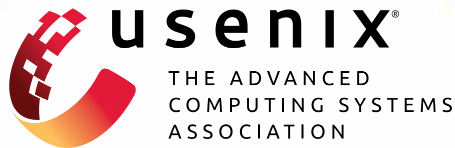


Faastlane: Accelerating Function-as-a-Service Workflows

Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu,
Indian Institute of Science

https://www.usenix.org/conference/atc21/presentation/kotni

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.
July 14–16, 2021
978-1-939133-23-6

Open access to the Proceedings of the
2021 USENIX Annual Technical Conference
is sponsored by USENIX.

Faastlane: Accelerating Function-as-a-Service Workflows

Swaroop Kotni,[*] Ajay Nayak, Vinod Ganapathy, Arkaprava Basu
Department of Computer Science and Automation
Indian Institute of Science, Bangalore

Abstract

In FaaS workflows, a set of functions implement application logic by interacting and exchanging data among themselves. Contemporary FaaS platforms execute each function of a workflow in separate containers. When functions in a workflow interact, the resulting latency slows execution.

Faastlane minimizes function interaction latency by striving to execute functions of a workflow as threads within a single process of a container instance, which eases data sharing via simple load/store instructions. For FaaS workflows that operate on sensitive data, Faastlane provides lightweight thread-level isolation domains using Intel Memory Protection Keys (MPK). While threads ease sharing, implementations of languages such as Python and Node.js (widely used in FaaS applications) disallow concurrent execution of threads. Faastlane dynamically identifies opportunities for parallelism in FaaS workflows and fork processes (instead of threads) or spawns new container instances to concurrently execute parallel functions of a workflow. We implemented Faastlane atop Apache OpenWhisk and show that it accelerates workflow instances by up to 15×, and reduces function interaction latency by up to 99.95% compared to OpenWhisk.

1 Introduction

Function-as-a-Service (FaaS) is emerging as a preferred cloud-based programming paradigm due to its simplicity, client-friendly cost model, and automatic scaling. The unit of computation on a FaaS platform is a developer-provided function. Contemporary FaaS applications typically comprise a set of functions expressed as a *workflow*. A workflow is a directed acyclic graph that specifies the order in which a set of functions must process the input to the application. When an external request such as a web request or a trigger (*e.g.,* timer) arrives for an application, *an instance* of its workflow takes life. AWS Step Functions (ASF) [9], IBM Action Sequences [28], and OpenWhisk Composers [43] enable developers to create and execute such workflows.

[*]Author is currently affiliated with Microsoft Research India. This work was performed when the author was at the Indian Institute of Science.

FaaS shifts the responsibility of managing compute resources from the developer to the cloud provider. The cloud provider charges the developer (*i.e.,* cloud client) only for the resources (*e.g.,* execution time) used to execute functions in the application (workflow). Scaling is automatic for the developer—as the workload (*i.e.,* number of requests) increases, the provider spawns more instances of the workflow.

In contemporary FaaS offerings, each function, even those that belong to the *same workflow instance*, is executed on a separate container. This setup is ill-suited for many FaaS applications (*e.g.,* image- or text-processing) in which a workflow consists of multiple interacting functions. A key performance bottleneck is *function interaction latency*—the latency of copying *transient* state (*e.g.,* partially-processed images) across functions within a workflow instance. The problem is exacerbated when FaaS platforms limit the size of the directly communicable state across functions. For example, ASF limits the size of arguments that can be passed across functions to 32KB [35]. However, many applications (*e.g.,* image processing) may need to share larger objects [2]. They are forced to pass state across functions of a workflow instance via cloud storage services (*e.g.,* Amazon S3), which typically takes hundreds of milliseconds. Consequently, the function interaction latency could account upto 95% of the execution time of a workflow instance on ASF and OpenWhisk (Figure 6).

Prior works [2, 11, 31, 52] have proposed reducing function interaction latency, but they are far from optimal. First, the proposed software layers to communicate state still incur significant, avoidable overhead. Second, they introduce programming and/or state management complexity. As examples, SAND orchestrates via global message queues for remote communication [2], while Crucial introduces a new programming API for managing distributed shared objects [11].

We observe that if functions of the *same workflow instance* were to execute on threads of a single process in a container, then functions could communicate amongst themselves through load/stores to the shared virtual address space of the encompassing process. The use of load/stores minimizes function interaction latency by avoiding any additional software

overhead and exposes the simplest form of communication possible—no new API is needed. The underlying hardware cache-coherence ensures strong consistency of the shared state at low overhead, without requiring elaborate software management of data consistency, unlike prior works [11, 52].

Faastlane strives to execute functions of a workflow instance on separate threads of a process to minimize function interaction latency. This choice is well-suited in light of a recent study on Azure Functions, which observed that 95% of FaaS workflows consisted of ten or fewer functions [50]. This observation limits the number of threads instantiated in a process. Faastlane retains the auto-scaling benefits of FaaS. Different instances of a workflow spawned in response for each trigger still run on separate containers, possibly on different machines.

While threaded execution of functions simplify state sharing within FaaS workflows, it introduces two new challenges:
① **Isolated execution for sensitive data.** Several FaaS use-cases process sensitive data. In such cases, unrestricted sharing of data even within a *workflow instance* leads to privacy concerns. For example, a workflow that runs analytics on health records may contain functions that preprocess the records (*e.g.,* by adding differentially-private noise [19]), and subsequent functions run analytics queries on those records. Workflows are often composed using off-the-shelf functions (*e.g.,* from the AWS Lambda Application Repository [3] or Azure Serverless Community Library [40]). In the example above, raw health records are accessible to a trusted preprocessor function. However, they should not be available to untrusted analytics functions from a library.

Unfortunately, threads share an address space, eschewing the isolated execution of functions within a workflow instance. Thus Faastlane enables lightweight thread-level isolation of sensitive data by leveraging Intel Memory Protection Keys (MPK) [29].[1] MPK allows a group of virtual memory pages (*i.e.,* parts of process address space) to be assigned a specific protection key. Threads can have different protection keys and, thus, different access rights to the same region of the process's address space. The hardware then efficiently enforces the access-rights. Faastlane uses MPK to ensure that functions in a workflow instance, executing on separate threads, have different access rights to different parts of the address space. Simultaneously, it enables efficient sharing of non-sensitive data by placing it in pages shared across the functions.
② **Concurrent function execution.** Some workflow structures allow many functions within an instance to be executed in parallel. However, most FaaS applications (*e.g.,* 94%) are written in interpreted languages, like Python and Node.js [48], whose popular runtimes disallow concurrent execution of threads by acquiring a global interpreter lock (GIL) [12, 16].

Threaded-execution thus prevents concurrent execution of functions in a workflow instance even if the workflow and the underlying hardware admit parallelism. Faastlane addresses this problem via its *adaptive workflow composer*, which analyzes the workflow's structure to identify opportunities for concurrent execution, and forks processes (within the same container) to execute parallel portions of the workflow. Specifically, Faastlane uses threads when the workflow structure dictates sequential execution of functions and processes when there is parallelism. Functions running on separate processes use pipes provided in the multiprocessing module of Python for sharing state. Python pipes internally use shared memory communication. Given that processors with 64-80 cores are commercially available ([46, 47]), while those with 128 cores are on the horizon [45], we expect that containers can be configured with enough vCPUs to allow most workflows to execute entirely within a single container instance. Thus, our key objective is to enable FaaS workflows to leverage efficient communication within a single server.

However, it is also possible for some workflow structures to allow hundreds and thousands of parallel functions. A single container may not have enough vCPUs to fully leverage the available parallelism within a workflow instance. Faastlane's adaptive workflow composer thus judges if the benefit from leveraging parallelism in function execution likely to outweigh the benefits of reduced function interaction latency due to execution of all functions of a workflow instance within one container. If so, the composer falls back on the traditional method of launching multiple containers, each of which would run the number of parallel functions that it can execute concurrently. Functions of a workflow instance executing in separate containers communicate state over the network as happens on contemporary FaaS platforms.

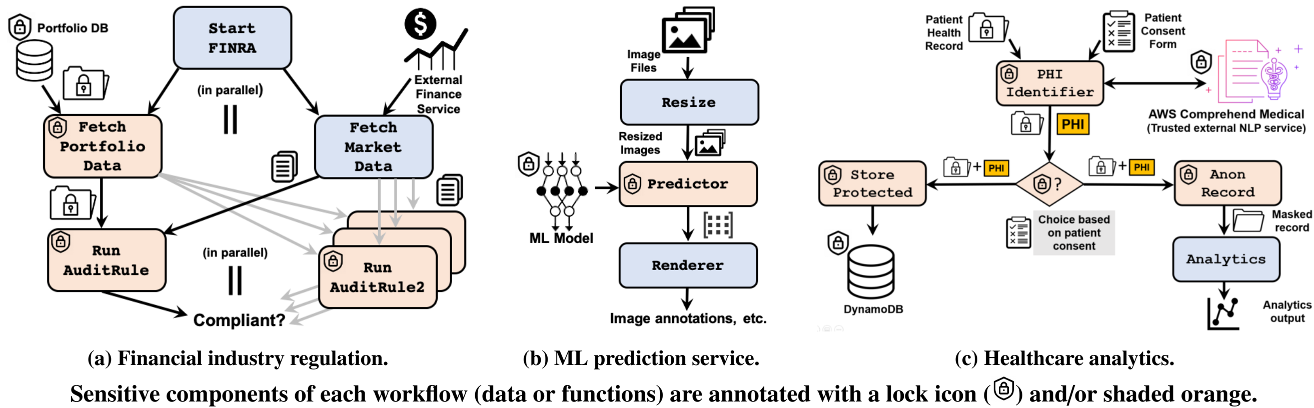
Designing for efficient function interactions also helped us significantly reduce the dollar-cost of executing FaaS applications. On FaaS platforms such as ASF, a developer is charged for ① the cost of executing functions, ② the cost of transitioning between nodes of a workflow across containers, and ③ the cost of storage services for transferring large transient states. Faastlane reduces the first two components and eliminates the last one.

To summarize, Faastlane makes the following advances:

- It reduces function interaction latency over OpenWhisk by up to 2307× by executing functions on threads. Consequently, it speeds up a set of example FaaS applications by up to 15×.
- It provides lightweight thread-level isolation using Intel MPK to support applications that process sensitive data.
- It leverages parallelism in workflows by adapting to execute functions as processes, when appropriate, to avoid serialization by GILs in popular FaaS language runtimes.
- Further, if the underlying container cannot fully leverage the available parallelism in a workflow instance, Faastlane falls back on using multiple containers as appropriate.

The source code for Faastlane and the applications are available at https://github.com/csl-iisc/faastlane.

[1] ARM [7] and IBM processors [27] also have MPK-like page-grouping support in their processors. AMD has announced MPK-like feature [6].



(a) Financial industry regulation. **(b) ML prediction service.** **(c) Healthcare analytics.**

Sensitive components of each workflow (data or functions) are annotated with a lock icon ( ) and/or shaded orange.

Figure 1: Examples of commercially-important FaaS workflows referenced in Section 2.

2 Function Interaction and State Isolation

We present three commercially-important FaaS workflows that illustrate the importance of minimizing function interaction latency and isolated execution of functions within a workflow instance. We also identify three key design patterns that suffice to express a wide variety of FaaS workflows.

2.1 Financial Industry Regulation

The US Financial Industry Regulatory Authority (FINRA) provides oversight on the operation of broker-dealers to detect malpractices [21]. FINRA requires every broker-dealer to periodically provide it an electronic record of its trades performed during that period. It then validates these trades against market data for about 200 pre-determined rules that check them against a variety of compliance criteria [20].

On average, FINRA validates about half a trillion trades daily [20]. However, this computation only needs to run for some time during the day, and the trade volume to be validated fluctuates daily. The pricing and auto-scaling models of FaaS make FINRA validation an ideal candidate for this platform.

Consider a FaaS workflow to validate a trade against an audit rule (Figure 1a). `StartFINRA` triggers this application's workflow by invoking two functions. `FetchPortfolioData`, which is invoked on each hedge-fund's trading portfolio, takes a portfolio ID as input and fetches highly-sensitive trade data, private to each hedge-fund. `FetchMarketData` fetches publicly-available market data based on portfolio type. This function accesses external services to access stock market trade data (*e.g.,* Nasdaq). These functions can run concurrently in a given workflow instance.

The analyzer function (`RunAuditRule`) takes the sensitive output of `FetchPortfolioData` and the non-sensitive information obtained by `FetchMarketData` and validates the trades in each portfolio against the rules. Typically, the size of portfolio data (*i.e.,* the state) shared between the functions runs in KBs, while the market data can be several MBs. Although we have discussed only one checker, FINRA checks compliance against about 200 rules [20]. The checker for each such rule (*e.g.,* `RunAuditRule2`, *etc.*) can execute in *parallel* with the other checkers, and each must obtain the outputs of `FetchPortfolioData` and `FetchMarketData`.

To protect sensitive portfolio data from accidental disclosure, `FetchPortfolioData` and `RunAuditRule` simply perform local computations, and do not access external sources. It is also important to ensure that `FetchMarketData` cannot access sensitive data of the other two functions, even though all three functions are part of the same workflow instance of the same application for the same external request.

Current FaaS platforms isolate functions by executing each function in a different container. However, the functions in a workflow may be data-dependent and need to share several MBs of state. Heavy-handed isolation via containers drives up the cost of sharing state and contributes to a significant component of the total workflow execution time (Section 5).

2.2 ML Prediction Service

ML prediction is a computationally-intensive task that takes input data (*e.g.,* images or text), extract features, and provides domain-specific predictions (*e.g.,* detecting objects or predicting sentiments) [17, 34, 52]. Real-world prediction services have real-time response requirements [24]. The workload of an ML prediction service can vary over time and, therefore, can benefit from auto-scaling as provided by FaaS.

Figure 1b presents an example workflow of an ML prediction service for image classification. The workflow has three data-dependent functions, and must, therefore, execute in *sequence*. `Resize` pre-processes an input image and passes the resized image to the main prediction function. `Predictor` consults an ML model and outputs a vector of probabilities that is then post-processed by `Renderer` and rendered suitably *e.g.,* by identifying object boundaries, or suitably labeling objects in an image. The raw image is shared between `Resize` and `Predictor`, and can be 100s-1000s of KB in size, while the output of `Predictor` is in the order of 100s of bytes.

In a typical ML prediction service, the pre-trained ML model is usually the secret sauce, created using sensitive train-

ing data sets and/or at considerable expense. The service provider may not wish to release the model. However, other parts of the workflow may use functions developed by third-parties (*e.g.,* [3, 40]). Here, it is crucial to ensure that `Resize` and `Renderer` do not have access to the state of `Predictor`, which loads the ML model in memory to performs its task.

2.3 Healthcare Analytics

Hospitals generate various health records that can be analyzed to create valuable information for population health management. Applications that process these health records need to meet strict compliance rules (*e.g.,* HIPAA) regarding sensitive data in medical records, including patient details and doctors' notes, which we refer to as protected health information (PHI). Healthcare analytics applications are also a good fit for FaaS pricing and scaling models; AWS Lambda hosts many such applications [41]. Figure 1c depicts an example of healthcare analytics workflow. `PHIIdentifier` takes two inputs: a patient's health record, which may contain unstructured text or images, and a consent form. Using the consent form, the patient can choose to: ① store the medical record securely on a cloud-based service, *e.g.,* DynamoDB in Figure 1c; and ② offer to make available an anonymized version of their medical record to analytics functions downstream. The workflow conditionally executes one or both of the subsequent steps (`StoreProtected` and `AnonRecord`) depending on the patient's consent—we call this design pattern a *choice*.

Both these subsequent steps require identifying PHI data in the unstructured text of the medical record. The `StoreProtected` function segregates PHI data from the medical record, and stores both the PHI data and the stripped medical record securely on a cloud server; the two parts can be combined later when the patient submits a request to retrieve the record. `AnonRecord` anonymizes the medical record by masking out the PHI data identified in the record.

The `PHIIdentifier` function securely communicates with a trusted, external service to identify PHI data in the medical record, *e.g.,* a HIPAA-compliant NLP service like AWS Comprehend Medical [5] or a medical image analyzer like Amazon Rekognition [22]. The medical record and the PHI data are sensitive, and the untrusted `Analytics` function must not access them. The internal state of `PHIIdentifier`, `StoreProtected`, `AnonRecord` must also be protected from `Analytics`. As in the other two examples, this workflow also requires communication of a significant amount of state (*e.g.,* medical records containing images) between functions.

2.4 Function Interaction Patterns in FaaS Workflows

The three workflows discussed thus far illustrate three design patterns for function interaction in FaaS workflows:

- The *parallel* pattern as seen in FINRA, in which functions that are not data-dependent can execute concurrently;
- The *sequence* pattern as seen in the ML prediction service, in which data-dependent functions must execute in sequence;
- The *choice* pattern as seen in Healthcare analytics, in which functions in the workflow are conditionally executed based on the user's input or the output of a function in the workflow.

With these design patterns, we can succinctly encode all constructs of Amazon States Language (ASL) [4, 8]. ASL is the format in which application developers specify workflows for ASF. Thus, it suffices for Faastlane to incorporate support for these three design patterns to express most of the workflows available on contemporary FaaS platforms.

2.5 Threat Model

As illustrated in our examples, functions in a FaaS workflow produce/consume sensitive and non-sensitive data. Even functions that belong to the same workflow instance of a FaaS application should not have unfettered access to each other's state. It is not uncommon for a single FaaS application to use untrusted functions from public repositories (*e.g.,* [3, 40]) along with trusted functions and services that can access sensitive data. As we saw, functions do communicate a substantial amount of state, and the use of any heavy-handed mechanisms for isolation, *e.g.,* executing function within separate containers, inevitably leads to performance bottlenecks.

Our goal is to enable efficient sharing of state across functions in a workflow with just the right amount of isolation. We assume that the cloud provider offers an MPK-based hardware platform and the operating system (OS) used to run FaaS processes includes standard, widely-deployed data-execution prevention features (*e.g.,* W⊕X). Additionally, Faastlane does not change the OS features that aid in defending against microarchitectural side-channel attacks. Best practices to defend against such attacks, such as those developed in Firecracker [1], can also be adapted for use with Faastlane. We implemented Faastlane's mechanisms as extensions to the language runtime (*i.e.,* Python interpreter). We, therefore, assume that the runtime is part of the trusted computing base. Note that we strive to isolate sensitive data and *not* code. Thus, we cannot guard the confidentiality of proprietary functions in FaaS workflows.

3 Design Considerations

Faastlane's design has three objectives. First, we aim to *minimize function interaction latency without sacrificing concurrency in parallel workflows*. It leverages fast communication within a single server (machine) to share state across functions of a single workflow instance, wherever possible. Loads and stores to a shared address space are the lowest-latency options for sharing data. Faastlane thus strives to map functions in a workflow instance to threads sharing a process address space. While threads serve as a vehicle of concurrency in most settings, they do not in this case. Interpreters for popular languages used to write FaaS applications, such as Python and Node.js, use a global interpreter lock (GIL) that prevents concurrent execution of application threads. Faastlane thus has a workflow composer, a static tool that analyzes the workflow

structure, and forks processes instead of threads wherever the workflow allows parallelism. These processes run within the same container and communicate via Python pipes.

The workflow composer is also cognizant of the fact that a single container may not have enough vCPUs to run all functions concurrently (as processes) for workflows that admit massive parallelism. If all functions of an instance of such workflows are packed onto a single container, the performance loss from the lack of concurrency may outweigh the benefits of reducing function interaction latency. Therefore, the composer periodically (*e.g.,* once a day) profiles containers on the FaaS platform to ascertain the available parallelism. Wherever the composer encounters large parallelism in the workflow structure, it determines whether it would be beneficial to deploy instances of that workflow across multiple containers based on measurements from profiling. Each container itself concurrently runs multiple functions as processes, commensurate to that container's vCPU allocation.

Second, Faastlane aims to *control the sharing of data within a workflow instance* when functions of a workflow instance run as threads. As motivated in Section 2, we discovered important workflows wherein sensitive data must be isolated and shared only with authorized functions. While there is a large literature on in-process isolation techniques [13, 15, 26, 36], recent work [25, 54] has shown that a hardware feature available on Intel server processors, called MPK, offers thread-granularity isolation at low overheads. Faastlane uses Intel MPK to provide thread-granularity memory isolation for FaaS functions that share a virtual address space.

Finally, Faastlane aims to meet the above two objectives *without needing FaaS application writers to modify their functions* and without requiring them to provide more information (than they already do) to FaaS platforms. It is also transparent to the cloud provider, except that Faastlane is most useful when the underlying hardware supports MPK or MPK-like features. Faastlane achieves this goal by designing a static client-side tool, the workflow composer, along with a profiler to achieve its objective of minimizing function interaction latency without sacrificing concurrency. A modified language runtime (here, Python), and the composer, ensure data isolation. Faastlane can be packaged in an enhanced container image. No new API is exposed to FaaS applications, and no additional information is required from developers. This differentiates Faastlane from many prior works (*e.g.,* [11]) that introduce new FaaS programming abstractions (*e.g.,* to specify data sharing). We demonstrate Faastlane's adoptability by deploying it atop an unmodified OpenWhisk-based FaaS platform on a server equipped with MPK.

4 Implementation of Faastlane

Faastlane needs to accomplish two primary tasks. First, it must spawn threads and/or processes and/or multiple containers to execute functions of a workflow instance for minimizing function interaction latency without sacrificing concurrency that exists in the workflow structure. Second, when executing functions as threads, it must ensure the isolation of sensitive state across the functions. Faastlane's workflow composer, aided by a simple profiler, accomplishes the first task. A modified language runtime and the composer accomplish the second.

```
1 def FetchPortfolioData(portfolioName, portfolioType):
2      portfolioData = fetchPrivateData(portfolioName)
3      validateDataFormat(portfolioData)
4      return portfolioData

1 def FetchMarketData(portfolioName, portfolioType):
2      # Accesses a third-party service
3      marketData = yahooFinanceAPI(portfolioType)
4      return marketData

1 def AuditRule([portfolioData, marketData]):
2     # Sample Regulatory rule
3     ruleSatisifed = applyRegulation(
4                          portfolioData, marketData)
5     return ruleSatisifed
```

Figure 2: Functions from the FINRA workflow (Figure 1a).

```
1  from FetchPortfolioData import FetchPortfolioData
2  from FetchMarketData import FetchMarketData
3  from AuditRule import AuditRule
4
5  marketData = {}; portfolioData = {}; ruleOut = {}
6
7  def AuditRule_Wrapper(portfolioData, marketData):
8      #Fetch output object(s) from global scope
9      global ruleOut
10     ruleOut = AuditRule([portfolioData, marketData])
11     memset_input(marketData) # zero-out the inputs
12     memset_input(portfolioData)
13
14 def FetchMarketData_Process(portfolio, queue):
15     response = FetchMarketData(portfolio.name,
16                                portfolio.type)
17     queue.put(response)
18
19 def FetchPortfolioData_Process(portfolio, queue):
20     response = FetchPortfolioData(portfolio.name)
21     queue.put(response)
22
23 def PStateWrapper(portfolio):
24     P1 = Process(target = FetchPortfolioData_Process,
25                  args = [portfolio, queue1])
26     P2 = Process(target = FetchMarketData_Process,
27                  args = [portfolio, queue2])
28     #Execute processes in parallel
29     P1.start(); P2.start()
30     P1.join(); P2.join()
31     #Fetch output object(s) from global scope
32     global marketData, portfolioData
33     #Update output object(s) with queue responses
34     portfolioData = queue1.get()
35     marketData = queue2.get()
36     memset_input(portfolio) # zero-out the inputs.
37
38 def Orchestrator(portfolio):
39     T1 = Thread(target = PStateWrapper,
40                 args = [portfolio])
41     T2 = Thread(target = AuditRule_Wrapper,
42                 args = [portfolioData, marketData])
43     T1.start(); T1.join()
44     T2.start(); T2.join()
45     return ruleOut
```

Figure 3: Workflow composer output for functions in Figure 2.

4.1 Minimizing Function Interaction Latency

Current FaaS platforms take a workflow description (Figure 1a) and function definitions as inputs (Figure 2), as sepa-

rate entities. They create one container for each of the functions in each executing instance of a workflow. The platform uses the workflow description to suitably route function calls.

Faastlane does not demand any new information from the FaaS application developer or modifications to the code. Faastlane's workflow composer tailors the JSON workflow description and the function code for execution in Faastlane's runtime. The workflow composer first analyzes the workflow structure (JSON description) to identify if it allows for any concurrency in executing the functions of an instance. If the workflow is sequential, the composer packs all functions of the workflow within a unified function called `Orchestrator`. To schedulers on FaaS platforms, the unified workflow provides the *illusion* of an application with a single-function. The entire workflow is thus scheduled for execution in a single container. The workflow composer also creates *function wrappers*, which the `Orchestrator` invokes suitably to reflect the DAG specified in the workflow structure. The `Orchestrator` method's return value is the output of the FaaS application.

Function wrappers explicitly identify the input and output of the function. Wrappers have an associated built-in `start()` method that is implemented in Faastlane's runtime. Invoking a `start()` method spawns a new thread to execute the corresponding function (`join()` denotes the end of the thread). As will be described later in this section, wrappers also implement data isolation using MPK primitives. Each function wrapper begins by reading input from the (shared) process heap and places its output back on the heap. All the other data of a function, not explicitly specified in the wrapper's I/O interface, is considered private to that function and stored in that thread's heap partition (detailed shortly). The `Orchestrator` function specifies the order in which threads are invoked, thereby enforcing the workflow's data sharing policy. The functions of a workflow instance executing as threads minimizes function interaction latency by sharing data using load/stores to the shared process heap.

Language runtimes use a GIL that prevents threads from executing concurrently. This does not present any issues for sequential workflows but leads to loss of concurrency for workflows that contain parallelism. Faastlane launches functions of a workflow instance that can execute in parallel as separate processes instead of threads. Processes can run concurrently on vCPUs of the container deployed on the FaaS platform. These functions communicate via Python pipes.

For example, consider the functions `FetchMarketData` and `FetchPortfolioData` that can run in parallel. The workflow composer creates a parallel-execution wrapper (`PStateWrapper` in Figure 3) that identifies parts of the workflow that can execute concurrently (using the `Process` keyword). The subsequent `start()` method for `P1` and `P2` spawns new processes to execute those portions of the workflow. Each such forked process can itself be a sub-workflow and can further spawn threads or processes as required. The inputs to these functions need not be explicitly copied, as they are already available in the address space before the `start()` method forks a new process. The output is copied back to the parent process using Python pipes (`queue1` and `queue2`). The `join()` methods in `Orchestrator` serve as barriers that prevent the workflow from progressing unless the corresponding function wrapper thread has completed execution.

Using processes eschews some of the benefits of sharing transient states via loads/stores. Moreover, the language runtime has to be set up in each process, resulting in extra page faults. Typically, functions in FaaS are short-running (<1 sec) [50]). Thus, the overheads due to page faults are non-negligible for these ephemeral processes (detailed in Section 5.2). However, the loss of concurrency with only threads justifies the use of processes for portions of a workflow that admit parallelism.

A container may not have enough vCPUs to run all functions for a parallel section of a workflow concurrently. The number of vCPUs exposed to a container may be smaller than the number of cores in the underlying hardware. Further, FaaS platform providers may not even expose the number of vCPUs on the container it deploys to run a workflow instance. Therefore, Faastlane deploys a simple compute-bound micro-benchmark on the FaaS platform and observes its scalability to infer the number vCPUs in the container deployed by the platform. Such profiling is needed only very infrequently (*e.g.,* once a day) since the number of vCPUs in a container for a given class typically does not change [33].

On encountering a large, parallel workflow structure, the composer spawns multiple containers, each packed with the number of processes equal to the (inferred) number of vCPUs in the container. Networked communication across functions on different containers happens via a REST API [32].

To support cloud hardware that does not offer MPK (or MPK-like) support, the profiler checks for pku and ospke flags in /proc/cpuinfo/ to ascertain if the FaaS platform supports MPK. If not, the workflow composer uses only processes to execute functions. This fallback option sacrifices the gains in lowering function interaction latency that threads offer, but adheres to Faastlane's principle of being transparent to both the application writer and the cloud provider.

4.2 Isolation for Sensitive Data

The use of threads for efficient state sharing among functions in a workflow fundamentally conflicts with the goal of protecting sensitive data in situations such as those illustrated in Section 2. Faastlane leverages an existing hardware feature to enforce thread-granularity isolation of data. Note that data isolation is not a concern when functions execute on separate processes. Each process has its own isolated address space, and the state-less nature of FaaS means that they cannot communicate amongst themselves via the local file system, even when they run on the same container.

Faastlane's runtime ensures that each thread gets a private, isolated portion of address space. There is also a shared parti-

tion of the address space that the threads use to communicate the transient state via load/store. Isolation is ensured efficiently using MPK hardware. The memory allocator of the language runtime is extended to ensure that each thread allocates its memory needs from its private partition by default.

Memory protection keys. Intel MPK provides hardware support for each page in a process's address space to be annotated with a 4-bit protection key. Logically, these keys allow us to partition the virtual address space into 16 sets of pages. MPK uses a 32-bit register, called the PKRU register, to specify access rights for each set of pages. Each key is mapped to 2 bits in the PKRU register, that specify access rights of the currently-executing thread to pages in each set. On a memory access, permissions are checked against the PKRU register. Thus, using MPK, it is possible to specify read/write accesses for a set of pages that share the same protection key.

When a process starts, all pages in its address space are assigned a default protection key, and the PKRU value allows both read and write access to all pages. A process assigns a protection key to a page using a system call (`pkey_alloc` in Linux), and the protection key is written into the page-table entry. To specify access rights for pages with that protection key, the process uses the `WRPKRU` instruction, which allows the process to modify the value of the PKRU register from user-space. Since the `WRPKRU` instruction can be executed from user-space, it is important to ensure that it is only invoked from trusted code (called *thread gates*). Faastlane uses binary inspection (as in ERIM [54]) to verify the absence of `WRPKRU` instructions in locations other than thread gates. Faastlane performs this inspection before the workflow is deployed in the FaaS environment. Therefore, it does not incur any run-time performance overheads.

Thread Gates. A thread gate is a sequence of instructions that contains MPK-specific instructions to modify the PKRU register or to assign protection keys to pages. A thread gate is classified as an entry gate or an exit gate, based upon whether it executes at the beginning or end of a thread. Faastlane modifies the Python runtime to implement the instruction sequence corresponding to entry and exit gates in the builtin `start()` and `join()` methods.

An entry gate accomplishes three tasks. First, it attaches a protection key to the thread, using the `pkey_alloc` system call. Second, it communicates this key to the memory manager in Faastlane's Python runtime. The memory manager ensures that all subsequent memory requests are satisfied from pages tagged with the thread's protection key. Finally, the gate uses the `WRPKRU` instruction to write to the PKRU register to ensure that only the current thread has read and write access to pages tagged with that protection key. In effect, this establishes a heap partition accessible only to that thread.

When the thread completes, the exit gate frees the protection key for further use using the `pkey_free` system call. It also zeroes-out the memory region allocated to serve memory for that protection key. Such cleanup allows Faastlane to reuse protection keys for multiple threads without compromising the isolation guarantees. Without reuse, Faastlane's Python runtime would restrict the number of available protection domains to 16, thereby also restricting the number of functions in the workflow to 15.[2] One domain is reserved for the parent thread (`Orchestrator`). The shared heap is accessible to all threads and serves as the shared memory region to which all threads enjoy unfettered read/write access.

Thread Memory Management. Faastlane modifies Python's memory manager to map requests from different threads to different virtual address regions. Faastlane's modifications are packaged as a CPython module (tested for Python3.5). Faastlane modifies the default memory allocator to maintain separate arenas (contiguous regions of virtual address space; typically 256KB) for different threads and ensures that memory requests from one thread are always mapped to the thread's private arenas. After requesting an arena from `mmap`, Faastlane attaches a protection key to the arena using the `pkey_mprotect` system call, effectively reserving the arena for memory requests to the thread owning the protection key.

As discussed, the thread-entry gate associates a protection key for a thread executing a function in the workflow when it starts execution and maps all memory requests from the thread to that private arena. If the arena becomes full, it allocates another arena. When the wrapper finishes, it destroys all the arenas associated with that thread's protection key. The only exception is that of the main thread (`Orchestrator`), whose arenas are accessible to all threads in the process.

4.3 Putting It All Together

Figure 4 summarizes the lifecycle of a FaaS workflow in Faastlane. The lifecycle starts with the application developer supplying the functions and a workflow description connecting them. The client supplies these entities to Faastlane's workflow composer. The composer analyzes the workflow and produces a unified description using the design patterns discussed in Section 2.4, capturing the available parallelism in the workflow. Based on the available parallelism (if any) and the number of vCPUs in the container (determined via profiling), the workflow is deployed in the FaaS platform.

For workflows without or limited parallelism, the composer provides a single top-level function, called the `Orchestrator`, that encapsulates the entire workflow description. The client supplies this unified workflow description to the FaaS platform, which gives the platform the illusion of a single-function FaaS workflow. The platform schedules the entire FaaS application for execution within a single container. If the workflow contains parallelism that exceeds the estimated number of vCPUs in a container, the composer creates multiple `Orchestrator` functions. Each `Orchestrator` subsumes a sub-workflow with functions that can run concurrently within a single container. Each `Orchestrator` is scheduled on a dif-

[2] A vast majority of workflows contain fewer than 10 functions [50]. Previous work has also shown ways to virtualize MPK partitions [44].

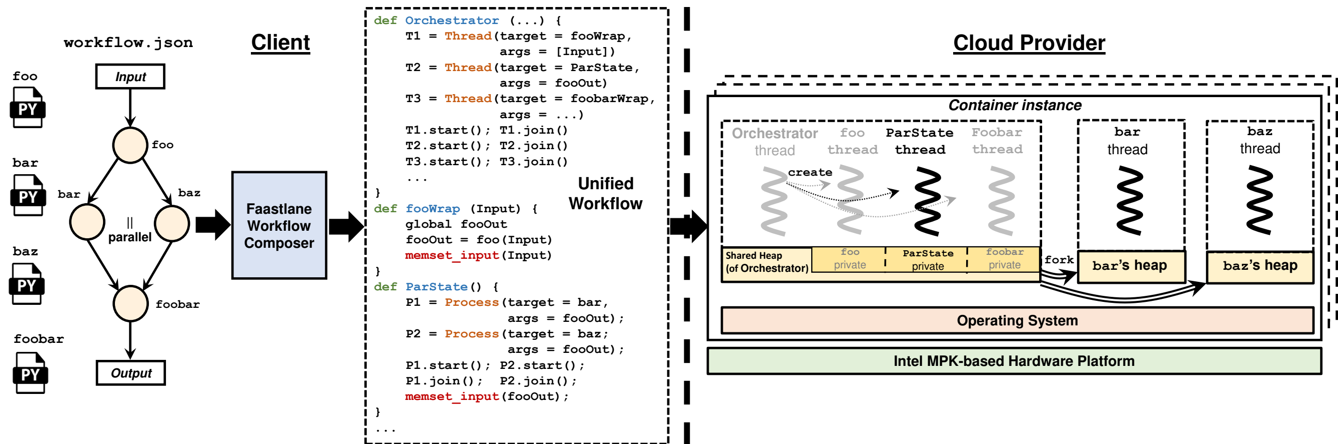


Figure 4: This picture shows a snapshot of the workflow when the functions bar and baz are executing concurrently. The threads shown in grey have either completed execution (foo), are currently paused (Orchestrator) or are yet to be created (foobar). ParState is the wrapper for the bar and baz functions in the parent process. Although not shown in this example, Faastlane allows child processes to be sub-workflows that can recursively spawn threads or processes. If the workflow is highly parallel, the workflow composer creates multiple top-level Orchestrator functions, each of which executes in a different container.

ferent container by the FaaS platform.

Faastlane's runtime component within the container accepts the unified workflow description and decomposes it to identify the functions within the workflow. Based on the Orchestrator, it either starts threads (one per function) or forks processes to execute functions of the workflow. When functions execute as threads, each thread executes instructions to identify a partition of the process heap accessible only to that thread (using MPK primitives). It uses this partition to store data private to the function. The functions running on the thread can share state via load/stores to a designated shared heap for the process. When a thread completes, the function's output is made available (via the Orchestrator method) only to the functions in the workflow that consume that output. When Faastlane forks a process, it passes only the output of the last method from the parent process to the Orchestrator method of the child process via a Python pipe. For large parallel workflows, functions in different containers communicate over the network stack as in contemporary FaaS platforms.

Faastlane does not impact auto-scaling in FaaS. Different instances of a given workflow spawned in response to a trigger are scheduled on different containers, as is typical.

5 Evaluation

We evaluate Faastlane against four real-world applications that broadly capture popular FaaS function interaction patterns [39]. Besides the applications in Section 2, we include Sentiment analysis, which is another real-world use case [49].

Sentiment analysis evaluates user reviews for different products of a company. Its workflow contains two choice states. The first choice state chooses an analysis model based on the product. The second choice state publishes to a database versus a message queue (for manual analysis of negative reviews) based on review sentiments. Together, these applications encapsulate all the patterns discussed in Section 2.4.

Processor	2 × Intel(R) Xeon(R) Gold 6140 CPU
No. of cores	36 (2×18)
DRAM	384 GB, 2666 MHz
LLC Cache	24 MB
Linux Kernel	v. 4.19.90
Docker	19.03.5, build 633a0ea838

Table 1: Experimental Setup.

Table 1 shows the configuration of our experimental system, including the software stack. We measure how Faastlane improves function interaction latency, end-to-end latency, throughput, and reduces dollar-cost. We perform the experiments on a single machine where Faastlane can pack all functions of a workflow instance within a single container. However, in Section 5.4 we further show how Faastlane scales up to use multiple containers in face of large parallelism within a workflow structure.

We compare Faastlane's performance against ASF, OpenWhisk and SAND [2]. SAND aims to reduce function interaction latency. It executes functions in a workflow as separate processes within a container and shares state using a hierarchical message queue. Our performance experiments with OpenWhisk, SAND and Faastlane were run on our local hardware platform, while ASF experiments were run on the AWS cloud infrastructure. We cannot control the AWS cloud hardware platform. Thus, the ASF measurements are not necessarily directly comparable to those of the other platforms.

5.1 Function Interaction Latency

Figure 5 reports the function interaction latency of the four applications under different FaaS platforms. The applications are as described in Section 2, with the only difference that the FINRA application checks portfolio data against 50 audit rules. We run each application at least 100 times and report the median. We observe that function interaction latency is

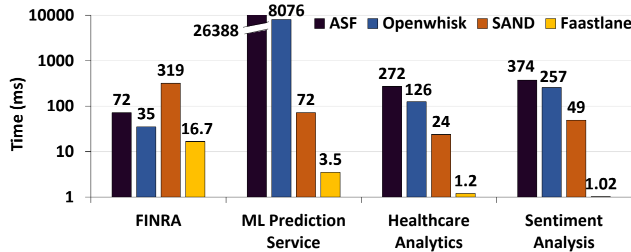


Figure 5: Median function interaction latency (in milliseconds)

lowest with Faastlane across all four applications, each with different function interaction patterns. Faastlane reduces the latency over its *closest* competitor by 52.3%, 95.1%, 95% and 98% for FINRA, ML prediction service, Healthcare analytics and Sentiment analysis, respectively.

SAND reduces the function interaction latency compared to current FaaS platforms, except for FINRA. FINRA runs 50 audit rules in parallel, where SAND's hierarchical messaging queues serialize communication from concurrently executing functions. Consequently, SAND increases the function interaction latency to 319ms from 35ms on OpenWhisk. However, Faastlane reduces it to 16.7ms. For ML prediction service, SAND reduces the interaction latency to 72ms from thousands of milliseconds on OpenWhisk while Faastlane almost eliminates it. Similarly, for the remaining apps, the interaction latency is a mere 1-1.2ms with Faastlane.

Note that the ML prediction service has very high function interaction latencies (in thousands of milliseconds) on both ASF and OpenWhisk. This is because the ML prediction service transmits the largest state across functions—an image of about 1.6MB size from the `Resize` function to the `Predictor` function (Figure 1b). However, on ASF and OpenWhisk, functions in a workflow can directly transmit a maximum state of size 32KB [35] and 1MB, respectively. Consequently, the ML prediction service is forced to fall back upon a slow storage service like Amazon S3 for state sharing on these platforms. The other applications transmits tens of KBs of state across functions in a workflow (for the payloads we used).

Though Faastlane offers the lowest function interaction latency for FINRA, the latency of 16.7ms is higher than the latency observed for the other applications even if it transmits only tens of KBs of state across functions. This behavior is because FINRA's workflow invokes 50 functions in parallel to audit the portfolios. Faastlane executes these functions as separate processes where the state is shared via Python pipes, instead of loads/stores within a process address space. While this helps end-to-end latency by leveraging concurrency (Section 5.2), the cost of sharing state increases compared to sharing state across threads of the same process.

5.2 End-to-End Application Performance

Latency. The end-to-end execution latency of an application's workflow is the time that elapses between the start of the first function of the workflow and the completion of the final function in the workflow. We measure end-to-end latency by executing each application at least 100 times and report both the median and tail (99%-ile) values.

We dissect the end-to-end latency as follows:

① The time spent in external service requests. An external service request is one that requires communication outside the FaaS platform. For example, the healthcare analytics application uses the AWS Comprehend Medical service to identify PHI in health records (Section 2.3). This component depends on several factors, including time over the network and the latency of processing the request at the external entity.

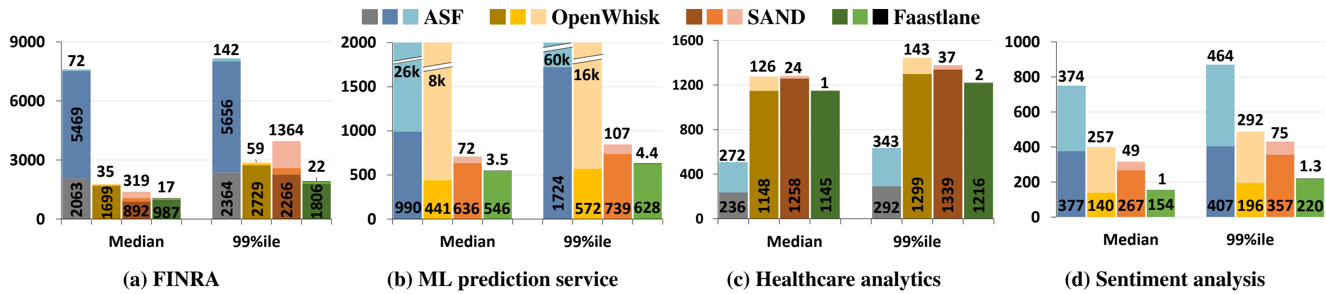
② The compute time, which is the time during which the workflow functions execute on the processor. It does not include the time spent on external service requests.

③ The function interaction latency on the FaaS platform.

Figure 6 presents the end-to-end latency broken down into the above components. For each application, we report median and 99%-ile latency on ASF, OpenWhisk, SAND, and Faastlane. The total height of each bar represents the end-to-end latency, and the stacks in the bar show the breakdown. In Figure 6a, we first note that the compute time is significantly more on ASF than on other platforms. While the FINRA application has 50 parallel functions, ASF never runs more than 10-12 of them concurrently. On other platforms, the compute time is much shorter because all the functions execute in parallel. Most of the time is spent on the external service request for retrieving the trade data. Overall, we find that Faastlane improves end-to-end latency by 40.5%, 23.7% over OpenWhisk and SAND, respectively.

In ML prediction service (Figure 6b), OpenWhisk's and ASF's end-to-end execution latency are dominated by the function interaction latency, as discussed earlier. In comparison, SAND significantly reduces function interaction latency, while Faastlane practically eliminates it. Consequently, even in comparison to SAND, Faastlane improves end-to-end latency by 22.3%. Over OpenWhisk, the improvement due to Faastlane is 15×. Note that the compute time on OpenWhisk is shorter (by about 100ms) than that on Faastlane and SAND, although we ran OpenWhisk, Faastlane, and SAND on the same hardware platform. This difference is because, on Faastlane and SAND, the ML prediction service application transmits data between functions using JSON (de)serialization methods, contributing to the compute time. In contrast, on OpenWhisk, the application is forced to use S3, and the application does not use JSON when communicating with S3.

In Healthcare analytics (Figure 6c), the time spent on external service request (AWS Comprehend Medical) is significant. The response latency from AWS Comprehend Medical depends on external factors and is orthogonal to our system design. We observed that the latency is much lower on ASF but similar on the FaaS platforms that were executed on our local hardware. In comparison to OpenWhisk and SAND, Faastlane improves end-to-end latency by 9.9% and 10.4%, respectively. Faastlane improves end-to-end latency of Sen-



(a) FINRA **(b) ML prediction service** **(c) Healthcare analytics** **(d) Sentiment analysis**

Figure 6: Application end-to-end latency (in ms). Legend shows external service time, compute time and function interaction latency for each platform in that order. Note that (b), (d) do not have external services and (c) has negligible compute time (2-4ms).

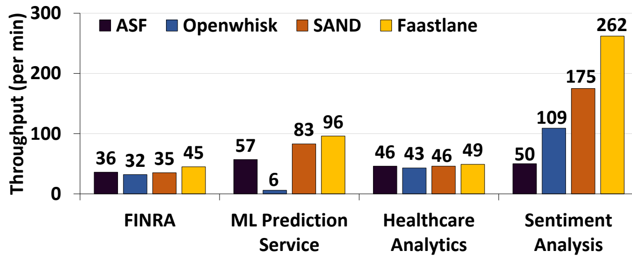


Figure 7: Measuring application throughput.

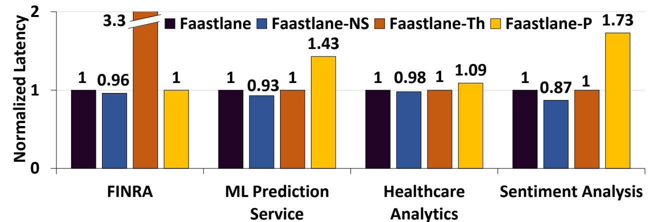


Figure 8: Analysis of the cost of isolation in Faastlane.

timent analysis (Figure 6d) by 61% over OpenWhisk and 51% over SAND. The tail (99%-ile) latency trends (Figure 6), mirror the above observations. In short, Faastlane provides the least end-to-end latency across platforms that are directly comparable, *i.e.,* run on the same hardware.

Throughput. Figure 7 shows the application throughput, measured as the number of application requests serviced per minute on different FaaS platforms. We use the time between the invocation of the first request and the response receipt for the last request. We repeat this experiment over several requests to measure throughput as the number of requests-per-minute. Unlike end-to-end execution latency, throughput measurement also accounts for initialization (*e.g.,* spawning containers), and post-processing time. Note that to compare throughput, we also need to ensure that all configurations run on the same hardware. The total number of cores on the hardware platform (and therefore, the available parallelism) has a *first-order* effect on throughput, unlike latency. While we present numbers from ASF for completeness, we forewarn the readers against directly comparing ASF's throughput numbers (run on hardware not in our control), with the numbers obtained on other FaaS platforms executed locally.

Figure 7 shows the throughput observed for each of our applications on ASF, OpenWhisk, SAND and Faastlane. In general, we observe that throughput follows the trends in the end-to-end latency. Faastlane provides the best throughput across all applications. For the FINRA application, Faastlane improves throughput by 28.6% over its nearest competitor (SAND). For ML prediction service and Sentiment analysis, Faastlane improves throughput by 15.6%, 49.7% over SAND and by 16×, 2.4× over OpenWhisk, respectively. For the healthcare analytics application, Faastlane provides a modest throughput improvement of 6% over SAND.

Cost of Isolation. For a deeper understanding of the design choices made in Faastlane, we create three variants of Faastlane. First, we turn off thread-level memory isolation in Faastlane by disabling its MPK-enabled memory manager and the zeroing-out of function's state in the shared domain (Faastlane-NS). The difference in performance between Faastlane-NS and Faastlane quantifies the performance cost of data isolation using MPK. Second, we constrain Faastlane to only use threads, and not a mix of processes and threads based on the workflow structure (Faastlane-Th). The performance difference between Faastlane-Th and Faastlane quantifies the usefulness of adapting between processes and threads. Third, we constrain Faastlane to use *only* processes, and share state via Python pipes (Faastlane-P). The performance difference between Faastlane-P and Faastlane quantifies the importance of employing threads when possible.

Figure 8 shows the normalized median end-to-end latency for Faastlane, Faastlane-NS, Faastlane-Th and Faastlane-P for all four applications. The height of each bar is normalized to latency under Faastlane. Comparing Faastlane-NS and Faastlane, we observe an increase of 1.9-14.9% in the end-to-end latency in Faastlane, which denotes the cost of MPK-based isolation mechanism. We argue that this performance cost is a reasonable price to pay for data isolation.

Next, we observe that the latency for Faastlane-Th is 3.3× of the Faastlane for FINRA. Recall that FINRA has 50 parallel functions in its workflow. Since the threads cannot execute concurrently in our Python implementation (due to the GIL), Faastlane-Th cannot execute these parallel functions concurrently. In contrast, Faastlane employs processes instead of threads and can leverage the underlying hardware resources for concurrency. For the rest of the applications, there is no difference between Faastlane-Th and Faastlane as they do not

Application	AWS Step Functions (ASF) Lambda	Step	Storage	Total	Faastlane Lambda
FINRA	31.17	1325	0	1356.17	10.2
ML Prediction Service	21.75	125	27.85	174.6	11.87
Healthcare Analytics	1.65	150	0	151.65	0.83
Sentiment Analysis	1.85	175	0	176.85	1.03

Table 2: Cost (in USD) per 1 million application requests

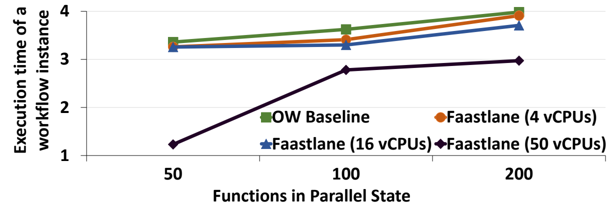


Figure 9: Scalability for FaaS Applications with parallelism.

contain any parallel functions. This experiment shows the importance of analyzing the workflow structure and using processes instead of threads in the presence of GIL.

Finally, the latency for Faastlane-P is 9-73% higher than Faastlane in applications except FINRA. Lifetimes of FaaS functions are short, *e.g.,* 50% of the functions execute in less than 1s [50]. Thus, processes running these functions in Faastlane-P are ephemeral and we found the page-fault latency in setting up language runtime in each process dominates their execution times. Faastlane reduces page-faults by 1.43-3.95× using threads over processes. For FINRA, Faastlane uses processes to leverage parallelism. Thus, there is no difference between Faastlane-P and Faastlane.

5.3 Dollar-cost of Application Execution

We now discuss the dollar-cost implications to developers for executing applications on Faastlane versus atop ASF. The cost of executing applications on ASF includes: ① *Lambda* costs, which is the cost of executing the functions of a workflow, ② *Step* costs, which is the cost of transitioning from one node to another in an ASF workflow across containers, and ③ *Storage* costs, for transferring large transient state [53]. We noticed that execution times of functions on ASF varies run-to-run. We used the smallest execution time to estimate the least cost of running an application on ASF.

Unlike with ASF, the dollar-cost of running on Faastlane *only* includes the *Lambda* cost. The *Step* cost of transitioning across containers is zero because Faastlane executes an entire workflow instance within a container. Faastlane does not need a storage service to transfer state and therefore incurs no *Storage* cost. To measure the *Lambda* cost on Faastlane, we deployed applications with Faastlane-enhanced virtual environments on AWS Lambda. We disabled MPK instructions because AWS machines may not have MPK-enabled today. However, we adjusted the billed-duration and maximum memory of each application with MPK overheads (1.9-14.9%). We exclude the cost of any external service that a function uses for its execution (*e.g.,* AWS Comprehend Medical).

Table 2 reports the total cost incurred for each application on ASF and Faastlane. Faastlane executes applications only at 0.6-6.8% of the cost of ASF. Faastlane eliminates expensive step costs and storage costs by designing for efficient function interactions. Despite running on containers with larger memory, Faastlane reduces Lambda costs, partly due to the reduction in runtime because of efficient function interactions and partly due to the current AWS Lambda pricing model.

5.4 Scalability with Multiple Containers

All experiments thus far were limited to one server, and the underlying container had sufficiently many vCPUs to run all parallel functions in applications like FINRA concurrently. We now explore how Faastlane would scale if the underlying container had a limited number of vCPUs and when multiple servers are to be deployed to cater to the parallelism available within a single workflow instance. For this purpose, we scaled up the number of functions in the parallel step in FINRA's workflow from 50 to 200. We also varied the number of vCPUs in each container from 4 to 50.

When Faastlane detects that the parallelism in the workflow exceeds the parallelism serviceable by the vCPUs of a single container, it spawns multiple containers, possibly scheduled on multiple servers. Each container is packed with a number of processes commensurate with the vCPUs alloted to it.

Figure 9 shows how Faastlane performs under such circumstances vis-a-vis OpenWhisk baseline that runs every function in separate containers. The y-axis reports the average execution time of a workflow instance as the number of parallel functions are scaled (x-axis). We observe that when the number of vCPUs per container is limited, Faastlane offers limited uplift (2.46%). This is expected since Faastlane falls back on network-based communication across containers when spawning multiple containers. As we increase the number of vCPUs per container, Faastlane launches fewer containers, and performs much better. At 50 vCPUs per container, Faastlane speeds up by 2.73× over the baseline when there are 50 parallel functions. Even when there are 200 functions, Faastlane reduces execution time by 25.3%. In short, Faastlane is most effective when it can leverage fast communication within a container, particularly for sequential workflows and those with limited parallelism. However, even when faced with large parallelism in workflows, coupled with limited parallelism in underlying containers, Faastlane scales at least as well as the current offerings.

6 Related Work

Table 3 compares Faastlane against closely related systems using three criteria—function interaction latency, ability to support unmodified FaaS workflows (*e.g.,* those written for commodity FaaS offerings such as OpenWhisk and ASF), and whether the solution supports function interaction across machines. Commercial FaaS solutions satisfy the latter two criteria, but have high function interaction latencies [10, 23].

	Function Interaction Latency	Unmodified Application	Interaction Across Machines
OpenWhisk/ASF	High	✓	✓
SAND [2]	Medium	✓	✓
SONIC [37]	Medium	✓	✓
Crucial [11]	Medium	✗	✓
Cloudburst [52]	Medium	✗	✓
Faasm [51]	Low	✗	✗
Nightcore [30]	Low	✗	✗
Faastlane	Low	✓	✓

Table 3: Comparing Faastlane with related work.

SAND [2] reduces function interaction latency via hierarchical message queues, while allowing unmodified FaaS applications and workflows spanning multiple machines. However, as our evaluation showed, Faastlane's approach of using memory load/store instructions reduces function interaction latency even further. SONIC [37] aims to reduce function interaction latency by jointly optimizing how data is shared (*e.g.,* by sharing files within a VM or copying files across VMs) and the placement of functions in a workflow. However, SONIC uses files as the core data-sharing mechanism, leaving room for latency reduction in several cases.

Crucial [11] improves state-sharing in highly parallelizable functions using distributed shared objects (DSOs). DSOs are implemented atop a modified Infinispan in-memory data grid. Unfortunately, Crucial's approach is not compatible with unmodified FaaS applications. Cloudburst [52] focuses on distributed consistency of shared data using Anna key-value stores [56]; other systems, like Pocket [31], use specialized distributed data stores for transient data. These systems improve function interaction latency, but leave significant headroom for improvement because of the overheads of accessing the remote data store.

While all the aforementioned systems support distributed FaaS applications, in which interacting functions execute on different machines, Faasm [51] and Nightcore [30] aim to reduce function interaction latency by executing workflows on a single machine. Faasm [51] uses threads and shared memory with software-fault isolation [55] (more heavy-weight) to provide thread-private memory partitions. Unfortunately, Faasm requires custom-written FaaS applications that are then compiled down to WebAssembly for execution. Nightcore [30] accelerates stateless microservices by co-locating them on the same physical machine and using shared memory to enable efficient data sharing. However, the FaaS applications must be modified to use Nightcore's libraries to avail of these benefits. Faastlane's function interaction latencies are comparable to Faasm and Nightcore, but it additionally supports unmodified FaaS workflows and distributed FaaS applications.

Aside from these systems that share Faastlane's goal of reducing function interaction latency in FaaS workflows, there is prior work on reducing the memory footprint and performance overhead of other aspects of FaaS. Chief among these are methods that aim to reduce the memory footprint and performance overheads of containers by creating stripped-down containers [42], checkpointing [18], language-level isolation [14], or create bare-bones VMs tailored to the application loaded in the VM [1, 38]. Faastlane's lightweight sandboxing approach can be used in conjunction with prior work to improve the overall performance of FaaS applications.

Faastlane uses MPK to offer lightweight memory isolation between threads when they are used in the workflow. Recent works introduce new OS-level primitives for efficient in-process isolation [13, 15, 26, 36]. However, the overheads are still substantial compared to MPK which offers user-level instructions to switch memory domains. In contrast, the other solutions add new system calls, or introduce non-negligible runtime overheads. Prior studies have aimed to provide intra-process isolation using MPK. We borrow the ideas of thread gates and binary inspection from ERIM [54]. While Faastlane creates in-process thread-private memory domains, Hodor [25] creates protected libraries using MPK-backed protection domains. MPK offers 16 domains, which is sufficient for most current FaaS applications [50]. For larger workflows, Faastlane reuses thread domains wherever possible (Section 4.2). Libmpk [44] offers a software abstraction of MPK to create a larger number of domains. However, it adds non-negligible software overheads.

7 Conclusion

Developers structure FaaS applications as workflows consisting of functions that interact by passing transient state between each other. Commercial FaaS platforms execute functions of a workflow instance in different containers. The transient state must be copied across these containers or transmitted via cloud-based storage services, both of which contribute to the latency of function interaction.

Faastlane reduces function interaction latency by sharing data via memory load/store instructions. To accomplish this goal, it strives to execute functions of a workflow instance as threads within a shared virtual address space. However, this approach raises new challenges associated with isolating sensitive data and availing the parallelism afforded by the underlying hardware. Faastlane's novel design uses Intel MPK for lightweight thread-level isolation and an adaptive mixture of threads and processes to leverage hardware parallelism. Our experiments show that Faastlane outperforms commercial FaaS offerings and also recent research systems designed to reduce function interaction latency.

Acknowledgments

We are grateful to the anonymous reviewers for their thoughtful feedback on the work. Thanks to Onur Mutlu for shepherding the paper. This work was supported in part by a Ramanujan Fellowship from the Government of India, a Young Investigator fellowship from Pratiksha Trust, and research gifts from Intel Labs and VMware Inc.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, February 2020.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *USENIX Annual Technical Conference*, 2018.

[3] Amazon. AWS Serverless Application Repository—Discover, deploy, and publish serverless applications. https://aws.amazon.com/serverless/serverlessrepo/.

[4] Amazon States Language—AWS Step Functions Developer Guide. https://docs.aws.amazon.com/step-functions/latest/dg/concepts-amazon-states-language.html.

[5] Amazon Comprehend Medical—extract information from unstructured medical text accurately and quickly. https://aws.amazon.com/comprehend/medical/.

[6] Memory Protection Keys on AMD processors. https://www.anandtech.com/show/16214/amd-zen-3-ryzen-deep-dive-review-5950x-5900x-5800x-and-5700x-tested/6.

[7] ARM. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, 2018.

[8] Amazon States Language Specifications. https://states-language.net/spec.html.

[9] AWS Step Functions—build distributed applications using visual workflows. https://aws.amazon.com/step-functions/.

[10] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. Faas orchestration of parallel workloads. In *Proceedings of the 5th International Workshop on Serverless Computing*, 2019.

[11] Daniel Barcelona-Pons, Marc Sanchez-Artigas, Gerard Paris, Pierre Sutra, and Pedro Garcia-Lopez. On the FaaS Track: Building stateful distributed applications with serverless architectures. In *ACM Middleware Conference*, 2019.

[12] David Beazly. Inside the Python GIL. In *Python Concurrency and Distributed Computing Workshop*, May 2009. http://www.dabeaz.com/python/GIL.pdf.

[13] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[14] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[15] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *IEEE Symposium on Security and Privacy*, 2016.

[16] Python–Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock.

[17] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *USENIX Symposium on Networked Systems Design and Implementation*, 2017.

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

[19] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[20] FINRA adopts AWS to perform 500 billion validation checks daily. https://aws.amazon.com/solutions/case-studies/finra-data-validation/.

[21] United States Financial Industry Regulatory Authority. https://www.finra.org/.

[22] Sarah Gabelman. How to use Amazon Rekognition and Amazon Comprehend Medical to get the most out of medical imaging data in research, September 2019. https://aws.amazon.com/blogs/apn/how-to-use-amazon-rekognition-and-amazon-comprehend-medical-to-get-the-most-out-of-medical-imaging-data-in-research/.

[23] Pedro Garcia-Lopez, Marc Sanchez-Artigas, Gerar Paris, Daniel Barcelona-Pons, Alvaro Ruiz Ollobarren, and David Arroyo Pinto. Comparison of FaaS Orchestration Systems. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018.

[24] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dymtro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yanqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiadong Wang. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture*, 2018.

[25] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, July 2019.

[26] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2016.

[27] IBM. Power ISA^{TM} version 3.0 b, 2017.

[28] IBM Cloud Functions–Creating Sequences. https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-sequences.

[29] Intel. Intel-64 and IA-32 architectures software developer's manual, 2018.

[30] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[31] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[32] API Gateway for AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway.html.

[33] Container Configuration in AWS Lambda. https://docs.amazonaws.cn/en_us/lambda/latest/dg/configuration-memory.html.

[34] Yunseong Lee, Alberto Scolari, Byoung-Gon Chun, Marco D. Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine-learning prediction serving systems. In *USENIX Symposium on Operating Systems Design and Implementation*, 2018.

[35] AWS Step Function limits. https://docs.aws.amazon.com/step-functions/latest/dg/limits.html.

[36] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2016.

[37] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware data passing for chained serverless applications. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[38] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[39] Microsoft. Azure durable popular application patterns. https://docs.microsoft.com/en-in/azure/azure-functions/durable/durable-functions-overview?tabs=csharp#application-patterns.

[40] Microsoft. Azure serverless community library. https://serverlesslibrary.net/.

[41] Chris Munns. Powering HIPAA-compliant workloads using AWS Serverless technologies. In *AWS Compute Blog*, July 2018. https://aws.amazon.com/blogs/compute/powering-hipaa-compliant-workloads-using-aws-serverless-technologies/.

[42] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[43] Apache OpenWhisk Composer—A high-level programming model in JavaScript for composing serverless functions. https://github.com/apache/openwhisk-composer.

[44] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. `libmpk`: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *USENIX Annual Technical Conference*, 2019.

[45] 128 Core Processor. https://www.crn.com/news/components-peripherals/ampere-s-new-128-core-altra-cpu-targets-intel-amd-in-the-cloud.

[46] 80 Core Processor. https://amperecomputing.com/ampere-altra-industrys-first-80-core-server-processor-unveiled/.

[47] AMD Epyc 7002 series. https://www.amd.com/en/processors/epyc-7002-series.

[48] Ran Ribenzaft. What AWS Lambda's performance stats reveal, April 2019. https://thenewstack.io/what-aws-lambdas-performance-stats-reveal/.

[49] Serverless data processing with AWS step functions. https://medium.com/weareservian/serverless-data-processing-with-aws-step-functions-an-example-6876e9bea4c0.

[50] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[51] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[52] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-as-a-Service. In *International Conference on Very Large Data Bases*, 2020.

[53] AWS Step Functions pricing. https://aws.amazon.com/s3/pricing/.

[54] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *USENIX Security Symposium*, 2019. Tool available at: https://gitlab.mpi-sws.org/vahldiek/erim/-/tree/master/.

[55] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *ACM Symposium on Operating Systems Principles*, 1993.

[56] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joeseph M. Hellerstein. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018.