BeeHive: Sub-second Elasticity for Web Services with Semi-FaaS Execution

Ziming Zhao
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Shanghai, China
dumplings_ming@sjtu.edu.cn

Mingyu Wu
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Shanghai AI Laboratory
Shanghai, China
mingyuwu@sjtu.edu.cn

Jiawei Tang
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Shanghai, China
jiawei_tang@sjtu.edu.cn

Binyu Zang
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Shanghai, China
byzang@sjtu.edu.cn

Zhaoguo Wang
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Shanghai, China
zhaoguowang@sjtu.edu.cn

Haibo Chen
Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Shanghai, China
haibochen@sjtu.edu.cn

ABSTRACT

Function-as-a-service (FaaS), an emerging cloud computing paradigm, is expected to provide strong elasticity due to its promise to auto-scale fine-grained functions rapidly. Although appealing for applications with good parallelism and dynamic workload, this paper shows that it is non-trivial to adapt existing monolithic applications (like web services) to FaaS due to their complexity. To bridge the gap between complicated web services and FaaS, this paper proposes a runtime-based *Semi-FaaS* execution model, which dynamically extracts time-consuming code snippets (closures) from applications and offloads them to FaaS platforms for execution. It further proposes *BeeHive*, an offloading framework for Semi-FaaS, which relies on the managed runtime to provide a fallback-based execution model and addresses the performance issues in traditional offloading mechanisms for FaaS. Meanwhile, the runtime system of *BeeHive* selects offloading candidates in a user-transparent way and supports efficient object sharing, memory management, and failure recovery in a distributed environment. The evaluation using various web applications suggests that the Semi-FaaS execution supported by *BeeHive* can reach sub-second resource provisioning on commercialized FaaS platforms like AWS Lambda, which is up to two orders of magnitude better than other alternative scaling approaches in cloud computing.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575752

CCS CONCEPTS

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

KEYWORDS

Cloud Computing, Function-as-a-Service, Java Virtual Machine

ACM Reference Format:
Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, Haibo Chen. 2023. BeeHive: Sub-second Elasticity for Web Services with Semi-FaaS Execution. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3575693.3575752

1 INTRODUCTION

The dynamic nature of the real-world web environment stimulates strong demand for resource elasticity, i.e., to rapidly and automatically scale with the fluctuating workload. Fortunately, cloud vendors have proposed many different scaling mechanisms, and function-as-a-service (FaaS) is one of the most recent and popular solutions. Compared with others, FaaS automatically scales applications in a finer granularity (namely functions) and shorter reaction time. It also embraces a pay-as-you-go model for cost-efficient computation.

FaaS has drawn great attention since its birth. Mainstream cloud vendors have provided their own FaaS platforms [12, 17, 32, 36, 47], while prior work has proposed to run various applications as FaaS functions, including video processing [30], software compilation [29], micro-services [31, 39], data-parallel execution [55, 56], etc. Those applications exhibit massive parallelism, which suits FaaS well in that they can be supported by elastic and unbounded computing resources in the cloud with acceptable budgets. However, FaaS encounters challenges when applying to more complicated scenarios. A typical example is traditional monolithic web applications. Although they also require elastic resources to tackle request bursts, it is quite difficult to migrate them to FaaS.

To further understand the challenges in adapting web applications to FaaS, this work tries three different mechanisms on an enterprise-level web application named *pybbs* [8], but none of them is satisfying. First, it is inappropriate to directly run those complicated applications atop FaaS due to their stateful nature violates the stateless assumption of FaaS. Second, it is not practical to manually break them into fine-grained functions for FaaS due to their code complexity and tight integration with underlying development frameworks. Third, it is not feasible to statically slice web applications due to dynamically generated classes and general call stubs from frameworks. The above observations call for a new execution paradigm, which leverages the power of FaaS while still keeping the monolithic nature of web applications.

This work thus proposes *Semi-FaaS*, a new execution model for complicated applications (like web services) to embrace FaaS. Figure 1 illustrates that Semi-FaaS combines the execution model of both traditional monolithic services and FaaS: instead of directly running or code refactoring, Semi-FaaS only uploads fine-grained code snippets to FaaS for execution while executing the rest and maintaining states on the monolith side (referred to as *server*). Due to the complexity of static analysis, Semi-FaaS relies on managed runtimes from high-level languages to dynamically extract code snippets and offload them to FaaS. We have implemented *BeeHive* in Java, a managed language widely used in web applications, to realize the Semi-FaaS model. Our contributions are as follows.

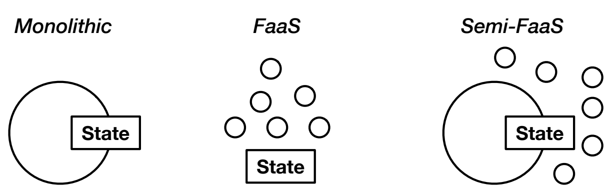


Figure 1: The Semi-FaaS model combines both

An offloading-based Semi-FaaS execution model. *BeeHive* provides an offloading-based execution model, which automatically extracts and offloads sliced Java code snippets from the original monolithic application (or *server*) to FaaS platforms for execution. Rather than statically analyzing complicated code of web applications, *BeeHive* relies on the language runtimes (JVMs) to collect the application profiles and select time-consuming functions for offloading. Afterward, *BeeHive* calculates an initial *closure* from the function and sends it to FaaS. Due to the inaccuracy of dynamic analysis, the initial closure is incomplete, so *BeeHive* embraces a fallback-based mechanism, which returns the control flow from FaaS back to the server to handle issues like missing code and data. The fallback mechanism continuously completes the closure and quickly achieves comparable performance against that on the server.

Comprehensive analysis and optimizations to reduce the fallback overhead. The major shortcoming of *BeeHive*'s offloading mechanism is the fallback overhead, which mainly consists of three parts according to our analysis. First, frameworks in web applications heavily rely on native invocations, many of which are coupled with native states and cannot be offloaded. *BeeHive* proposes the *Packageble* abstraction, which allows packing related native states into closures and thus enables offloading native invocations. Second, web applications frequently communicate with external services (like databases) through stateful connections. Since those connections are not offloadable, *BeeHive* provides a proxy-based communication mechanism to eliminate network-related fallbacks. Third, the first few invocations to FaaS functions suffer from large overhead due to excessive fallbacks, the warm-up phase in JVM, and the instance construction in FaaS platforms (also known as *cold boot*). *BeeHive* thus provides the *shadow execution* mechanism, which executes duplicated user requests on FaaS with no side effects, while the real requests are executed in servers and thus not bothered by any fallbacks.

A runtime system to support consistent execution on multiple endpoints. The runtime system in *BeeHive* is responsible for data sharing, offloading method selection, and memory management among multiple endpoints (servers and FaaS functions). To enable efficient data sharing among multiple endpoints, *BeeHive* modifies the object address layout to distinguish remote references, and it also relies on the Java Memory Model (JMM) to handle synchronizations among endpoints. It further leverages the intensively-used annotations in frameworks to filter candidate functions and provides a profiler to find time-consuming ones for FaaS execution. *BeeHive* also proposes a low-pause garbage collector to reclaim memory resources among different endpoints and provides a re-execution-based failure recovery mechanism to handle failures on FaaS invocations.

A thorough evaluation against other scaling solutions. *BeeHive* is implemented atop the HotSpot JVM for OpenJDK 8. We have employed it on two widely-used FaaS platforms, OpenWhisk [15] and AWS lambda, and evaluated it with enterprise-level web applications [13]. The result shows that *BeeHive* can react to the dynamic workload with instances in the FaaS platform and reduce the tail latency in less than one second at best, which is up to two orders of magnitude better than other scaling approaches provided by AWS. Thanks to the optimizations in *BeeHive*, the number of fallbacks is trivial and leads to moderate execution overhead. However, for applications inducing many fallbacks (e.g., frequent synchronization on shared variables), the overhead of *BeeHive* may still be considerable.

2 ANALYSIS AND MOTIVATION

2.1 Tackling Request Bursts with FaaS

Request bursts are long-term enemies for web applications. Due to the highly dynamic characteristics of requests, bursts are usually unpredictable but damaging. In the last few years, well-known web services like Twitter and Paypal have suffered from unavailability when facing sudden request bursts [1, 2]. To show the effect of request bursts on web applications, this paper uses *pybbs*, an enterprise-level web service studied by prior work, as an example for analysis. pybbs is a popular open-source forum built with mainstream web frameworks like Spring [59] and contains 24692 Java classes in all [13]. We run the pybbs web server in an AWS m4.xlarge instance (4 vCPUs) and simulate clients to comment on different topics. Figure 2 indicates that both the average and tail latencies of pybbs dramatically increase with the number of concurrent clients, which may lead to the degradation of user experiences.

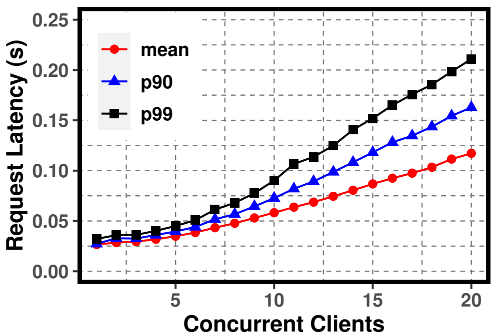


Figure 2: The latency of web service (pybbs) rapidly increases with the number of concurrent clients

An intuitive solution to handle request bursts is adding more resources. Fortunately, cloud vendors have provided various solutions for resource scaling. Taking AWS as an example, it provides the following choices.

- **Reserved instance.** Customers can reserve cloud resources and use them once request bursts happen. Since those instances are prepared in advance, they can rapidly handle request bursts, but the cost is relatively high: the instances must be active no matter if they are used, and they should be reserved for at least one year [18].
- **On-demand instance.** As the name suggests, this type of instance can be created on-demand to handle request bursts. However, the instance creation time is relatively long.
- **Burstable instance.** This type of instance is similar to reserved instances but embraces a different billing model: when the resource is not sufficiently used, AWS reduces its cost. The burstable instance can also be used on-demand, but we mainly discuss its reserved use case in this work.
- **Fargate.** AWS Fargate provides a similar abstraction to FaaS: it automatically scales the resources (in containers) to meet the demand of dynamic workloads. Nevertheless, the granularity for configuration and billing is not flexible compared with FaaS.
- **Lambda (FaaS).** AWS lambda claims to scale smaller pieces of code (i.e., functions) in a rapid, elastic, and automatic fashion.

Table 1 summarizes the characteristics of the above scaling solutions. The preparation time for different solutions is evaluated by employing a prepared system image with OpenJDK 8 installed. Compared with others, the FaaS solution (Lambda) finishes resource provisioning in a second or less, and the billing granularity is as small as a millisecond. As for configuration, FaaS allows customers to configure memory resources in MB, while others only support GB-level configuration. Last but not least, only FaaS and Fargate support *auto-scaling*, which automatically scales resources up and down due to workload fluctuation, while others need to involve manual resource management[1]. To summarize, FaaS would be an appealing choice to rapidly auto-scale and fight against request bursts.

2.2 Implications of Applying FaaS to Web Applications

Although FaaS has virtues like rapid auto-scaling and pay-as-you-go billing model, it is not trivial to adapt existing web applications to FaaS platforms. In this section, we show three different methods to migrate an enterprise-level web service (*pybbs*) into FaaS platforms for execution, but none of them is satisfying. Our experiences are shown below.

Method 1: direct execution. The most straightforward way is to directly run web applications as a whole atop FaaS platforms. Although the notion of FaaS suggests a brand-new instance should be created for each request, mainstream platforms have provided instance caches to mitigate the overhead of environment setup (or *cold boot*). The life span of a cached instance is usually hours [64], which is enough for a short-term request burst.

However, FaaS is mainly designed for auto-scaling fine-grained and stateless tasks (functions), which is not the case for monolithic and stateful applications. Web services usually contain complex local states like sessions and local files, which violates the stateless nature of FaaS. When a function (e.g., a request) finishes its execution, the local states might be abandoned by the FaaS platform, which leads to data loss and unavailability. Prior work has shown that both popular monolithic web services [13] and smaller micro-services [40] contain complex states, which makes it difficult to migrate them to FaaS. Although FaaS platforms have provided stateful support recently [22, 38, 58, 66], they still require significant modifications to existing applications for state management. Despite local states, the large code base of web services also makes it inconvenient for FaaS execution. For example, mainstream FaaS platforms only support launching a function from an uploaded jar file whose size is not larger than 50MB. In contrast, the packed jar file for pybbs is 67MB and thus cannot be directly uploaded for execution.

Experience 1: Direct execution is not appropriate as web applications' stateful and monolithic nature violates the stateless and lightweight assumptions of FaaS.

Method 2: manual rewriting. Our experience with direct execution suggests that directly running web applications on FaaS is not practical. We thus turn to a rewriting-based method, which manually splits applications into small pieces and selectively uploads some of them to FaaS. Unfortunately, this method is also unacceptable due to the complexity of web applications. Those applications contain a large number of classes, which are mainly contributed by web development frameworks. Although those frameworks greatly reduce the development labor with their expressive annotations and versatile functionalities, their complexity makes rewriting quite difficult. As for *pybbs*, 99.6% of its compiled jar file is filled with framework-related Java classes, including user authentication (Spring [59]), object-relational mapping required

[1]EC2 also supports auto-scaling, but it can only create new instances and users are still responsible for launching services on them

Table 1: Comparisons on existing scaling solutions exemplified by AWS

Scaling solution	Minimum running time	Billing granularity	Preparation time	Configuration granularity (memory)	Auto-scaling
Reserved	1 year	years	-	GB	no
On-demand	1 minute	seconds	~40 seconds	GB	no
Burstable	1 year	years	-	GB	no
Fargate	1 minute	seconds	~40 seconds	GB	yes
Lambda (FaaS)	**1 millisecond**	**milliseconds**	**<1 second**	**MB**	**yes**

by storage services (MyBatis [6, 7]), and optimizers (HikariCP [5]). To rewrite web applications into smaller parts, developers need to manually refactor those frameworks into lightweight ones. Recent experiences [4] also show that totally rewriting web applications to replace the frameworks' functionalities may take years, which suggests the rewriting method is infeasible and thus necessitates an automatic approach [40].

Experience 2: Web applications are too complicated to manually rewrite into smaller, FaaS-friendly pieces.

Method 3: static code analysis. As rewriting is infeasible, an alternative solution is to automatically extract executable pieces from web applications through static analysis. Unfortunately, static analysis has also proven difficult when applied to monolithic web applications [13] since development frameworks heavily rely on dynamic code generation for helper classes and general call stubs. Taking the comment request in pybbs as an example, frameworks generate 287 new classes for the request class, which greatly enlarges the code base for analysis. Those generated classes wrap the real comment request with nearly 20 indirect invocations, resulting in a deep call stack. Furthermore, many call sites use general stubs for invocations, which contain tens of possible call targets for each. A typical example is *MethodInterceptor*, a commonly-used stub to intervene in user-defined methods. In pybbs, *MethodInterceptor* has 31 different kinds of implementations. All dynamically generated classes and stubs create obstacles for a static analyzer to split web applications into FaaS-friendly functions.

Experience 3: Web applications are too dynamic to be statically analyzed.

2.3 Design Principles of Semi-FaaS

Our analysis necessitates a new execution model for web applications to leverage the power of FaaS, which should conform to the following principles:

- **Partial**. Web applications should be partitioned, and only a part of them should be offloaded to FaaS for cost-efficient execution.
- **Automatic**. Due to the code complexity of web applications, they should be automatically partitioned and uploaded to FaaS.
- **Dynamic**. Due to the dynamic nature of web applications, they should be dynamically analyzed for smart partitioning and offloading.

Considering such principles, we thus propose *Semi-FaaS*, a model combining the normal execution with FaaS for web applications. We implement *BeeHive* to realize the Semi-FaaS execution model.

3 OFFLOADING-BASED SEMI-FAAS WITH BEEHIVE

3.1 Overview

According to experiences in Section 2.2, we build a Semi-FaaS execution model atop *BeeHive*, an automatic and dynamic offloading framework supported by managed runtimes (like JVMs). The architecture of *BeeHive* is shown in Figure 3. Note that the design of *BeeHive* is not restricted to JVMs and can be used in other language virtual machines like JavaScript V8.

BeeHive mainly contains two parts: long-running *servers*, which originally accept and process user requests, and *FaaS platforms*. It is non-intrusive to both FaaS platforms and operating systems and can be directly used by commercialized ones like AWS Lambda. When facing request bursts, *BeeHive* controls servers to proactively offload a part of its workload as functions to FaaS platforms for execution, while the rest is still handled by the server (namely *Semi-FaaS*). The number of offloaded requests is determined by an *offloading ratio*, and *BeeHive* can scale in and out by setting the ratio. For each offloaded function, *BeeHive* leverages the managed runtimes provided by high-level languages like Java to track their runtime behaviors and construct an *initial closure*, which is the basic unit for FaaS execution. The initial closure contains code (Java bytecode) and data likely to be used according to dynamic profiling, which is sent together with user arguments to the FaaS platform. After receiving the initial closure, the FaaS platform assigns it to an instance (usually containers or virtual machines) for execution.

With the offloading mechanism above, *BeeHive* can achieve user-transparent and lightweight offloading by only sending closures to FaaS platforms. Nevertheless, due to the dynamic nature of *BeeHive*'s offloading mechanism and the complexity of monolithic web applications, the initial closure is incomplete, and the execution on FaaS can encounter issues like missing code or data. To this end, *BeeHive* embraces a fallback-based approach and relies on the managed runtime to detect and handle all fallbacks. As shown in Figure 3, functions on FaaS send fallback-related requests to the server, while the server handles requests and sends the results back so that the offloaded function can resume its execution. For example, if the function encounters a missing code, it sends a request with the name of the missing Java class, while the server sends the corresponding class file back to the function. The fallback-based mechanism repeats until the offloaded function exits with a return value.

The fallback mechanism in *BeeHive* ensures the progress of FaaS execution. Furthermore, the frequency of fallbacks gradually decreases as the closure is refined by receiving results from the server (e.g., fetched code and data). Nevertheless, the overhead of fallbacks is still considerable, and *BeeHive* is responsible for (1) reducing the

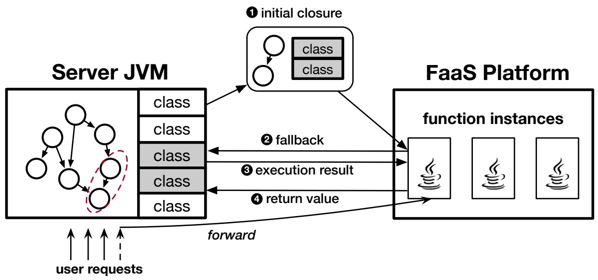


Figure 3: The workflow of *BeeHive*'s offloading mechanism

number of fallbacks and (2) reducing the performance overhead of fallbacks. Specifically, *BeeHive* needs to handle the following challenges given the characteristics of web applications, and we will elaborate on how *BeeHive* solves them in the rest of this section.

- **Native invocations.** Native invocations are intensively used in web applications, but they are treated as not offloadable due to tight coupling with JVM-specific native states, which leads to fallbacks for each native call.
- **Stateful connections.** Web applications are interactive and contain connections with external services like databases. Those connections cannot be directly offloaded and cause fallbacks for each inter-service communication.
- **Warmup overhead.** The number of fallbacks is large for the first few executions on FaaS. Furthermore, the FaaS platform needs to establish a fresh runtime environment for function execution, which also contributes to prohibitive execution overhead, and naively offloading functions leads to a long tail problem.

3.2 Handling Native Invocations

High-level languages like Java allow applications to use native libraries through their native interfaces. Due to the dynamic nature of web applications, native invocations are heavily used. For example, the frameworks frequently use the Reflection APIs to query the metadata of classes and methods, which invokes corresponding native methods to access off-heap data. Since native data is out of management by the Java heap, prior work usually avoids offloading native invocations. For example, COMET [34] returns from the offloaded endpoint to the original device when encountering a native call. However, the fallback overhead is not acceptable for web services, and *BeeHive* should reduce fallbacks related to native invocations.

Table 2: Native methods used in pybbs request handling

Categories	Invocation Numbers	Representative Methods
Pure on-heap	226643	System.arraycopy
Hidden states	34749	MethodAccessor.invoke0
Network	248	socketRead0
Others	415	Thread.currentThread

We first analyze which native methods are heavily used in web applications, exemplified by *pybbs*. Table 2 shows the number of native invocations by dividing them into different categories. Since each request contains over 200 thousand invocations to different native methods, simply returning to servers for handling would introduce prohibitive overhead.

Fortunately, most native invocations do not involve complicated manipulations on native states and can be directly handled on the FaaS side. We divide those native methods into four categories and handle them separately.

Pure on-heap operations. Most invoked native methods only manipulate data on the heap. Applications and libraries use them to improve their performance (for example, using *arraycopy* to copy a large array). Since those methods do not impact off-heap states, they can be directly executed on FaaS.

Hidden states. Although most invocations are easy to handle, web applications also invoke methods that contain hidden states stored off-heap. Those methods are frequently invoked by web development frameworks mainly because they need to access object metadata like classes and methods. Therefore, *BeeHive* proposes an abstraction named *packageable classes*, which pack native states together with Java objects to support direct invocation on FaaS without fallbacks.

Packageable classes are implemented via the *packageable* Java interface. A Java class implementing the *packageable* interface contains methods to specify how to marshal/unmarshal the native states owned by a Java class. During offloading, if the type of an object is packageable, *BeeHive* will invoke its marshal method to pack native states into the closure and subsequently call its unmarshal method to transform and store the native states on the FaaS side. A typical example is a *Method* object which stores an off-heap reference to corresponding method-related metadata. If *BeeHive* only offloads the Method object, its native states become uninterpretable and induce correctness issues for related methods like *invoke0*. Therefore, we refactor the *Method* class to implement *packageable* to include necessary metadata (such as the method name) into the closure and unmarshal it on the FaaS side. Thanks to the packageable interface, *BeeHive* can offload native methods together with their corresponding states and avoid a large number of fallbacks. The annotation burden is also acceptable: we manually enhance 15 Java classes as packageable. Since all those classes are in the Java system library (JDK), other applications can also reuse them for offloading.

Network-related. Web applications rely on native socket APIs to communicate with others (like databases). *BeeHive* proposes a separate approach for those network-related invocations, which relies on the support of packageable classes (discussed later in Section 3.3).

Others. Other methods, such as *currentThread()* are stateless and cause no side effects among invocations. Those methods can also be tagged and directly executed on FaaS.

After handling the above four categories of native methods, *BeeHive* allows most native invocations in offloaded code snippets to execute directly on FaaS. As shown in Section 5, fallbacks due to native invocations have been eliminated in all evaluated applications.

3.3 Proxy-Based Connection Management

Web applications are also connected with others during their execution. A typical example is storage services (or databases), which are used to maintain persistent states. As for pybbs, the execution of each comment request includes more than 80 rounds of communication with databases. If network communication cannot be offloaded, the fallback overhead for each request would become considerable. Nevertheless, the connections between web applications and databases contain complex system states, and offloading them may involve migrating kernel-related states to the FaaS platform, which is not practical in our user-level design. To this end, we propose a proxy-based approach to manage those stateful connections.

The core idea of our proxy is to share a connection to external services between servers and FaaS functions. As shown in Figure 4, each database service can have its own proxy for connection management, which runs on the same machine as the service. When the server establishes a connection with the database, it is actually connected via the proxy. The proxy originally maintains connections to both database and server by memorizing their corresponding descriptors. When a connection needs to be offloaded, the server sends a special *prepare* request to the proxy to generate a unique ID for the connection. This ID is stored in the proxy and returned to the server. Afterward, the server treats the ID as a part of native states related to the offloaded network-related object (the corresponding type is *SocketImpl*) and packs it into the initial closure. As for the FaaS side, it unpacks the native states and connects to the proxy with the unique ID. After receiving the ID, the proxy can determine that the request is from an offloaded function, so it maintains a descriptor mapping among the server, FaaS, and database. Subsequently, requests from the FaaS function will be redirected and sent to the database through the same connection it uses before offloading, and no fallback is required.

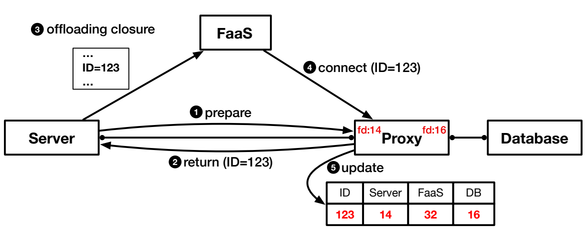


Figure 4: *BeeHive*'s proxy-based connection management

3.4 Hiding Warmup with Shadow Execution

A newly-offloaded function may encounter an extremely long execution time due to a warmup phase whose overhead consists of three parts. First, as the resources are supplied on demand in FaaS platforms, a function may suffer a *cold boot* when it is running for the first time, which involves launching new instances (virtual machines or containers), deploying JVMs, creating network connections, etc. Second, since JVMs need to load required classes and profile user methods for optimizations, the first-time execution is usually slow. Third, as the initial closure is incomplete, it triggers many fallbacks to fetch missing code and data. Although prior work [23, 26, 52, 57, 62] has proposed solutions to mitigating the effects of cold boots in FaaS, they cannot solve fallback-related problems and still cause the long-tail problem for offloaded functions. Instead of optimizing the warmup, *BeeHive* proposes an alternative method named *shadow execution* to hide the warmup overhead from users.

The shadow execution in *BeeHive* offloads a duplicated user request to the FaaS platform, and the execution introduces no side effects on observable states. The real request is executed on the server side and directly returned to users once complete. When the shadow execution finishes, the warmup phase is passed and the closure on the FaaS side has been refined, so the subsequent requests can be effectively offloaded and executed.

The major challenge for shadow execution is how to process a duplicated user request without introducing observable state modifications. *BeeHive* divides potentially observable states into two parts: memory states and external states. As for memory states, *BeeHive* relies on the *stateless* feature of FaaS functions, which suggests all states on the FaaS platform can be treated as invisible (an exception is shared states between FaaS and servers, details in Section 4). As for external states, they are usually persisted in databases, which are managed by neither the server nor the FaaS platform. To this end, *BeeHive* leverages the aforementioned proxies to intercept all operations on external states and specially handle those from a shadow FaaS execution. When the shadow execution begins, the FaaS platform sends a *shadowbegin* message to the proxy. The message also contains an identifier to specify the FaaS function so that its subsequent write requests introduce no side effects. When the shadow execution ends, the FaaS platform sends a *shadowend* message, and subsequent requests from the function can be normally handled.

4 THE BEEHIVE RUNTIME SYSTEM

Laying the foundation of offloading, the *BeeHive* runtime is responsible for handling all communications between servers and FaaS platforms, including state management, closure construction, and memory management. Building such runtime for offloading also faces several challenges. First, *BeeHive* should correctly handle cases when states are shared among FaaS functions and servers. Second, *BeeHive* should choose suitable functions for offloading regardless of the code complexity of monolithic web services. Third, *BeeHive* should manage memory resources in a distributed manner. Lastly, *BeeHive* should recover from cases where a remote invocation to FaaS fails.

4.1 Distributed Object Sharing

Since the offloaded functions contain objects originally residing in the server, they are potentially shared with other endpoints (the server and other FaaS functions), and *BeeHive* needs to handle them specially. Figure 5 illustrates an example where objects are shared between a newly offloaded function and the server. When the function is is being launched, the server JVM constructs the initial closure to include objects likely to be used by the offloaded function. Suppose the dynamic analysis includes object a and c in the closure but excludes b. The chosen objects are copied into a buffer and sent to a FaaS instance for execution. Since object a points to b, which is not in the closure, *BeeHive* needs to mark the reference as a remote one. As Figure 5 shows, *BeeHive* modifies the reference in *a* to mark the most significant bit as 1. Since the resulting address is only used in the kernel space, it cannot be confused with normal heap references in a JVM on FaaS.

When the FaaS function receives the closure, it directly copies all objects therein to its own heap. Those objects are stored in a separate space called *closure space* for ease of memory management (details in Section 4.4). After copying, the FaaS function responds with the start address of the closure space. Since the copied objects remain in the same order as those in the initial closure, the server can easily calculate the new address in FaaS and establish a one-to-one address mapping for each offloaded object. This mapping is responsible for synchronizing updates on the shared objects between FaaS functions and the server.

Introducing remote references allows *BeeHive* to restrict the initial closure size, but it may cause unexpected behaviors when FaaS functions access them. For example, the FaaS function in Figure 5 may access object *b*, whose address is out of the heap range. To ensure correctness, *BeeHive* instruments check for each reference loading operation to detect remote references. When the most significant bit is set, *BeeHive* triggers a fallback, fetches the corresponding object from the server, and resets the bit to avoid repeated fallbacks. Note that the check instructions are only added on the FaaS side and thus induce no overhead on the server.

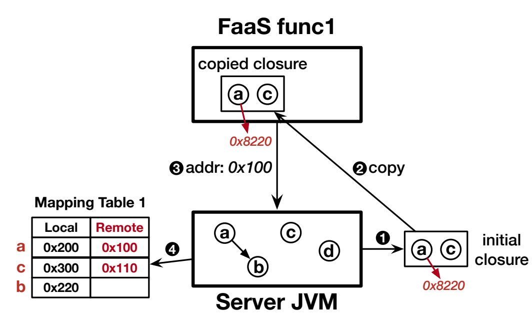


Figure 5: *BeeHive*'s object sharing mechanism

4.2 Shared State Synchronization

Applications would leverage synchronization primitives to coordinate accesses on shared objects which may be distributed to FaaS functions in *BeeHive*. To this end, *BeeHive* needs to support state synchronization to ensure consistent execution for multiple endpoints. A naive solution would be broadcasting all updates conducted by any endpoint, which can cause considerable overhead. Similar to prior work [34], *BeeHive* embraces a release consistency model based on the Java Memory Model (JMM) [35], which synchronizes states among multiple endpoints for each synchronization primitive. For simplicity, we introduce the implementation of commonly-used monitor-based synchronizations as an example. Other synchronization operations, like volatile memory accesses, are also supported by *BeeHive*.

In Java, every object can be used as a lock to handle synchronizations with the *synchronized* keyword. JMM states the happen-before relationship with object locks: if thread *A* acquires a lock previously released by thread *B*, then all memory operations before the lock releasing operation in thread *B* should be observed by thread *A* before any of its future memory operations. *BeeHive* conforms to JMM and leverages locks for state synchronizations among endpoints. Figure 6 shows the workflow of a state synchronization between two FaaS functions. When function 1 acquires the lock, it checks the last owner of it. If it is previously held by other endpoints (function 2 in this example), a happen-before relationship should be established between them, and a synchronization is required. To this end, function 1 first communicates with the server for coordination, and the server subsequently contacts the previous lock owner (function 2). On receiving the acquiring request, function 2 sends related states (in objects) together with the lock to the server. Since the server has maintained the address mapping for all functions, it translates the object addresses from function 2 to function 1 and finally responds to function 1 for both lock granting and heap synchronization. Although this design involves the server for each synchronization, it avoids computing and maintaining address mappings in individual FaaS functions, which are volatile and can be destroyed by the FaaS platform. In our evaluation, we show that the overhead is acceptable given the low frequency of synchronizations.

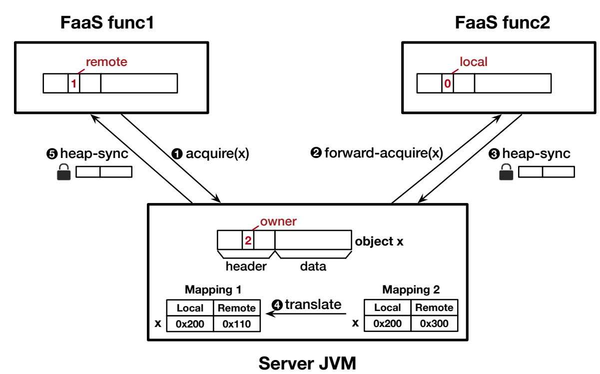


Figure 6: Lock-based synchronization in *BeeHive*

The remaining issue is to determine which objects should be sent for each synchronization. First, we only need to consider objects in the closure as they are shareable with other endpoints. To further reduce the data size for each synchronization, *BeeHive* instruments write operations to maintain a dirty object list for each endpoint and only send them upon synchronization. The instrumented instructions are also simple and induce moderate overhead.

4.3 Root Method Selection

The initial closure for offloading is constructed from a *root method*. Once selected, *BeeHive* recursively traverses code and data likely used by the root method and packs them as an initial closure. However, due to the complexity of monolithic web applications, it is difficult to determine which method should be chosen as the root to construct the initial closure. A strawman design would be relying on the dynamic profiling module embedded in JVM's interpreters to find suitable methods according to their invocation count and frequency. Unfortunately, web development frameworks like Spring introduce call stubs, or *interceptors*, to manage user-provided methods. Those interceptors are frequently invoked, but they contain many possible target methods, so offloading them would induce a large closure and unsatisfying performance. Therefore, *BeeHive* needs to filter out those framework-related methods before selection.

To capture the business logic, web development frameworks require users to annotate their methods (for example, *comment* in pybbs) so that they can be instrumented and managed. Similar annotations are commonly supported in various frameworks regardless of language, such as Java Struts, Python Django, and NodeJS Express. We find that they can be used to distinguish user-provided methods from framework-generated ones without any modifications to applications. To this end, we introduce the notion of *offloading candidates*, which consists of methods already annotated by users during development. Only those candidates can be chosen as root methods and constructed as offloadable initial closures.

The profiler in *BeeHive* is implemented via a Java agent, which records the invocation count and the accumulated execution time for each candidate method through code instrumentation. *BeeHive* leverages two heuristics to choose offloaded methods. First, the accumulated execution time should be large. Second, the average execution time should not be short (e.g., less than one millisecond) to avoid large execution overhead. Although simple, this policy can pick out suitable root methods for offloading and mitigate the pressure on the server. Advanced root method selection policies taking method behavior (e.g., synchronization operations) into consideration may result in better decisions. We left the policy design as our future work.

4.4 Memory Management

After offloading code snippets to FaaS for execution, *BeeHive* is responsible for managing memory resources for both servers and FaaS instances. Fortunately, the execution model of FaaS functions in *BeeHive* is clearly defined, which makes it simple to implement a collector with short pauses. As illustrated in Section 3.1, the execution of all FaaS functions in *BeeHive* starts from an initial closure. *BeeHive* assumes all objects in the initial closure are useful for function execution, so none of them is collected or moved unless the FaaS instance is destroyed. Similarly, if a remote object is fetched from the server or other FaaS functions, it is also treated as alive. In contrast, objects created during execution are only useful in the context of a single invocation. When the function finishes its execution, those objects can be collected.

Considering the different lifecycles of objects, *BeeHive* implements a two-space garbage collector for FaaS functions. It first constructs a *closure space* for the initial closure, and objects fetched from remote are also added to the closure space. As for other newly created objects, *BeeHive* uses an *allocation space* to serve normal heap allocation. The closure space and allocation space resemble the old-young heap layout in traditional generational collectors, except that closure space is never collected as all objects are treated alive. When the allocation space is exhausted, *BeeHive* traverses the heap from roots on the stack and the closure space to reclaim dead objects. After GC, the allocation space becomes nearly empty and thus ready for subsequent allocation requests. To avoid costly scanning on the whole closure space during GC, *BeeHive* inherits the *card table* design from generational GC and marks a range of memory (512 bytes in *BeeHive*) as *dirty* if a cross-space reference exists. When GC is triggered, only the part marked as dirty should be scanned, so the performance overhead is trivial. As shown in Section 5.6, the pause time for *BeeHive*'s GC is only several milliseconds, and can be further hided by overlapping with network communications.

As for the server, its GC should consider the shared states with FaaS functions. Since *BeeHive* has maintained object mapping tables in the server to track shared objects, it only needs to add objects in the tables to the root set when GC starts, and other phases (such as mark and copy) remain unchanged. If a shared object is moved by the server, the corresponding mapping table should also be updated. Since the closure space for each FaaS function is not large, keeping those objects alive only introduces moderate overhead for the server.

4.5 Failure Recovery

The Semi-FaaS execution in *BeeHive* turns normal function calls in monolithic applications into remote invocations to FaaS, which increases the risk of failure. To this end, *BeeHive* provides an optional failure recovery mechanism to handle failed invocations.

Since the visible states of a FaaS function include shared memory with the server and persistent data on external storage, *BeeHive* needs to handle them upon recovery. Since prior work [66] has proposed methods to reach exactly-once execution for external storage, *BeeHive* can directly leverage it to ensure data consistency. Nevertheless, if a FaaS function has made its memory states visible by synchronization with the server, those states should be handled separately.

Since states on FaaS can only become visible via synchronizations, *BeeHive* embraces a re-execution mechanism to handle failures. When a synchronization operation is triggered, *BeeHive* asks for the function instance to send its execution stack, all objects referenced by the stack, and updated shared objects back to the server. Such a mechanism introduces moderate overhead since the size of the Java stack and related objects are usually restricted (several KBs). The stack information is stored together with the object mapping table for each function. If an invocation to FaaS fails, *BeeHive* sends the latest stack information together with the closure so that the FaaS function can resume its execution from the last synchronization point. The re-execution does not violate JMM as it only defers the execution of a remote FaaS function at other endpoints' views.

As for the server, since its role is the same as that of a monolithic web application, *BeeHive* does not need to add extra failure recovery mechanisms.

5 EVALUATION

5.1 Experiment Setup

We implement *BeeHive* on the HotSpot JVM of OpenJDK 8u265-ga. To evaluate the Semi-FaaS execution supported by *BeeHive*, we have compared it with other scaling solutions from AWS, including on-demand instances, burstable instances, and Fargate. We also assume a perfect burst handler to immediately forward requests with pre-defined policies once a burst happens. More complicated policies are out of this paper's scope and left as our future work.

We leverage two platforms to deploy *BeeHive*. OpenWhisk [14] is a prevalent open-source FaaS platform used by IBM Cloud Functions [37]. We launch OpenWhisk on a control node to manage all other EC2 instances in the us-east-1 region. Furthermore, we also deploy *BeeHive* on AWS Lambda, a commercialized FaaS platform, to study its performance. To leverage Lambda, we create a Semi-FaaS template as a container image, which only contains *BeeHive*'s JVM for the function to connect with the server. When facing request bursts, the server JVM sends requests to launch functions on Lambda from the Semi-FaaS template. Afterward, FaaS functions automatically connect with the server to receive user requests and closures for execution.

Since the CPUs used in Fargate and Lambda are Xeon vCPUs with 2.50GHz, we choose similar configurations for other instances. The on-demand instances are m4.xlarge (4 vCPUs/2.30GHz, 16GB DRAM), and the burstable instances are t3.xlarge (4 vCPUs/3.10GHz, 16GB DRAM). Instances used in Fargate also have 4 vCPUs and 16GB DRAM. Since *BeeHive* only handles one request at a time in each instance, those used by OpenWhisk are m4.large (2 vCPUs/2.30GHz, 8GB DRAM), and the DRAM size for Lambda instances is 1GB (0.6 vCPUs) or 2GB (1.2 vCPUs). As for the database, if the instance is small, it would become a performance bottleneck for all scaling solutions. To this end, we launch the database on an m4.10xlarge instance (40 vCPUs/2.40GHz, 160GB DRAM).

As for applications, we leverage the following web services for evaluation.

Image processing. This application simulates a web server that generates thumbnails for images (abbreviated as *thumbnail* hereafter). It is developed by ourselves with Spring [59] and used as a micro-benchmark to show how *BeeHive* performs with computation-intensive workloads. Since the thumbnail application requires more computation resources, its Lambda instance has 2GB DRAM while others have 1GB.

pybbs. An open-source forum containing 24692 classes. We use its *comment* request (containing both I/O and computation workload) for evaluation.

SpringBlog [9]. An open-source blogging system containing 18493 classes in all (abbreviated as *blog* hereafter). We leverage its *archive* request for evaluation, which fetches a large number of records from databases and thus becomes I/O-intensive.

5.2 Burst Reduction

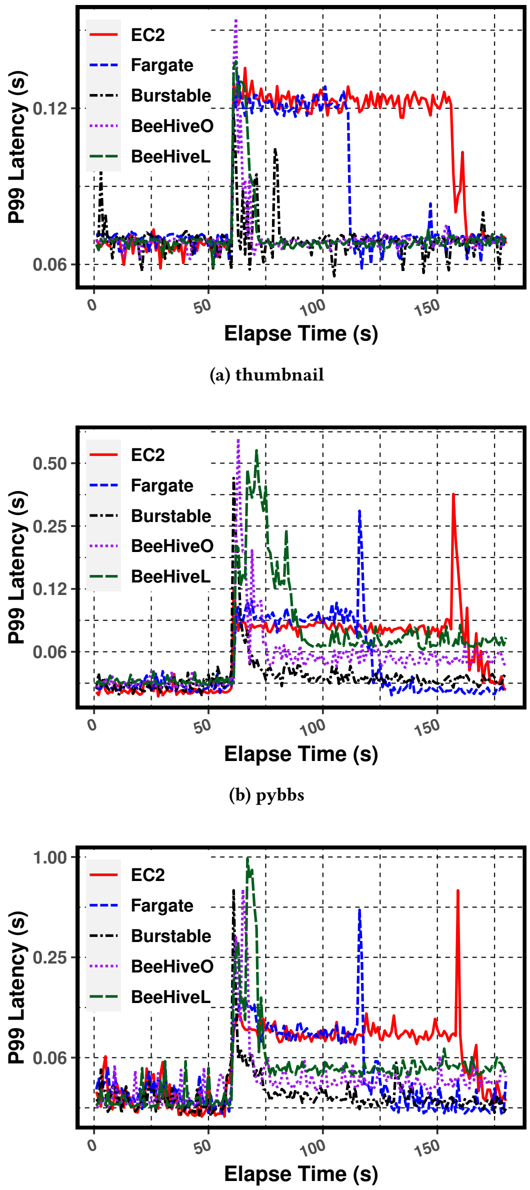
We first evaluate the time required to stabilize tail latency when facing bursts. In this evaluation, the normal workload is generated by concurrent clients sending requests repetitively (the number of clients is chosen to reach nearly peak throughput), while the workload for bursts is twice as heavy and lasts from the 60th second to the end. Once new instances become ready, the burst handler immediately forwards half of the workload to them. Figure 7 shows the per-second tail latency in 3 minutes for applications. The ideal case is the burstable instance, which keeps an always-on idle instance and leverages it immediately after a burst happens, so the tail latency remains relatively stable, but it results in idle resources without bursts. The EC2 reserved instance has a similar performance and thus does not show in the figure. In contrast, EC2 on-demand and Fargate instances require more time for resource provision and result in severe latency fluctuation. Although their preparation time is similar (mentioned in Section 2.1), on-demand instances suffer from a slower startup and require more time to launch applications. Since *BeeHive* only offloads lightweight code snippets to FaaS for execution, it reacts to request bursts more quickly. For the OpenWhisk configuration (*BeeHiveO*), the average duration to reach stable latency for all three applications is 9.33 seconds, which is 11.25× and 6.32× smaller than EC2 on-demand and Fargate, respectively. As for Lambda (*BeeHiveL*), since each instance has fewer CPU resources, it requires more time to warm up the JVM, and the average duration for stable latency is 16.33s (6.43× and 3.61× better than EC2 and Fargate).

Meanwhile, since FaaS platforms can keep function instances alive to reduce cold boot frequency (also known as *warm boot*), we also evaluate the case when cached instances are available on FaaS. In this case, *BeeHive* can reach sub-second resource provision for both OpenWhisk and Lambda and only take 632.78ms and 668.56ms on average to stabilize request latency for evaluated applications, which are two orders of magnitude better than other scaling solutions.

Although effective in reducing request bursts, the Semi-FaaS approach also introduces performance overhead. As shown in Figure 7, the stabilized p99 latency is larger than other scaling alternatives (by 15.0% compared with EC2 on-demand instances on average). The slowdown is mainly caused by fallbacks, barriers, and proxies. As for Lambda, the overhead becomes larger (averaging 31.0% compared with EC2). We find the performance difference mainly comes from larger network latency between Lambda function instances and EC2 servers even when they are configured in the same virtual private cloud (recommended by AWS to reach short network latency). Since pybbs and blog require frequent network communication with the database server, their performance significantly degrades compared with those on OpenWhisk. We further configure instances in OpenWhisk into different AWS available zones and the resulting overhead increases to 23.2% on average, which suggests the importance of network latency.

5.3 Throughput Analysis

We also study *BeeHive*'s throughput by comparing it with the vanilla JVM running on an always-on m4.xlarge server. Figure 8 shows that when running on the same server, *BeeHive* causes a 7.14% drop in peak throughput for pybbs. The overhead mainly comes from barriers to maintain dirty objects, and it can be further eliminated through recompilation if offloading is never required. As for blog and thumbnail, the peak throughput is almost the same. Meanwhile, the always-on configuration cannot further scale when its server is saturated, while *BeeHive* can easily scale to higher throughput by offloading more requests and creating more instances on the FaaS side. The saturated throughput on OpenWhisk is 840, 640, and 920 requests per second for thumbnail, pybbs, and blog, respectively,

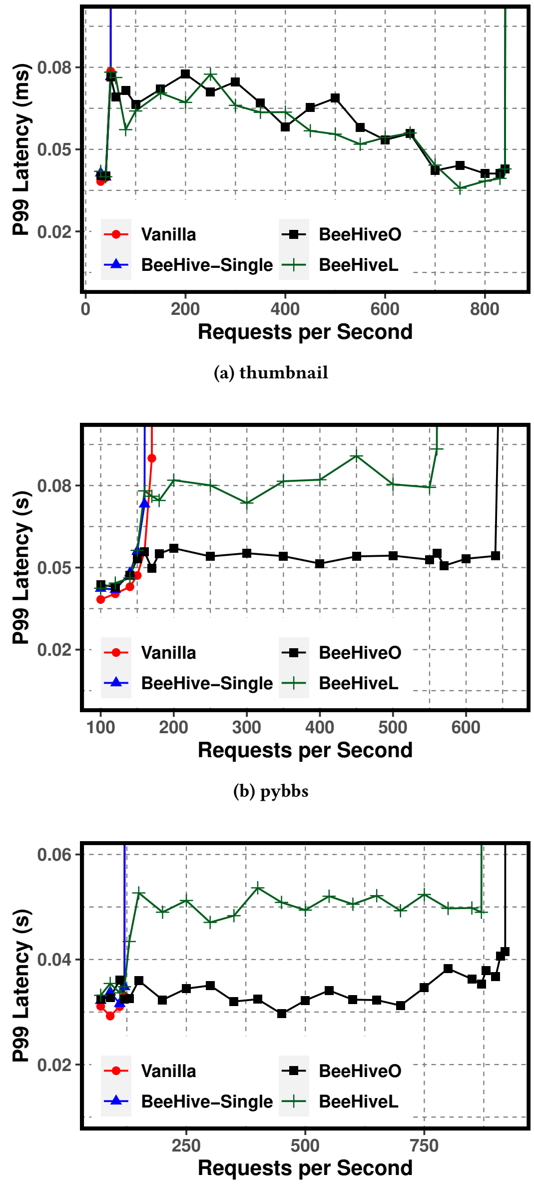


(a) thumbnail

(b) pybbs

(c) blog

Figure 7: Tail latency under dynamic workload



(a) thumbnail

(b) pybbs

(c) blog

Figure 8: Latency under various throughput settings

which averages 9.41× larger compared with the always-on configuration. As for Lambda, the saturated throughput is smaller for pybbs and blog due to its execution overhead, but it is still 9.11× better than the baseline. The maximum throughput of *BeeHive* is limited by the centralized server, and it can be further resolved by externalizing the contended shared states to distributed storage.

5.4 Cost Analysis

We also show the end-to-end financial cost of scaling solutions in Figure 7. For *BeeHive*'s OpenWhisk configuration, we assume the price of each instance is equal to EC2 on-demand ones. Since other scaling solutions always use one more instance for scaling, their cost remains the same among applications. As shown in Table 3, the overall cost ($) of *BeeHive* is larger than others. Since each FaaS instance contains a full-fledged JVM, *BeeHive* introduces more overhead on issues like dynamic compilation and thus requires more computation resources.

Nevertheless, when the frequency of bursts becomes lower, the cost of *BeeHive* declines. Take pybbs as an example: Figure 9 shows the per-hour cost of scaling solutions when varying the burst ratio

Table 3: Financial cost for scaling in Figure 7

Scaling solutions	thumbnail	pybbs	blog
EC2	0.007	0.007	0.007
Fargate	0.008	0.008	0.008
Burstable	0.005	0.005	0.005
BeeHiveO	0.010	0.017	0.013
BeeHiveL	0.012	0.010	0.008

(the duration of bursts in an hour) while the burst workload is the same as Figure 7b. Since the burstable instance is reserved for a long time, its cost remains constant regardless of the burst ratio. When the ratio is 67% (the vertical line in Figure 9), the scenario is the same as that in Figure 7 and *BeeHive* introduces more cost. But when it drops to 30%, the per-hour cost of *BeeHive* on Lambda is smaller than the always-on burstable configuration. When it reaches 10%, *BeeHive* can achieve 3.47× cost reduction with Lambda (2.08× for OpenWhisk). The Other two applications show similar results on the 10% burst ratio: the Lambda configuration reaches 4.33× and 2.89× cost reduction for blog and thumbnail (2.60× and 3.47× on OpenWhisk). The results suggest that *BeeHive* would be cost-effective especially when the frequency of bursts is relatively low.

When compared to other on-demand scaling methods (EC2 on-demand and Fargate), the cost of *BeeHive* is always higher since it takes more computation resources due to the offloading overhead. However, the additional cost brings a faster reaction to request burst. Besides, the cost can be further eliminated by combining *BeeHive* with other scaling solutions, as discussed in Section 5.7.

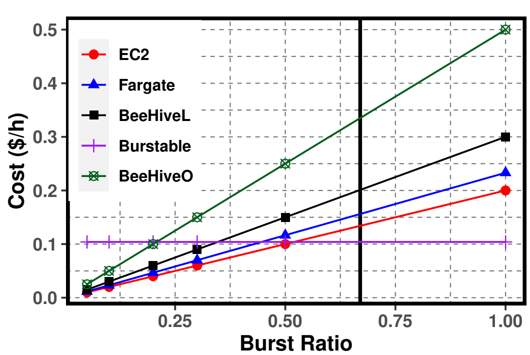


Figure 9: Cost with various burst ratios

Table 4: Minimal tail latency under a fixed throughput

Scaling solutions	thumbnail	pybbs	blog
Vanilla	41.41	34.77	26.72
BeeHiveO	41.99	43.81	29.69
BeeHiveL	41.00	68.30	42.56

5.5 Performance under Various SLOs

Since the tail latency of *BeeHive* increases, we also study how it meets the latency requirements (specified by service-level objects

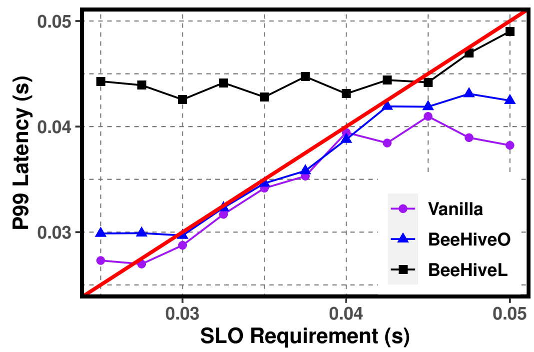


Figure 10: Tail latency under various SLOs

or SLOs). Table 4 shows the minimal tail latency of *BeeHive* under a fixed throughput (50, 170, 130 for thumbnail, pybbs, and blog). Compared with vanilla JVMs running on m4.xlarge instances (the same baseline for throughput analysis), the best achieved p99 latency is 12.8% larger on average for OpenWhisk (51.6% for Lambda). Figure 10 further shows the tail latency under various SLOs with blog as an example. When the SLO becomes lower, all scaling solutions continuously offload more requests until it is satisfied. However, *BeeHive* fails to meet strict SLOs as the vanilla setting due to its execution overhead.

5.6 Breakdown Analysis

Memory consumption and GC. Since each FaaS instance only handles one request at a time, the memory consumption is relatively small. By using the GC mechanism introduced in Section 4.4, the peak heap memory consumption for each function is restricted and remains stable (about 3MB, 29MB, and 22MB for thumbnail, pybbs, and blog, respectively). The memory consumption on the server side is also moderate: a per-function address mapping table only occupies hundreds of KBs. Besides, the median GC pause time for the FaaS instance is 0.92ms, 2.64ms, and 1.42ms, respectively, which can be overlapped when waiting for the next request from the server (about 3ms) and cause trivial overhead for the end-to-end request latency.

Fallback overhead. Table 5 analyzes the fallback overhead of *BeeHive* when offloading requests to OpenWhisk. Although *BeeHive* induces many fallbacks during shadow execution, the number soon decreases with more invocations. After the first few invocations, the largest average number of fallbacks per invocation is only seven in pybbs, and all of them are synchronization-related. Since fallbacks are rarely triggered, the resulting overhead on Lambda is only slightly worse (e.g., 6.29ms for pybbs). The number of updated objects is much larger than that of synchronization fallbacks because applications guarantee that most shared objects can only be exclusively accessed.

Shadow execution. *BeeHive*'s current implementation only shadows the first invocation to FaaS for each function. The duration for shadow execution on OpenWhisk is 2.50s on average, which mainly contains four parts: initialization, remote fetching for code or data, inter-endpoint synchronization, and normal execution. The initialization part contains launching container instances and a

Table 5: Fallback analysis on OpenWhisk

Metrics (Avg.)	thumbnail	pybbs	blog
Fallbacks	1	7	3
Fallback overhead (ms)	0.51	4.15	1.87
Remote fetching	0	0	0
Synchronized objects	5	88	29
Fallbacks (shadow)	64	1525	348
Remote fetching (shadow)	63	1518	345
Fetching overhead (shadow) (ms)	207.75	695.51	246.60

JVM to execute functions (cold boot), whose duration is similar among applications (about 1s). Note that the time for computing initial closures is also included in this part, but the duration is 133.66ms on average and can fully overlap with the cold boot phase. Since the initial closure is not complete, shadow execution also takes a considerable portion of time to fetch code and data remotely (shown in Table 5). Thanks to the Packageable interface and proxies provided by *BeeHive*, no fallbacks related to native invocations or network communication are triggered. Finally, the overhead for synchronization is also trivial (2.84ms on average), given its low trigger frequency. After shadow execution, the request latency dramatically decreases, which helps to reduce the worse case latency by 6.45× on average and thus mitigate the long tail problem.

5.7 Discussion

Limitations of Semi-FaaS. Although the Semi-FaaS execution model can provide rapid resource provision, it also introduces performance overhead and costs more when bursts frequently happen. Therefore, applications satisfying the following requirements are more suitable for Semi-FaaS. First, the overall execution time should be at least at the millisecond level considering the performance overhead. Second, the number of fallbacks should be restricted during Semi-FaaS execution, which suggests applications should induce infrequent synchronizations, limited remote code and data fetching, and inevitable native fallbacks (e.g., accessing local files). Third, the request burst should not happen frequently so the cost of FaaS execution is acceptable. Finally, the offloading candidate selection mechanism in *BeeHive* can perform better if applications have annotated their critical methods.

Combination of Semi-FaaS and other scaling solutions. *BeeHive* can be further combined with other scaling solutions. As mentioned in Section 3.1, *BeeHive* maintains an offloading ratio to scale in and out. Therefore, applications can scale out with *BeeHive* before on-demand instances are launched. When instances are ready, *BeeHive* can set the ratio to zero to stop offloading to FaaS. With this solution, applications can achieve rapid resource provisioning and less performance overhead when facing bursts.

6 RELATED WORK

6.1 Stateful Support for FaaS

Although FaaS is originally designed for stateless execution, prior work has made proposals to support stateful applications. Crucial [54] and Faasm [58] propose annotating objects for sharing through functions via a distributed data store. Azure Durable Functions [22] enables stateful workflows atop Azure's FaaS platforms. AFT [60] and Beldi [66] provide transactional support for FaaS applications, while Boki [38] further optimizes their performance with a distributed shared log. *BeeHive* also allows stateful applications to execute on FaaS, but with its Semi-FaaS execution model.

6.2 Runtime Optimizations for FaaS

Although appealing for applications with large parallelism and dynamic workload, FaaS also shows disadvantages like large startup time (cold boot) and prohibitive communication overhead. Therefore, prior work has proposed many different solutions to reduce the frequency of cold boots [3, 26, 57, 62], shorten the launch time [10, 23, 33, 50–52, 61], and improve inter-function communication [11, 30, 41, 42, 45].

High-level languages like JavaScript and Java are intensively used in FaaS, which stimulates specialized language runtime support. ReplayableJVM [63] checkpoints an initialized JVM image to accelerate the startup time for Java FaaS functions, while Catalyzer [26] uses a similar mechanism for the Go runtime. Photons [27] allows co-executing multiple functions in the same JVM and improves memory consumption and startup latency. Shredder [67] and CloudFlare [20] use lightweight JavaScript V8 contexts to execute FaaS functions. GraalVM Native Image [21, 65] leverages ahead-of-time compilation to improve the startup time of Java applications, while JWarmup [68] integrates other techniques like class data sharing (CDS) [53] for further optimizations. *BeeHive* also encounters problems like cold boot in FaaS offloading and it proposes solutions like shadow execution to mitigate them.

6.3 Language-Assisted Offloading

Offloading is a general approach to leveraging distributed computation resources. Prior work has relied on programming languages to offload or migrate objects in a distributed environment. Emerald [19] provides distribution support with a uniform object model, where objects can be transparently moved among distributed nodes. Argus [43] introduces actions to allow concurrent execution on distributed objects. Gallifrey [48] proposes *restrictions* to safely share objects among distributed clients. *BeeHive* transparently migrates objects to FaaS for distributed execution, but it is mainly designed for a popular programming language (Java) and requires no modifications to applications. Cloud Haskell [28] and Scala spores [49] help to capture more information before sending closures for remote execution. *BeeHive* has a similar goal, but it further considers capturing native states to avoid fallbacks due to native method invocations.

Another line of work designs the offloading mechanism atop language runtimes due to their ability to track application behaviors with acceptable overhead. JESSICA [44, 69] and cJVM [16] propose a distributed Java virtual machine abstraction so that application threads can be transparently offloaded. Hera-JVM [46] allows task offloading in a heterogeneous environment. MAUI [25] and CloneCloud [24] offload energy-consuming code from mobile devices to servers. COMET [34] has a similar goal, but it builds a DSM model among servers and mobile devices and relies on the Java memory model to support inter-thread synchronizations. *BeeHive* instead aims at the FaaS scenario and proposes a partial, transparent, and dynamic offloading mechanism atop language runtimes.

7 CONCLUSION

This paper presents *BeeHive*, a partial, automatic, and dynamic offloading framework for web applications to leverage FaaS. *BeeHive* automatically extracts fine-grained code snippets from web applications and leverages a fallback-based mechanism to synchronize with the original server. *BeeHive* also conducts a series of optimizations to improve the performance of offloaded functions and provides runtime support in a distributed execution environment. The evaluation result shows that *BeeHive* improves the startup time by up to two orders of magnitude compared with other scaling alternatives.

ACKNOWLEDGMENTS

We sincerely thank the anonymous ASPLOS'23 reviewers for their insightful suggestions. This work was supported in part by the National Natural Science Foundation of China (No. 62172272, 62132014, 61925206). Corresponding author: Mingyu Wu (mingyuwu@sjtu.edu.cn).

REFERENCES

[1] 2015. Target and PayPal Sites Report Problems on Cyber Monday. https://www.nytimes.com/2015/12/01/technology/target-paypal-website-cyber-monday.html.
[2] 2018. Amazon's website crashed as soon as Prime Day began. https://www.theverge.com/2018/7/16/17577654/amazon-prime-day-website-down-deals-service-disruption.
[3] 2019. AWS Lambda announces Provisioned Concurrency. https://aws.amazon.com/cn/about-aws/whats-new/2019/12/aws-lambda-announces-provisioned-concurrency/.
[4] 2019. The Not-So-Straightforward Road from Microservices to Serverless. https://www.infoq.com/presentations/microservices-to-serverless/.
[5] 2021. HikariCP: A solid, high-performance, JDBC connection pool at last. https://github.com/brettwooldridge/HikariCP.
[6] 2021. MyBatis. https://mybatis.org/mybatis-3/.
[7] 2021. MyBatis-Plus: Born to Simplify Development. https://baomidou.com/en/.
[8] 2021. Pybbs: Better use of Java development community (forum). https://github.com/tomoya92/pybbs.
[9] 2021. Springblog: A simple blogging system implemented with Spring Boot + Hibernate + MySQL + Bootstrap4. https://github.com/Raysmond/SpringBlog.
[10] 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)*. 419–434.
[11] 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 *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 923–935. http://dl.acm.org/citation.cfm?id=3277355.3277444
[12] Alibaba Cloud. 2021. Function Compute. https://www.alibabacloud.com/product/function-compute.
[13] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static Analysis of Java Enterprise Applications: Frameworks and Caches, the Elephants in the Room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 794–807. https://doi.org/10.1145/3385412.3386026
[14] Apache. 2020. Apache OpenWhisk runtimes for java. https://github.com/apache/openwhisk-runtime-java.
[15] Apache OpenWhisk. 2020. Apache OpenWhisk - Open Source Serverless Cloud Platform. https://openwhisk.apache.org/.
[16] Yariv Aridor, Michael Factor, and Avi Teperman. 1999. cJVM: A Single System Image of a JVM on a Cluster. In *Proceedings of the International Conference on Parallel Processing 1999, ICPP 1999, Wakamatsu, Japan, September 21-24, 1999*. IEEE Computer Society, 4–11. https://doi.org/10.1109/ICPP.1999.797382
[17] AWS. 2020. AWS Lambda. https://aws.amazon.com/lambda/.
[18] AWS. 2022. Amazon EC2 Reserved Instances Pricing. https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/.
[19] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. 1987. Distrbution and abstract types in emerald. *IEEE transactions on software engineering* 1 (1987), 65–76. https://doi.org/10.1109/TSE.1987.232836
[20] Zack Bloom. 2018. Cloud Computing without Containers. https://blog.cloudflare.com/cloud-computing-without-containers/.
[21] Daniele Bonetta. 2018. GraalVM: Metaprogramming inside a Polyglot System (Invited Talk). In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection* (Boston, MA, USA) *(META 2018)*. Association for Computing Machinery, New York, NY, USA, 3–4. https://doi.org/10.1145/3281074.3284935
[22] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable Functions: Semantics for Stateful Serverless. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 133 (oct 2021), 27 pages. https://doi.org/10.1145/3485510
[23] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698
[24] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. Association for Computing Machinery, New York, NY, USA, 301–314. https://doi.org/10.1145/1966445.1966473
[25] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (San Francisco, California, USA) *(MobiSys '10)*. Association for Computing Machinery, New York, NY, USA, 49–62. https://doi.org/10.1145/1814433.1814441
[26] 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512
[27] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: Lambdas on a Diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 45–59. https://doi.org/10.1145/3419111.3421297
[28] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. 2011. Towards Haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*. 118–129. https://doi.org/10.1145/2034675.2034690
[29] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, Berkeley, CA, USA, 475–488. http://dl.acm.org/citation.cfm?id=3358807.3358848
[30] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 363–376.
[31] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/3297858.3304013
[32] Google. 2020. Cloud Functions - Google Cloud. https://cloud.google.com/functions/.
[33] Google. 2020. gvisor: A container sandbox runtime focused on security, efficiency, and ease of use. https://gvisor.dev/.
[34] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. 2012. {COMET}: Code Offload by Migrating Execution Transparently. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 93–106.
[35] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. The Java Language Specification, Java SE 8 Edition. (2014).
[36] IBM. 2020. IBM Cloud Functions. https://www.ibm.com/cloud/functions.
[37] IBM Developer. 2022. About Apache OpenWhisk. https://developer.ibm.com/components/apache-openwhisk/.
[38] 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

[39] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery Berger, and Christos Kozyrakis (Eds.). ACM, 152–166. https://doi.org/10.1145/3445814.3446701

[40] Zewen Jin, Yiming Zhu, Jiaan Zhu, Dongbo Yu, Cheng Li, Ruichuan Chen, Istemi Ekin Akkus, and Yinlong Xu. 2021. Lessons Learned from Migrating Complex Stateful Applications onto Serverless Platforms. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, China) *(APSys '21)*. Association for Computing Machinery, New York, NY, USA, 89–96. https://doi.org/10.1145/3476886.3477510

[41] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 789–794. http://dl.acm.org/citation.cfm?id=3277355.3277431

[42] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 427–444. http://dl.acm.org/citation.cfm?id=3291168.3291200

[43] Barbara Liskov. 1988. Distributed Programming in Argus. *Commun. ACM* 31, 3 (mar 1988), 300–312. https://doi.org/10.1145/42392.42399

[44] Matchy J.M. Ma, Cho-Li Wang, and Francis C.M. Lau. 2000. JESSICA: Java-Enabled Single-System-Image Computing Architecture. *J. Parallel and Distrib. Comput.* 60, 10 (2000), 1194–1222. https://doi.org/10.1006/jpdc.2000.1650

[45] 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 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 285–301. https://www.usenix.org/conference/atc21/presentation/mahgoub

[46] Ross McIlroy and Joe Sventek. 2010. Hera-JVM: A Runtime System for Heterogeneous Multi-Core Architectures. (2010), 205–222. https://doi.org/10.1145/1869459.1869478

[47] Microsoft. 2020. Microsoft Azure Functions. https://azure.microsoft.com/services/functions/.

[48] Mae Milano, Rolph Recto, Tom Magrino, and Andrew C. Myers. 2019. A Tour of Gallifrey, a Language for Geodistributed Programming. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA (LIPIcs)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 136. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:19. https://doi.org/10.4230/LIPIcs.SNAPL.2019.11

[49] Heather Miller, Philipp Haller, and Martin Odersky. 2014. Spores: A Type-Based Foundation for Closures in the Age of Concurrency and Distribution. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science)*, Richard E. Jones (Ed.), Vol. 8586. Springer, 308–333. https://doi.org/10.1007/978-3-662-44202-9_13

[50] Anup Mohan, Harshad S. Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile Cold Starts for Scalable Serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*, Christina Delimitrou and Dan R. K. Ports (Eds.). USENIX Association. https://www.usenix.org/conference/hotcloud19/presentation/mohan

[51] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Pipsqueak: Lean Lambdas with Large Libraries. In *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*, Aibek Musaev, João Eduardo Ferreira, and Teruo Higashino (Eds.). IEEE Computer Society, 395–400. https://doi.org/10.1109/ICDCSW.2017.32

[52] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-optimized Containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 57–69. http://dl.acm.org/citation.cfm?id=3277355.3277362

[53] OpenJDK. 2018. JEP 310: Application Class-Data Sharing. https://openjdk.java.net/jeps/310

[54] Daniel Barcelona Pons, Marc Sánchez Artigas, Gerard París, Pierre Sutra, and Pedro García López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference, Middleware 2019, Davis, CA, USA, December 9-13, 2019*. ACM, 41–54. https://doi.org/10.1145/3361525.3361535

[55] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 193–206.

[56] Qubole. 2017. Spark-on-Lambda. https://github.com/qubole/spark-on-lambda/.

[57] Mohammad Shahrad, Rodrigo Fonseca, Iñigo 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 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[58] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[59] Spring. 2021. Spring makes Java productive. https://spring.io/.

[60] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A fault-tolerance shim for serverless computing. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 15:1–15:15. https://doi.org/10.1145/3342195.3387535

[61] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. 2020. Particle: ephemeral endpoints for serverless networking. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi (Eds.). ACM, 16–29. https://doi.org/10.1145/3419111.3421275

[62] Markus Thömmes. 2017. Squeezing the milliseconds: How to make serverless platforms blazing fast.

[63] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, George Candea, Robbert van Renesse, and Christof Fetzer (Eds.). ACM, 39:1–39:16. https://doi.org/10.1145/3302424.3303978

[64] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, Berkeley, CA, USA, 133–145. http://dl.acm.org/citation.cfm?id=3277355.3277369

[65] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: application initialization at build time. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 184:1–184:29. https://doi.org/10.1145/3360610

[66] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 1187–1204. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[67] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 1–12. https://doi.org/10.1145/3357223.3362723

[68] Yifei Zhang, Tianxiao Gu, Xiaolin Zheng, Lei Yu, Wei Kuai, and Sanhong Li. 2021. Towards a Serverless Java Runtime. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 1156–1160. https://doi.org/10.1109/ASE51524.2021.9678709

[69] Wenzhang Zhu, Cho-Li Wang, and Francis C. M. Lau. 2002. JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support. In *2002 IEEE International Conference on Cluster Computing (CLUSTER 2002), 23-26 September 2002, Chicago, IL, USA*. IEEE Computer Society, 381–388. https://doi.org/10.1109/CLUSTR.2002.1137770

Received 2022-07-07; accepted 2022-09-22