwarding through *ptrace()* (our measurements show that approximately 50% of its runtime process CPU time is spent in kernel mode). (2) According to observations with perf [53], the overhead of the Go runtime and container-related OCI functionalities contribute to more than 20% of the total latency. Among all the systems, *Faasm-Py* and *AS-Py* have the slowest startup latency, primarily due to the high overhead of initializing the Python runtime.

### 8.3 Intermediate Data Transfer Latency

We then evaluate the intermediate data transfer latency by sending data from an upstream Function *A* to a downstream Function *B* using the *pipe* benchmark. For *Faastlane*, while it transmits intermediate data through reference passing by default, it switches to IPC mode when multiple functions need to run in parallel [37]. Therefore, in this experiment, we also evaluate the data transfer latency under IPC mode (denoted by *Faastlane-IPC*). For *OpenFaaS*, we use Redis to transfer the intermediate data.

**Intermediate data transfer latency: AlloyStack can reduce the data transfer latency by 1.8×-2.6×.** Figure 11 shows the data transfer latency under different data sizes. We measure the data transfer latency by printing timestamps before Function *A* sends the data and after Function *B* has completely received it. The experimental results demonstrate that AlloyStack achieves extremely high efficiency. In the APIs for Rust, C, and Python, AlloyStack takes 951 µs, 697 µs, and 9631 µs, respectively, to transfer 16MB of data. Compared to state-of-the-art systems, this represents performance improvements of 2.6×, 13.2×, and 1.8×, respectively. *OpenFaaS* has the highest latency in various experiments due to its "third-party forwarding" transfer method. *AS-C* has lower latency than AlloyStack, as higher compilation optimization options (e.g., -O3) are enabled when compiling the WASM image, resulting in better performance for C-based functions accessing intermediate data compared to Rust. When the data size is 4KB, *Faastlane* is faster than AlloyStack by 4 µs. This is because AlloyStack's `AsBuffer` is based on Rust smart pointers, leading to an additional constant time overhead for constructing smart pointers (approximately 4.4 µs). When the data size > 4KB, this overhead becomes negligible.

With inter-function isolation enabled ("AS-IFI"), AlloyStack's latency increases by 0.8%-33.7%. This is due to the additional overhead introduced by switching memory access permissions in user space through by updating the PKRU register. Compared to *Faastlane*, the slightly better performance of AlloyStack is attributed to its orchestrator, which typically schedules the sender and receiver functions in this test to the same CPU, thereby achieving better locality. AlloyStack utilizes trampoline mechanism to adjust a thread's memory access permissions, allowing the same thread to be safely shared between the `as-libos` and multiple functions. In contrast, *Faastlane* binds memory access permissions to thread IDs, requiring it to map upstream and downstream functions to different threads, which are typically scheduled on different CPU cores. However, as the data size increases (exceeding cache capacity) or when multiple functions run in parallel, the benefits brought by data locality become limited. *Faasm* exhibits higher data transfer latency due to page fault handling and the synchronization of global state. *Faasm* adopts a two-tier state architecture. It utilizes *mremap*

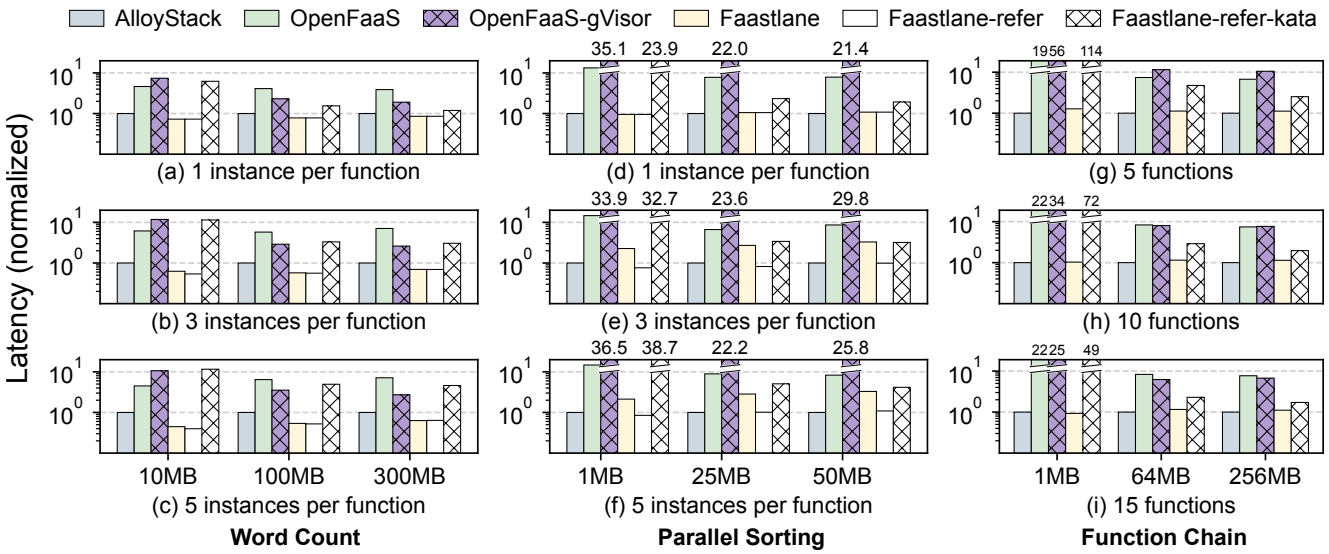


**Figure 12.** Comparison of end-to-end latency for Rust workflows across different platforms. "x instances per function" represents the number of concurrently executing function instances. "x functions" represents the length of the *FunctionChain*. The horizontal axis represents the sizes of the input data sets.

**Table 4.** Performance changes caused by the `as-libos` file system and network stack.

| | Modules | Read / RX | Write / TX |
|---|---|---|---|
| File System MB/s | rust-fatfs | 362 | 1562 |
| | Linux ext4 | 1351 | 1282 |
| TCP Gbit/s | smoltcp | 1.751 | 5.366 |
| | Linux | 27.76 | 28.56 |

to avoid serialization within the same worker. Nevertheless, this approach still incurs page faults when accessing data. When the sender and receiver functions are executed in different worker instances, the global state must be fetched from a distributed Redis instance, resulting in even higher overhead.

### 8.4 Rust Benchmark Evaluation

In this section, we compare AlloyStack with Rust-supported workflow runtimes (i.e., *OpenFaaS* and *Faastlane*) to evaluate the end-to-end latency of real-world applications implemented in the Rust language. Since *Faasm* supports WASM but not Rust, we do not include *Faasm* in the comparison in this section. For *OpenFaaS*, we further introduce its *gVisor* version, i.e., *OpenFaaS-gVisor*, which replaces containers with *gVisor*.

**End-to-End Latency: AlloyStack can reduce end-to-end latency by 1.2×-114×, under the condition of ensuring kernel isolation**. For the *WordCount* benchmark (Figure 12(a-c)), we set the input data size to 10MB, 100MB, and 300MB, and configure the number of instances for each *map* and *reduce* function to 1, 3, and 5, respectively. We find that *Faastlane*'s end-to-end latency is slightly shorter than AlloyStack. This is primarily due to the poor performance of the rust-fatfs file system used by AlloyStack. As shown in Table 4, rust-fatfs is 4.4× slower than the Linux ext4 file system under the read operation. Therefore, this performance degradation is unrelated to AlloyStack's design and can be resolved by replacing it with a high-performance user-space file system and network stack. As we were unable to find a high-performance user-space file system for AlloyStack, and designing a new one is beyond the scope of this paper, we leave this work for future efforts. In addition, we found that as the input data size increased from 10MB to 300MB, AlloyStack's advantage over *OpenFaaS-gVisor* diminished. This is mainly because the amount of intermediate data transferred in *WordCount* is smaller than the input data size, causing the high latency of rust-fatfs to offset the time saved from accelerated intermediate data transfer.

For *ParallelSorting* benchmark (Figure 12(d-f)), we set the input data size to 1MB, 25MB, and 50MB, and configure the number of instances for each function to 1, 3, and 5, respectively. Similar to the *WordCount* benchmark, when the number of function instances is 1, AlloyStack is slightly slower than *Faastlane* due to the slow rust-fatfs. When the number of function instances exceeds 1, as the amount of intermediate data increases, AlloyStack saves more time in intermediate data transfer, resulting in shorter end-to-end latency compared to *Faastlane*. When *Faastlane* uses reference passing (*Faastlane-refer*), its performance becomes comparable to that of AlloyStack. However, when isolation is enhanced using *Kata Containers*, the cold start overhead introduced by MicroVMs can result in end-to-end latency

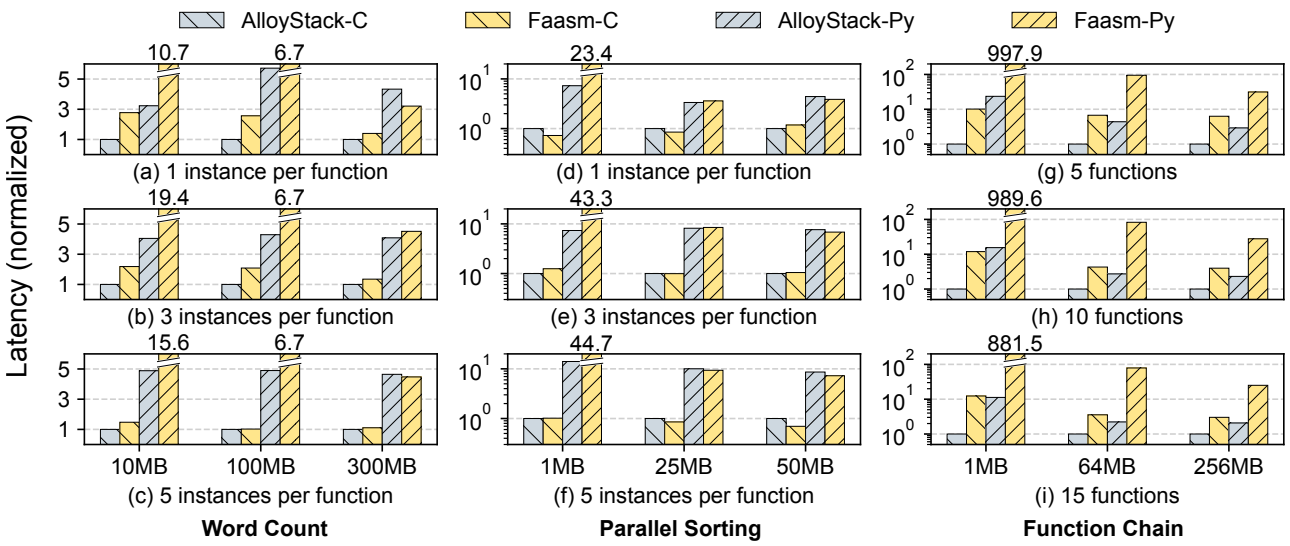


**Figure 13.** Comparison of end-to-end latency for C and Python workflows across different platforms. "x instances per function" represents the number of concurrently executing function instances. "x functions" represents the length of *FunctionChain*. The horizontal axis represents the sizes of input data sets.

up to 38.7× higher than that of AlloyStack. In summary, AlloyStack reduces end-to-end latency by 2.1~3.29 × compared to *Faastlane*, and by 6.5~29.3 × compared to *OpenFaaS* and *OpenFaaS-gVisor*.

For *FunctionChain* benchmark (Figure 12(g-i)), we set the intermediate data size to 1MB, 64MB, and 256MB, and configure the length of the function chain to be 5, 10, and 15, respectively. During the execution of *FunctionChain*, *Faastlane* does not use IPC for data transfer, as it is a workflow composed of multiple functions executed sequentially. Therefore, in this experiment, *Faastlane* and *Faastlane-refer* are identical. AlloyStack reduces the end-to-end latency by 4.08~10.15 × and 6.69~20.68× compared with *OpenFaaS* and *OpenFaaS-gVisor*. Due to better utilization of same-CPU data locality during data transfer, AlloyStack slightly outperforms *Faastlane*.

### 8.5 C and Python Benchmark Evaluation

In this section, we rewrite all benchmarks using C and Python, then compare AlloyStack with WASM-supported workflow runtimes (i.e., *Faasm*). We do not evaluate *Faastlane* in this section for fairness, since *Faastlane* does not support WASM.

For running C-based benchmarks (Figure 13), AlloyStack has a shorter end-to-end latency compared to *Faasm* in all configurations of *WordCount* and *FunctionChain*. Specifically, AlloyStack achieves a speedup of 1.02× to 2.77× in *WordCount*, and 3.01× to 12.41× in *FunctionChain* compared with *Faasm*. However, contrary to the trend observed in the Rust benchmark, in the case of *ParallelSorting*, AlloyStack's end-to-end latency is slightly longer than *Faasm*. This is primarily because Wasmtime [5], used by AlloyStack, is much slower than WAVM [10], which is used by *Faasm*. Our experimental results show that Wasmtime is 30.0% slower than WAVM. According to [22] and [69], this performance gap is due to the different code generators employed by the runtimes: WAVM utilizes the more mature and performance-oriented LLVM, whereas Wasmtime relies on Cranelift.

For running Python-based benchmarks, AlloyStack outperforms *Faasm* by up to 78.3× in terms of the end-to-end latency (Figure 13). Specifically, AlloyStack achieves a speedup of 1.11× to 4.79× in *WordCount*, 1.03× to 5.88× in *ParallelSorting* and 10.76× to 78.36× in *FunctionChain* compared with *Faasm*. For *WordCount* and *ParallelSorting*, the highest speedup ratios are achieved with input data sizes of 10MB and 1MB across three instances. This is due to the fact that, at these sizes, the data transfer latency, startup latency, and control plane overhead for AlloyStack are significantly lower than those for *Faasm*. However, as the data volume increases and the computational load becomes more significant, the benefits of AlloyStack diminish, leading to an average slowdown of 9.7% compared to *Faasm* at the largest input size. As the number of instances increases, the file reading process during the initialization of the Python runtime becomes a bottleneck, resulting in only a 1.74× average speedup compared to *Faasm* with 5 instances. For *FunctionChain*, as the function length increases, *Faasm* spends more time on the control plane. Consequently, AlloyStack significantly outperforms *Faasm* overall, achieving a maximum speedup of 78.4 × with 1MB data with 15 functions. As the data size increases, this overhead becomes amortized, leading to a reduction in the speedup ratio. Nevertheless, due to faster data transfer in AlloyStack, speedups of 10.8-12.1× are still achieved at 256MB.

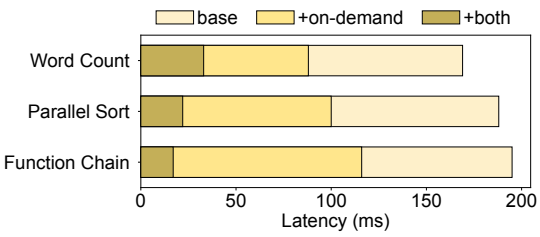


**Figure 14.** Contributions of techniques to reducing end-to-end latency. "base" refers to disabling both on-demand loading and reference passing, while "+both" indicates enabling both features.

### 8.6 Breakdown Analysis

To evaluate the individual impact of the on-demand loading and reference passing optimization techniques on end-to-end latency, we conducted comparative experiments by alternately enabling and disabling each technique based on *WordCount* (input data size: 10MB, number of function instances per stage: 5), *ParallelSorting* (input data size: 1MB, number of function instances per stage: 5), and *FunctionChain* (intermediate data size: 1MB, length: 15). When reference passing is disabled, AlloyStack uses files as an intermediary mechanism for data transfer. The intermediate data is first written to disk via fatfs and then copied to memory when the receiver reads it. This corresponds to the data transfer approach recommended by AWS Step Functions [24]. Figure 14 shows that on-demand loading reduces latency by 40.2% to 48.0%, while reference passing decreases it by 34.7% to 51.0%. Since the upper bound of `as-libos` cold start overhead ("load-all", § 8.2) remains nearly constant, the relative benefit of on-demand loading decreases as the end-to-end latency of the workflow increases or as the number of as-libos modules required by the workflow increases.

To further evaluate the reasons for the performance differences between AlloyStack and comparison systems, we divide the function execution process into three stages: *reading input*, *computation*, and *data transfer*, and record the time for each stage. Figure 15 describes the results for *WordCount* (with input size of 100MB, 3 instances), *ParallelSorting* (with input size of 25MB, 3 instances) and *FunctionChain* (with data transfer size of 64MB, 10 functions).

For Rust benchmarks, AlloyStack's *reading input* phase is significantly slower than that of *Faastlane* and *Faasm*, resulting in a decline in AlloyStack's end-to-end performance. Specifically, when reading 100MB data, AlloyStack's *reading input* latency is 6.9×-8.1× higher than *Faastlane*, due to the slow rust-fatfs as in Table 4. In the case of *ParallelSorting* (Figure 12(b)), since the *reading input* phase accounts for a small proportion of the overall end-to-end latency, the acceleration of intermediate data transfer in AlloyStack compensates for the slow file reading. For the *FunctionChain* application, the workflow does not involve reading input

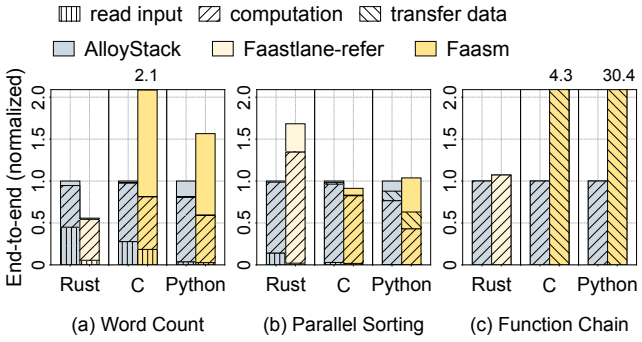


**Figure 15.** End-to-end latency breakdown. Input sizes for the three applications are 100MB, 25MB and 64MB.

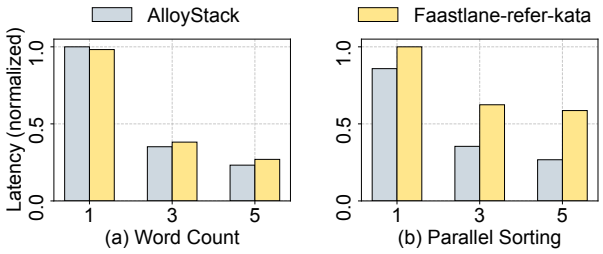


**Figure 16.** Comparison of end-to-end latency when running on ramfs. The x-axis represents the number of function instances in each parallel stage of the DAG.

data from files. Additionally, both AlloyStack and *Faastlane* use reference passing to transfer data. As a result, the read input and transfer data phases in the Rust version of *FunctionChain* are negligible. The slight performance advantage of AlloyStack over *Faastlane* is similar to the explanation in Section 8.3, primarily due to better data locality.

In the case of C and Python benchmarks, AlloyStack is slower than *Faasm* and *Faastlane* in the *computation* phase. Although both AlloyStack and *Faasm* support the execution of user functions through AOT-compiled WASM, the performance of Wasmtime used by AlloyStack is lower than that of WAVM used by *Faasm*, resulting in AlloyStack's computational efficiency being 1.4× lower than *Faasm*. The no-hatching pattern area in the Figure 15 corresponds to the synchronization waiting time during the fan-in phase. This waiting time occurs because the scheduling latency of the control plane and function cold starts cause variations in execution time across functions within each parallel stage, leading to functions having to wait for slower instances to complete before the fan-in can proceed.

In previous experiments, AlloyStack and *Faastlane* use different file systems, making performance comparison complex. Therefore, using an in-memory file system can eliminate this difference. For AlloyStack, we use the ramfs integrated within `as-libos`. *Faastlane* does not natively support ramfs, but Kata Container can provide it. We deploy *Faastlane* in a Kata Container, referred to as *Faastlane-refer-kata*,

to ensure a fair comparison. In Figure 16, after eliminating differences in *read input* and *transfer data*, AlloyStack still outperforms the comparison system slightly. This is because the hardware virtualization relied upon by MicroVMs can reduce computation efficiency, for example, by increasing the overhead of page fault handling [65].

## 9 Discussion

**Distributed/multi-node Setting.** According to traces from Azure Durable Functions [48], 95% of workflows have fewer than 8 stages. In most cases, a single WFD on one machine is sufficient to run multiple functions within the same workflow. When a single machine cannot handle the entire workflow, the serverless workflow DAG must be divided into multiple subgraphs [42], with each subgraph running in a WFD on separate nodes. Currently, AlloyStack does not automatically support such separation. Developers can manually divide the DAG and run the workflow using traditional intermediate data transfer methods.

**Resource Allocation and Elasticity**. Existing serverless systems typically require users to pre-determine the specifications of function instances (e.g., CPU and memory). Although not currently enabled, AlloyStack can also implement resource allocation based on user specifications, such as limiting the CPU bandwidth of function threads through cgroups. By default, AlloyStack requests the total resources required for all functions to ensure single-node deployment. To handle scenarios where concurrent invocations exceed the number of the corresponding function instance, AlloyStack can dynamically allocate additional resources to a WFD by creating Linux threads and establishing new function memory mappings (for example, using *dlmopen()*). This capability enables function-level dynamic scaling. If the orchestrator finds insufficient resources on the current node, the workflow will be split using the multi-node deployment mechanism.

## 10 Related Work

Our related work focuses on two aspects of serverless workflow applications: optimizing cold start latency and optimizing intermediate data transfer.

**Optimizing cold start latency**. Unikernels use specialization to reduce the initialization scope. Existing unikernels provide application compatibility at two levels: source code and binary. Systems such as [38, 46, 52] require a build system to generate unikernels from source code, similar to AlloyStack. FlexOS [40] enhances the fine-grained, configurable isolation mechanisms based on Unikraft. Additionally, solutions such as Lupine [39] offer binary compatibility. To the best of our knowledge, no existing unikernel supports on-demand loading like AlloyStack. Another approach is to use snapshots, such as [28, 66, 67], which can reduce the initialization time of secure containers and applications. However, snapshot-based approaches incur additional resource overhead to store and distribute snapshot templates. In contrast, the on-demand loading in AlloyStack reduces memory usage and optimizes cold start time.

**Optimizing intermediate data transfer latency**. Like AlloyStack, some existing systems also employ sharing within the same address space. PiP [30] utilizes *dlmopen()* to privatize variables across different tasks. However, in public cloud environments, this isolation approach may be circumvented by directly accessing the memory segments of other tasks. Singularity [31] enforces the use of ownership transfer rather than sharing to achieve efficient data transfer and isolation within a single address space. RedLeaf [51] leverages Rust's language features to make this approach more efficient and reliable. However, this introduces restrictions on the types of data that can be passed. Moreover, such isolation relying on memory-safe languages depends on a trusted compilation environment, which is not particularly practical for cloud providers. AlloyStack uses MPK hardware for isolation, eliminating these restrictions.

Additionally, several works enhance the efficiency of data transfer between serverless functions and third-party storage through caching mechanisms [50, 57]. SONIC [47] transparently selects the most efficient data-passing strategy for each edge of a serverless workflow DAG. In contrast, AlloyStack's reference passing technique does not rely on external storage or caching, resulting in lower latency and reduced storage overhead.

## 11 Conclusion

This paper introduces AlloyStack, a library operating system designed to optimize serverless workflow applications by addressing key performance challenges, including cold start latency and intermediate data transfer latency. By implementing on-demand loading of OS modules and reference passing within a single address space, AlloyStack significantly enhances execution efficiency. Our comprehensive evaluation shows that AlloyStack substantially reduces cold start latencies and the overhead of chained function calls, leading to shorter end-to-end execution times. AlloyStack achieves a speedup of 7.3× to 38.7× in Rust end-to-end latency and 4.8× to 78.3× in end-to-end latency in other languages compared to SOTA systems for intermediate-data-intensive workflows.

## 12 Acknowledgments

We thank the anonymous reviewers and our shepherd, Renaud Lachaize, for their insightful comments and suggestions that greatly improved this paper. This work is supported by the National Key Research and Development Program of China (No.2022YFB4500702); project ZR2022LZH018 supported by the Shandong Provincial Natural Science Foundation; and the National Natural Science Foundation of China under grant 62372322.

## References


[1] [n. d.]. Docker hub. https://hub.docker.com/ Referenced November 2024.

[2] 2016. a smol TCP/IP stack. https://github.com/smoltcp-rs/smoltcp Referenced April 2024.

[3] 2016. linked_list_allocator Github webpage. https://github.com/rust-osdev/linked-list-allocator Referenced May 2024.

[4] 2016. OpenFaaS: Serverless Functions, Made Simple. https://www.openfaas.com/ Referenced February 2024.

[5] 2017. A fast and secure runtime for WebAssembly. https://github.com/bytecodealliance/wasmtime Referenced November 2024.

[6] 2017. A FAT filesystem library implemented in Rust. https://github.com/rafalh/rust-fatfs Referenced April 2024.

[7] 2017. The Python programming language. https://github.com/python/cpython Referenced November 2024.

[8] 2018. Google Gvisor Github webpage. https://github.com/google/gvisor Referenced March 2024.

[9] 2018. The leading Wasm Runtime supporting WASIX, WASI and Emscripten. https://github.com/wasmerio/wasmer Referenced April 2024.

[10] 2019. WebAssembly Virtual Machine. https://github.com/WAVM/WAVM Referenced November 2024.

[11] 2021. A suite of representative serverless cloud-agnostic (i.e., dockerized) benchmarks. https://github.com/vhive-serverless/vSwarm Referenced May 2024.

[12] 2022. Faasm documentation. https://faasm.readthedocs.io/en/latest/source/cpp.html#writing-functions Referenced May 2024.

[13] 2023. Rust Unikernel OS. https://github.com/syswonder/ruxos.git Referenced January 2025.

[14] 2024. AWS Step Functions: Visual workflows for modern applications. https://aws.amazon.com/step-functions/ Referenced November 2024.

[15] 2024. Durable Functions is an extension of Azure Functions that lets you write stateful functions in a serverless compute environment. https://docs.microsoft.com/en-us/azure/azure-functions/durable/ Referenced November 2024.

[16] 2024. Dyninst. https://github.com/dyninst/dyninst Referenced October 2024.

[17] 2024. E9tool. https://github.com/GJDuck/e9patch/blob/master/doc/e9tool-user-guide.md Referenced October 2024.

[18] 2024. Objdump. https://sourceware.org/binutils/docs/binutils/objdump.html Referenced October 2024.

[19] 2024. Serverless Workflow: Visualization, O&M-free orchestration, and Coordination of Stateful Application Scenarios. https://www.alibabacloud.com/product/serverless-workflow Referenced November 2024.

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

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

[22] Bytecode Alliance. 2023. Documentation of Cranelift Code Generator. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/README.md. Referenced January 2025.

[23] Bytecode Alliance. 2024. Documentation of Wasmtime. https://github.com/bytecodealliance/wasmtime/blob/main/docs/stability-platform-support.md. Referenced January 2025.

[24] Amazon Web Services. 2025. Best practices for Step Functions. https://docs.aws.amazon.com/step-functions/latest/dg/sfn-best-practices.html#avoid-exec-failures Referenced February 2025.

[25] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. 2019. Leveraging Rust types for modular specification and verification. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[26] Marek Baranowski, Shaobo He, and Zvonimir Rakamarić. 2018. Verifying Rust Programs with SMACK. In *Automated Technology for Verification and Analysis*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 528–535.

[27] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. 2023. FMI: Fast and Cheap Message Passing for Serverless Functions. In *Proceedings of the 37th ACM International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 373–385. https://doi.org/10.1145/3577193.3593718

[28] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 467–481.

[29] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L Abad, and Alexandru Iosup. 2021. The state of serverless applications: Collection, characterization, and community consensus. *IEEE Transactions on Software Engineering* 48, 10 (2021), 4152–4166.

[30] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. 2018. Process-in-process: techniques for practical address-space sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona) *(HPDC '18)*. Association for Computing Machinery, New York, NY, USA, 131–143. https://doi.org/10.1145/3208040.3208045

[31] Galen C. Hunt and James R. Larus. 2007. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (April 2007), 37–49. https://doi.org/10.1145/1243418.1243424

[32] Intel. 2018. Intel-64 and IA-32 architectures software developer's manual. Referenced February 2024.

[33] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 691–707. https://doi.org/10.1145/3477132.3483541

[34] 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

[35] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

[36] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[37] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[38] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft:

fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. https://doi.org/10.1145/3447786.3456248

[39] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. 2020. A Linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 11, 15 pages. https://doi.org/10.1145/3342195.3387526

[40] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: towards flexible OS isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 467–482. https://doi.org/10.1145/3503222.3507759

[41] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 53–68. https://www.usenix.org/conference/atc22/presentation/li-zijun-rund

[42] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. FaaSFlow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 782–796. https://doi.org/10.1145/3503222.3507717

[43] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. 2020. Meltdown: Reading kernel memory from user space. *Commun. ACM* 63, 6 (2020), 46–56.

[44] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. 2023. Doing More with Less: Orchestrating Serverless Applications without an Orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1505–1519. https://www.usenix.org/conference/nsdi23/presentation/liu-david

[45] Guowei Liu, Laiping Zhao, Yiming Li, Zhaolin Duan, Sheng Chen, Yitao Hu, Zhiyuan Su, and Wenyu Qu. 2024. FUYAO: DPU-enabled Direct Data Transfer for Serverless Computing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 431–447. https://doi.org/10.1145/3620666.3651327

[46] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. (2013), 461–472. https://doi.org/10.1145/2451116.2451167

[47] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[48] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[49] Firecracker Maintainers. 2023. Documentation of Firecracker. https://github.com/firecracker-microvm/firecracker/blob/main/docs/prod-host-setup.md. Referenced January 2025.

[50] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 228–244. https://doi.org/10.1145/3447786.3456239

[51] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 21–39. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[52] Pierre Olivier, Hugo Lefeuvre, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2022. A Syscall-Level Binary-Compatible Unikernel. *IEEE Trans. Comput.* 71, 9 (2022), 2116–2127. https://doi.org/10.1109/TC.2021.3122896

[53] Linux perf wiki Contributors. 2025. *perf: Linux profiling with performance counters*. https://perf.wiki.kernel.org/ Referenced January 2025.

[54] Sheng Qi, Xuanzhe Liu, and Xin Jin. 2023. Halfmoon: Log-Optimal Fault-Tolerant Stateful Serverless Computing. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 314–330. https://doi.org/10.1145/3600006.3613154

[55] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: extracting the server from serverless computing! high-performance eBPF-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 780–794. https://doi.org/10.1145/3544216.3544259

[56] Alessandro Randazzo and Ilenia Tinnirello. 2019. Kata Containers: An Emerging Architecture for Enabling MEC Services in Fast and Secure Way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. 209–214. https://doi.org/10.1109/IOTSMS48152.2019.8939164

[57] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 122–137. https://doi.org/10.1145/3472883.3486974

[58] Vasily A. Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. 2022. CAP-VMs: Capability-Based Isolation and Sharing in the Cloud. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 597–612. https://www.usenix.org/conference/osdi22/presentation/sartakov

[59] Amazon Web Services. 2023. Amazon Simple Storage Service (Amazon S3). https://aws.amazon.com/s3/ Referenced May 2024.

[60] 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

[61] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433.

https://www.usenix.org/conference/atc20/presentation/shillaker
[62] John Toman, Stuart Pernsteiner, and Emina Torlak. 2015. Crust: a bounded verifier for rust (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 75–80.
[63] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572.
[64] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner
[65] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. 2011. Selective hardware/software memory virtualization. *SIGPLAN Not.* 46, 7 (March 2011), 217–226. https://doi.org/10.1145/2007477.1952710
[66] Nicholas C. Wanninger, Joshua J. Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C. Hale. 2022. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 644–662. https://doi.org/10.1145/3492321.3519553
[67] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 497–517. https://www.usenix.org/conference/osdi23/presentation/wei-rdma
[68] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. 2018. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*. 199–211.
[69] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-patch compilation: a fast compilation algorithm for high-level languages and bytecode. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 136 (Oct. 2021), 30 pages. https://doi.org/10.1145/3485513
[70] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, and Kui Ren. 2021. Demons in the Shared Kernel: Abstract Resource Attacks Against OS-level Virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 764–778. https://doi.org/10.1145/3460120.3484744
[71] Yanan Yang, Laiping Zhao, Yiming Li, Shihao Wu, Yuechan Hao, Yuchi Ma, and Keqiu Li. 2024. Flame: A Centralized Cache Controller for Serverless Computing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 153–168. https://doi.org/10.1145/3623278.3624769
[72] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 30–44. https://doi.org/10.1145/3419111.3421280

# A Appendix

## A.1 Additional Experiments

In this section, we evaluate the tail latency performance and resource consumption of AlloyStack.

**P99 Latency : AlloyStack can reduce the P99 latency by up to 7.4×.** To evaluate the impact of AlloyStack on tail latency, we select the *ParallelSorting* benchmark (with an input data size of 25MB and 3 function instances) and deploy it on both AlloyStack and the *Faastlane-refer-kata* systems. Compared to AlloyStack, the default *OpenFaaS* and *Faastlane* do not provide a guest kernel, meaning the measured latency in their systems does not include the overhead of guest kernel initialization. Therefore, to ensure fairness, we do not include them in this comparison. Figure 17(a) shows that the P99 latency of *Faastlane-refer-kata* significantly increases with the rise in QPS, which is attributed to the performance bottlenecks of Rootfs storage and the host kernel's Cgroup under high concurrency conditions [41]. When the QPS reaches 160, the P99 latency of AlloyStack increases sharply due to the CPU resource constraints on the physical machine.

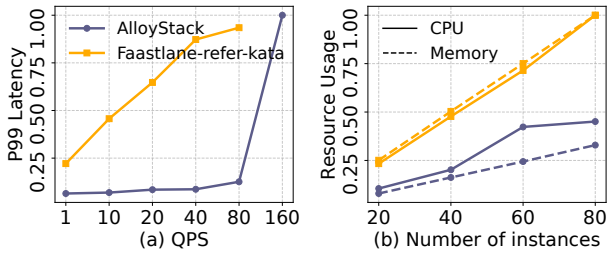


**Figure 17.** Comparison of tail latency (P99) and CPU/memory resource utilization. All data are normalized.

**CPU and Memory Usage: AlloyStack can reduce CPU usage by 2.4× and memory usage by 3.2× compared to *Faastlane-refer-kata*.** We further measure the CPU and memory usage under different numbers of workflow instances, using the *ParallelSorting* benchmark (with the input data size to 25MB and the number of function instances per stage to 5). Similarly, we do not evaluate *OpenFaaS* and *Faastlane* in this experiment because they do not provide guest kernel. Figure 17(b) shows that AlloyStack reduces resource usage by 40.8% to 68.4%. The reduction in resource usage of AlloyStack is primarily due to two reasons: (1) The LibOS can avoid approximately 90% of CPU time in kernel mode and context switching overhead. (2) On-demand loading prevents CPU time and memory from being consumed by unnecessary kernel modules.