BeeHive: Sub-second elasticity for web services with Semi-FaaS execution[*]

Ziming Zhao[†], Mingyu Wu[†§], Jiawei Tang[†], Binyu Zang[†‡], Zhaoguo Wang[†], Haibo Chen[†‡]
{dumplings_ming, mingyuwu, jiawei_tang, byzang, zhaoguowang, haibochen}@sjtu.edu.cn
[†]*Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University*
[§]*Shanghai AI Laboratory*
[‡]*Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

1. Motivation

The dynamic nature of the real-world web environment stimulates strong demand for resource elasticity, i.e., to rapidly scale up and down according to the fluctuated 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 a faster reaction while providing a pay-as-you-go model for cost-efficient computation. Mainstream cloud vendors have provided their own FaaS platforms [2, 5, 11, 12, 16], while prior work has proposed to run various applications with massive parallelism as FaaS functions [8, 9, 10, 14, 17, 18].

Although FaaS seems like a good fit for building elastic web applications, it is mainly designed for short-running, stateless functions and thus encounters challenges when applying to web service scenarios. The major problem for migration is the mismatch between the *stateless* nature of FaaS platforms and the stateful execution of web services. In FaaS, all internal states would be lost when a function finishes its execution. In contrast, web services maintain internal states like user sessions, which should not be abandoned. FaaS does not provide support to manage those states and is thus not suitable for directly executing web services.

Since directly migrating web services is infeasible, an alternative method would be rewriting them into FaaS-friendly components. Nevertheless, this method is also impractical, given the complexity of web applications. Enterprise-level web applications contain tens of thousands of classes [3], which involve various frameworks (like Spring [20]) and dynamically generated auxiliary methods and make rewriting difficult. Therefore, an automatic method is preferred for web applications to leverage the power of FaaS.

[*]This work has been accepted and will appear in ASPLOS'23.

2. Related Work

Prior efforts have proposed stateful support on FaaS platforms for more complicated applications. Crucial [6] and Faasm [19] allow developers to annotate shared objects across functions, and the runtime system would manage them in a distributed data store. Azure Durable Functions [7] provide a new programming model to define stateful workflow atop its own FaaS platform. Another line of work provides transaction semantics for applications [13, 21, 22]. However, their programming models are different from existing stateful applications, so rewriting is still necessary for web services, which is infeasible due to code complexity.

Instead of retrofitting FaaS platforms, Jin et al. [15] manually rewrite smaller web services (namely *micro-services*) for FaaS migration. They find that although the size of micro-services is quite small, the rewriting phase is still non-trivial as they contain complicated states. Recent experiences [1] also show the complexity of code rewriting. Their experiences necessitate an automatic mechanism to support the migration of complex stateful applications.

3. *BeeHive* Design

To automatically apply FaaS in existing stateful web services, we propose a novel execution model called Semi-FaaS. We further proposed a runtime-based offloading framework, *BeeHive*, to realize the Semi-FaaS model.

Semi-FaaS execution model. Given the complicated states inside web services, we do not need to migrate them completely to FaaS; we can migrate only a part of them instead. This work thus proposes the *Semi-FaaS*, a new execution model for complicated applications (like web services) to embrace FaaS. As illustrated in Figure 1, Semi-FaaS combines the execution model of traditional monolithic services and FaaS: it extracts the time-consuming part of monolithic web services and migrates them for FaaS execution while executing the rest and maintaining states on the monolith side (referred to as the *server*). Thanks to the Semi-FaaS model, web applications can be executed on FaaS without any code modifications.

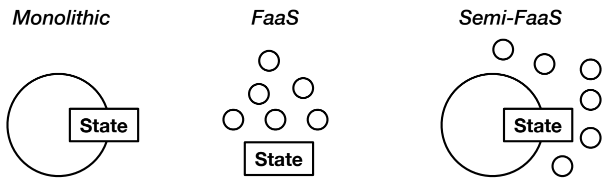

Figure 1: The Semi-FaaS execution model

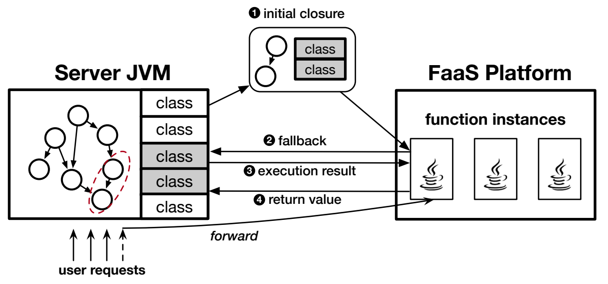

Figure 2: The workflow of *BeeHive*'s offloading mechanism

***BeeHive* Overview.** Although the Semi-FaaS model seems appealing for existing web applications, the extraction phase is not simple due to the code complexity. Fortunately, web services are usually written in high-level languages like Java, JavaScript, and Go. Those languages run atop managed runtimes, which are suitable for state management and code extraction by gaining application semantics during their execution. Therefore, we build our Semi-FaaS execution model atop managed runtimes to dynamically extract code snippets for FaaS execution. Our *BeeHive* framework realizes the Semi-FaaS model atop Java virtual machines (JVMs) to support enterprise-level Java web services. 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: an offloading-based mechanism for Semi-FaaS execution and an underlying runtime system to support consistent and efficient execution among multiple endpoints.

Offloading-based Semi-FaaS with *BeeHive*. The offloading mechanism is the core of *BeeHive*'s Semi-FaaS execution. As shown in Figure 2, *BeeHive* mainly contains two components: long-running servers and FaaS platforms. When facing request bursts, BeeHive controls servers to proactively offload a part of its workload to FaaS platforms for execution in FaaS functions while keeping the rest still handled by the server.

The unit for offloading is called a *closure*, which contains code and data likely to be used by FaaS execution, according to statistics collected by *BeeHive*'s language runtime. Note that the initial closure sent to the FaaS platform is incomplete as rarely accessed code and data are not included. To this end, *BeeHive* provides a *fallback* mechanism. Functions send fallback requests to the server when accessing missing code or data, while the server handles requests and sends the results back so that the offloaded functions can resume their execution.

Unfortunately, fallbacks also introduce considerable overheads due to the coordination between FaaS functions and the server. Therefore, *BeeHive* categorizes the fallbacks and provides corresponding optimizations to reduce the fallback count. For native invocations, *BeeHive* provides the *packageable* abstraction to pack native states into closures to eliminate related fallbacks. For network connections with other services, *BeeHive* manages them with proxies and allows FaaS functions to process network packets on the server's behalf directly. For inevitable missing code and data fallbacks, we observe fallbacks frequently occur at the beginning of FaaS execution and provide a *shadow execution* mechanism to warm up without introducing visible overhead to users' real requests.

The underlying runtime system. The runtime system in *BeeHive* is responsible for state management across multiple endpoints. It enables distributed object sharing by differentiating remote references in each FaaS instance and relies on the memory model in Java (JMM) to synchronize accesses on shared states stored in the original server. Besides, *BeeHive* should choose suitable methods for offloading during runtime. It treats methods containing business logic (i.e., request handler) as candidates according to existing annotations required by frameworks (e.g., Spring), and instruments profiling code to pick time-consuming ones for offloading. Further, *BeeHive* also provides efficient memory management (i.e., garbage collection) for all endpoints and an optional fault tolerance mechanism to recover from FaaS function failures.

4. Implementation and Evaluation

BeeHive is implemented atop the HotSpot JVM of OpenJDK 8u265-ga. We evaluated *BeeHive* with a small web service (*image processing*) and two enterprise-level Java web applications (*pybbs* and *SpringBlog*). We use both the open-source (OpenWhisk [4] on AWS EC2) and the commercialized FaaS platform (AWS Lambda [5]) for testing and leveraging other commercialized scaling methods in AWS as baselines.

Our evaluation shows that *BeeHive* can scale to higher throughput (9.41x) with the Semi-FaaS execution model while resulting in a 7.14% throughput drop given the same computation resource. When facing a request burst, *BeeHive* can scale out much faster than AWS EC2 on-demand instances (11.25x) while causing a 15.0% tail latency slowdown. Equipped with the instance caching mechanism in AWS lambda, *BeeHive* can tackle request bursts in less than a second (two orders of magnitude faster than other scaling solutions). Compared with reserving idle instances, *BeeHive* can reduce the financial cost when request bursts occur infrequently. Please refer to our full paper for other evaluation results.

5. 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, *BeeHive* can perform better if developers have annotated their critical methods as offloading candidates according to the application semantics.

Combination of Semi-FaaS and other scaling solutions. *BeeHive* can be further combined with other scaling solutions to reduce its overhead. *BeeHive* maintains an *offloading ratio* to control the number of offloaded and local-executed request, and thus can scale in and out by setting the ratio. 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 both rapid resource provisioning (provided by Semi-FaaS) and less performance overhead (provided by other scaling methods) when facing bursts.

6. Conclusion

This paper presents *BeeHive*, an offloading framework for web applications to leverage FaaS. *BeeHive* automatically extracts 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 for distributed execution. The evaluation result shows that *BeeHive* can automatically and rapidly scale out with FaaS execution.

References

[1] The not-so-straightforward road from microservices to serverless. https://www.infoq.com/presentations/microservices-to-serverless/, 2019.

[2] Alibaba Cloud. Function compute. https://www.alibabacloud.com/product/function-compute, 2021.

[3] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 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*, pages 794–807, 2020.

[4] Apache OpenWhisk. Apache openwhisk - open source serverless cloud platform. https://openwhisk.apache.org/, 2020.

[5] AWS. Aws lambda. https://aws.amazon.com/lambda/, 2020.

[6] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, pages 41–54, 2019.

[7] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S Meiklejohn. Durable functions: semantics for stateful serverless. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, 2021.

[8] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 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*, USENIX ATC '19, pages 475–488, Berkeley, CA, USA, 2019. USENIX Association.

[9] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 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)*, pages 363–376, 2017.

[10] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 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*, pages 3–18. ACM, 2019.

[11] Google. Cloud functions - google cloud. https://cloud.google.com/functions/, 2020.

[12] IBM. Ibm cloud functions. https://www.ibm.com/cloud/functions, 2020.

[13] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 691–707, 2021.

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

[15] Zewen Jin, Yiming Zhu, Jiaan Zhu, Dongbo Yu, Cheng Li, Ruichuan Chen, Istemi Ekin Akkus, and Yinlong Xu. Lessons learned from migrating complex stateful applications onto serverless platforms. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 89–96, 2021.

[16] Microsoft. Microsoft azure functions. https://azure.microsoft.com/services/functions/, 2020.

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

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

[19] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, 2020.

[20] Spring. Spring makes java productive. https://spring.io/, 2021.

[21] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E Gonzalez, Joseph M Hellerstein, and Jose M Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[22] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1187–1204, 2020.