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

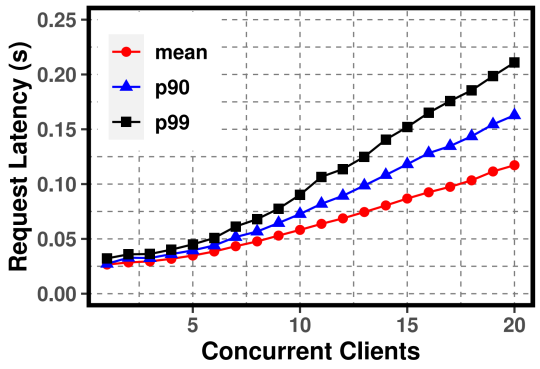
Ziming Zhao, Mingyu Wu, Jiawei Tang, Binyu Zang, Zhaoguo Wang, Haibo Chen

Shanghai Jiao Tong University

Web Application and Dynamic Workload

- **Request bursts are long-term enemies for web applications**

Slower response



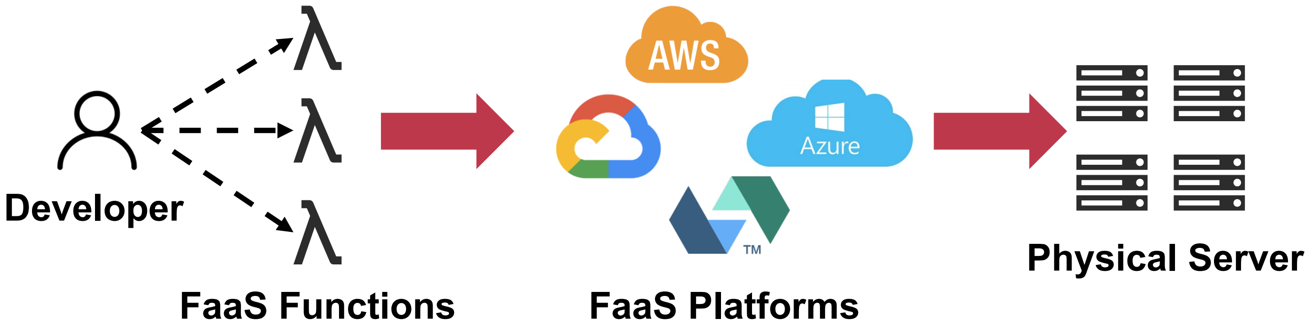
Unavailability

- **Dynamic workload demands <u>rapid</u> and <u>cost-efficient</u> burst handling**
  - Reserving computation resource -> high cost
  - On-demand scaling -> slow response

https://www.theverge.com/2018/7/16/17577654/amazon-prime-day-website-down-deals-service-disruption
https://www.nytimes.com/2015/12/01/technology/target-paypal-website-cyber-monday.html

Serverless Computing

- **Serverless computing (e.g., Function-as-a-Service) is a new cloud-computing paradigm**
  - Developers write fine-grand functions and submit them to FaaS platforms
  - FaaS platforms invoke functions on-demand and bill developers according to resource usage and execution time
  - Rapid auto-scaling, pay-as-you-go billing model, no management labor



Scaling with FaaS

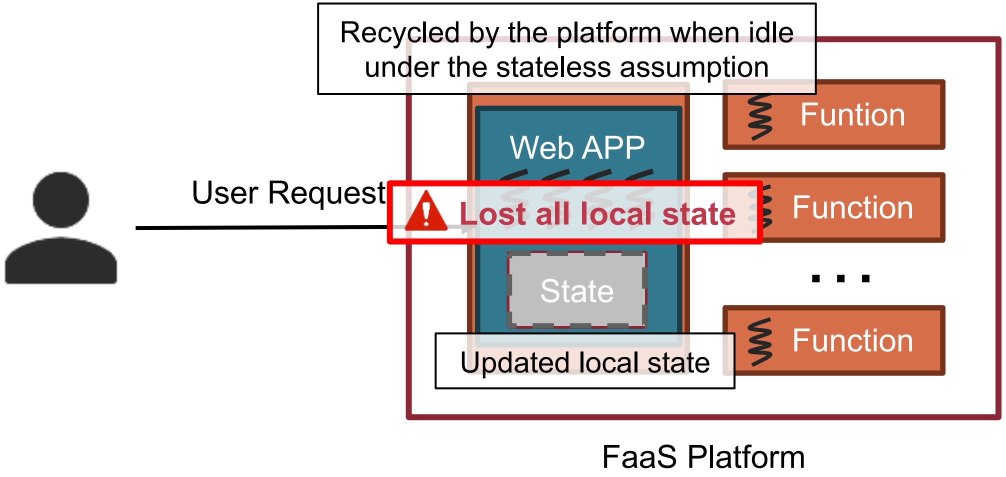
- **FaaS provides rapid-scaling and cost-efficient computing resources for web applications to handle request bursts**
  - Provide more computation resources on demand rapidly **(rapid-scaling)**
  - Fine-grand configuration and billing to eliminate the cost **(cost-efficient)**

Scaling solution	Min. Running Time	Conf. & Bill granularity	Preparation Time
Reserve resource (AWS Reserved Burstable)	1 year	GB, Years	--
On-demand virtual machine (AWS On-demand EC2)	1 min	GB, Seconds	~40s
On-demand container (AWS Fargate & ECS)	1 min	GB, Seconds	~40s
FaaS (AWS Lambda)	**1 ms**	**MB, Milliseconds**	**<1s**

Problem: How to run existing web applications with FaaS functions

Strawman 1: Direct Execution

- **Directly run existing web applications in FaaS**
- **Stateful applications vs. stateless functions**
  - FaaS platform manages functions under the stateless assumption
  - Web applications contain complex local state like user session
  - May cause unrecoverable state loss



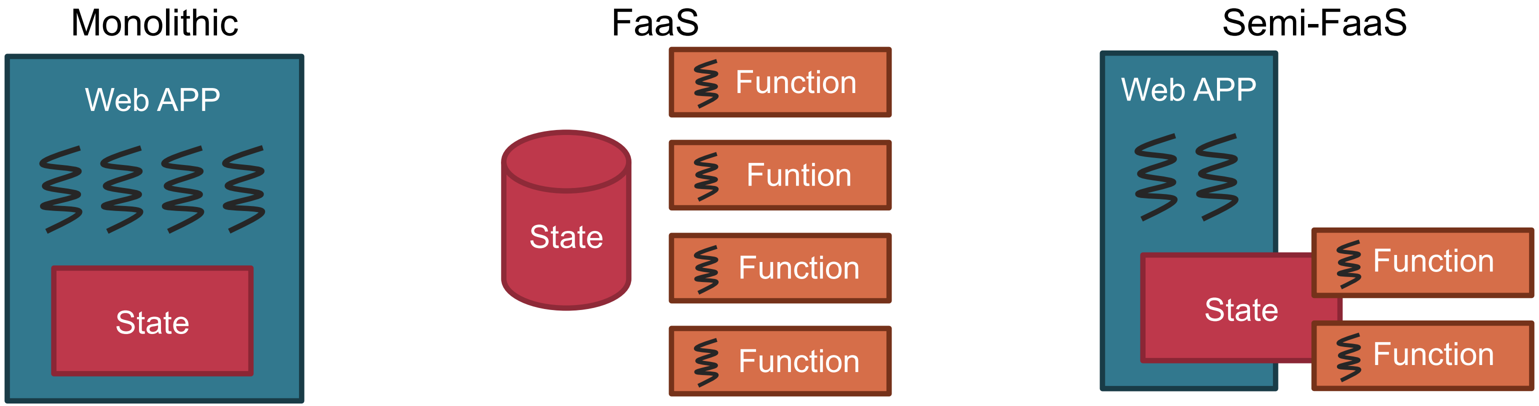
*All data except for evaluation are tested in *pybbs*, an open-source online forum web application

Strawman 2: Application Refactor

- **Refactor (part of) existing application to fit FaaS functions**
- **Manual rewriting**
  - Most code (99.6% of the jar file) are framework (e.g., spring) code
  - Tightly coupled user code and framework code
  - Too complex to manually refactor code
- **Static analysis**
  - Java is a highly dynamic language, especially in the web application case
  - Deep invocation depth (>20), complex polymorphism (31 implementations for 1 interface), dynamically generated classes (287 for one request)
  - Hard to perform static analysis

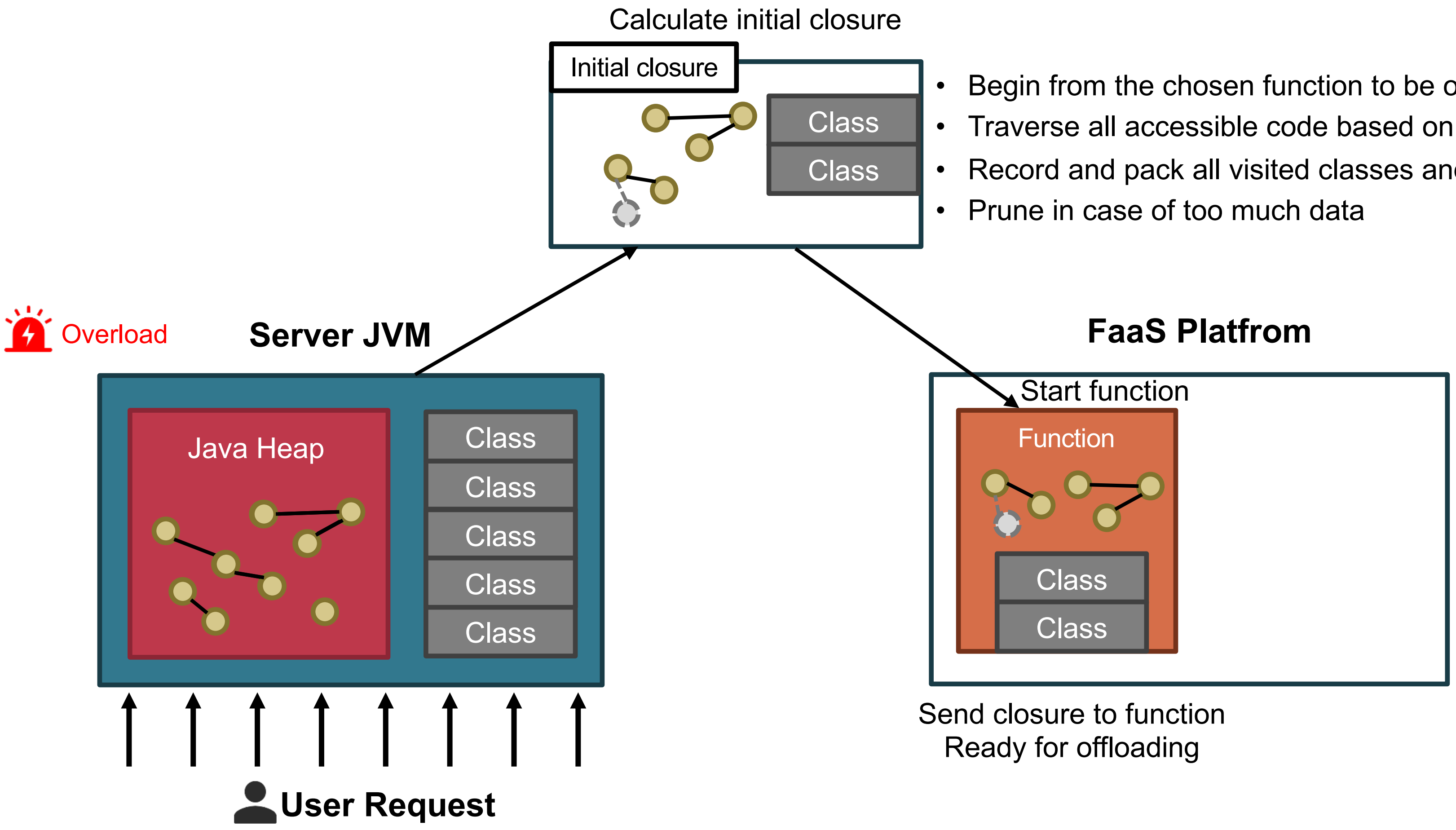
Our Solution: Offloading-based Semi-FaaS

- **Semi-FaaS: automatically extract time-consuming code snippets and offload their execution to FaaS at runtime**
  - Partial: keep the state at the *server*, extract and offload part of the application to *FaaS* **(direct execution)**
  - Automatic: atomically slice and run logic with *FaaS* **(manual rewrite)**
  - Dynamic: Analyze at runtime based on dynamic profiling **(static analysis)**



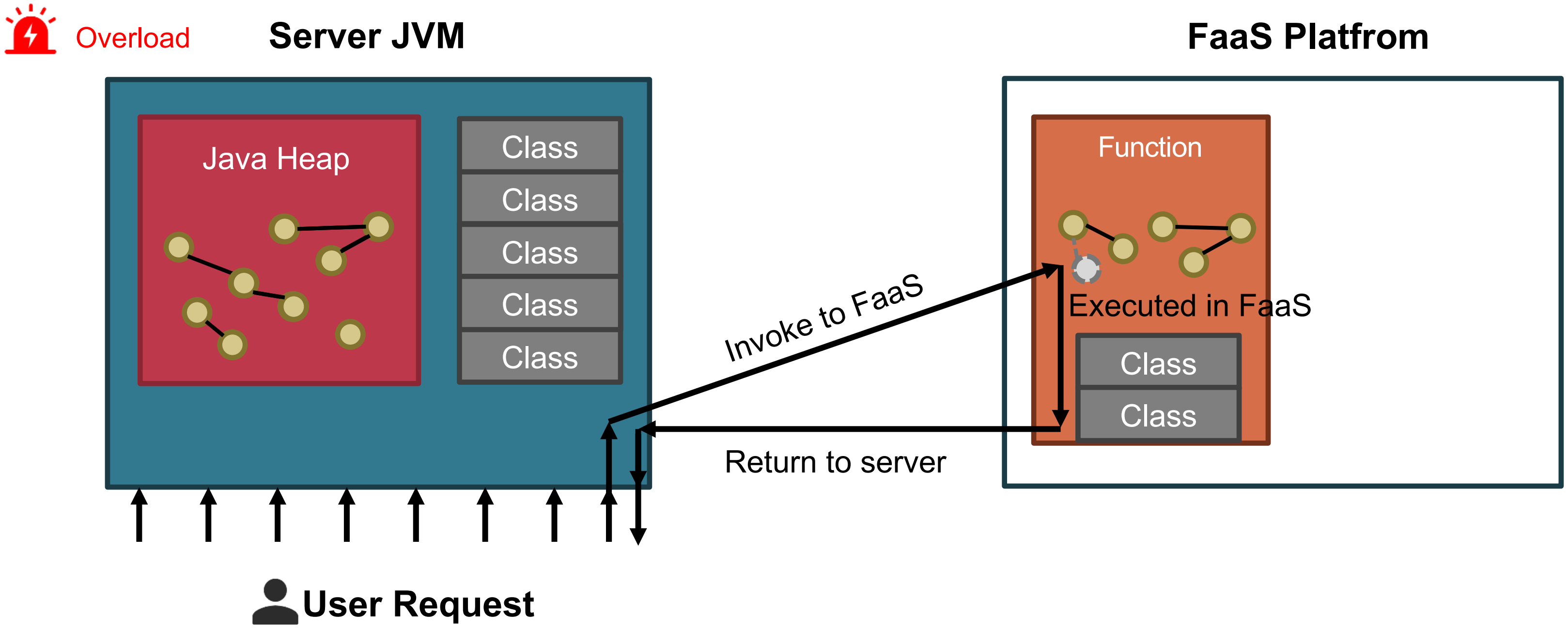
*We implemented Semi-FaaS on Java Virtual Machine (JVM), but Semi-FaaS be extended to other language runtimes

Semi-FaaS Execution

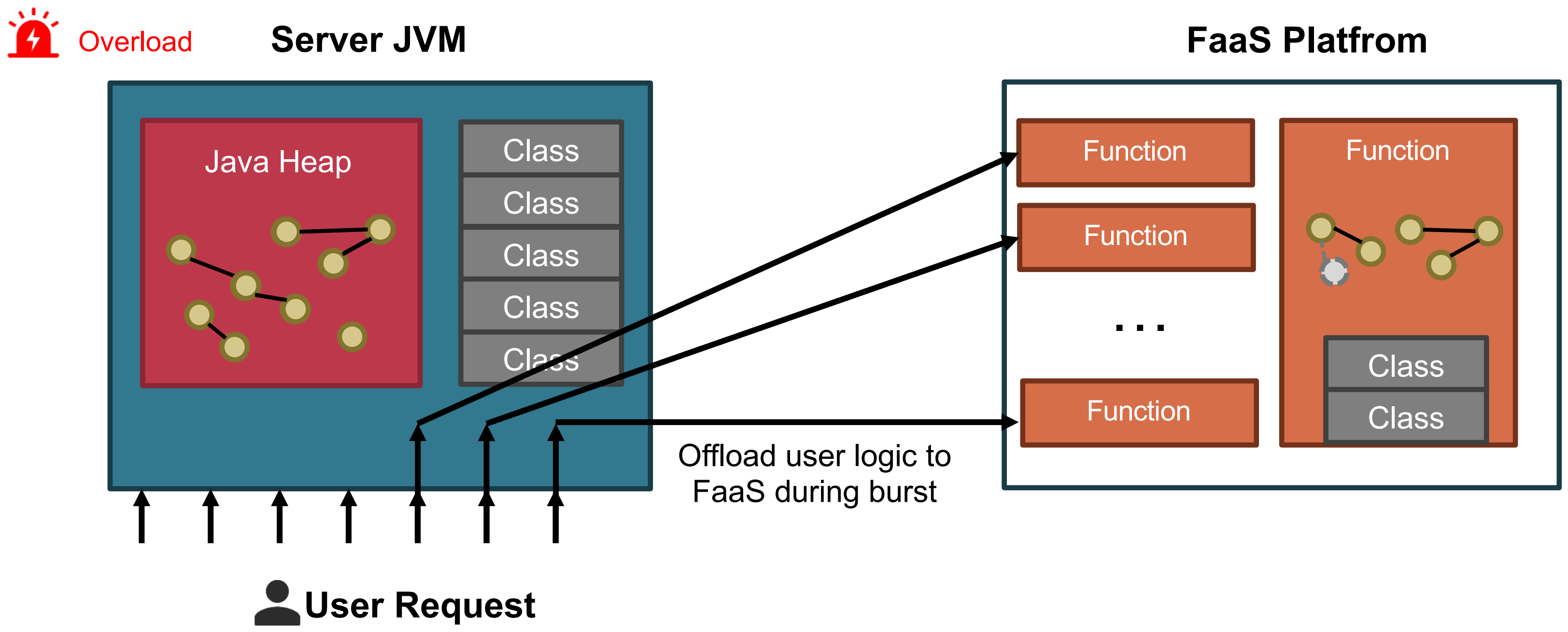


- Begin from the chosen function to be offloaded
- Traverse all accessible code based on dynamic profiling
- Record and pack all visited classes and data
- Prune in case of too much data

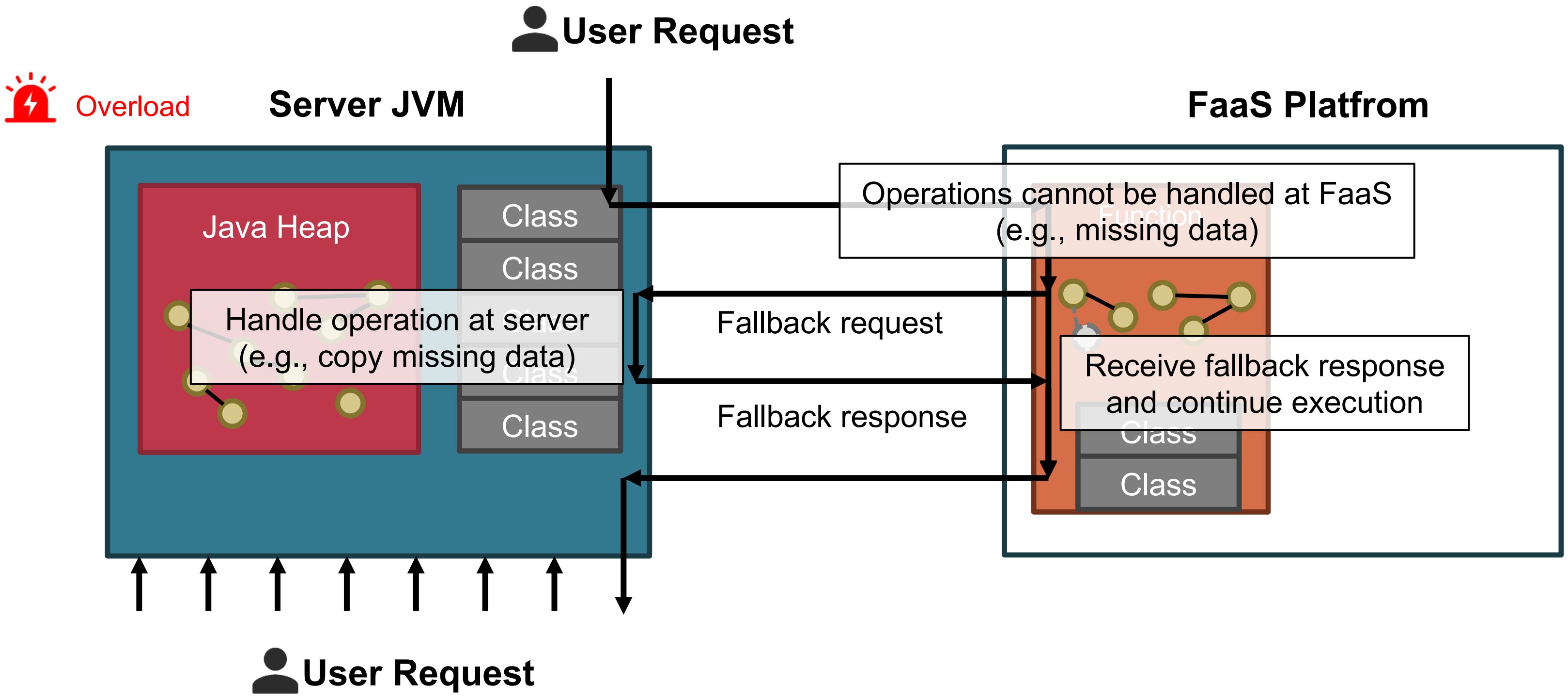
Semi-FaaS Execution



Semi-FaaS Execution



Fallback-based Offloaded Execution



Problem: Fallbacks slow down offloaded execution

Frequent fallbacks hurt performance

- Native invocation
- Network access
- Missing code or data

```java
public interface Packageable {
    public void pack();
    public void unpack();
}
```

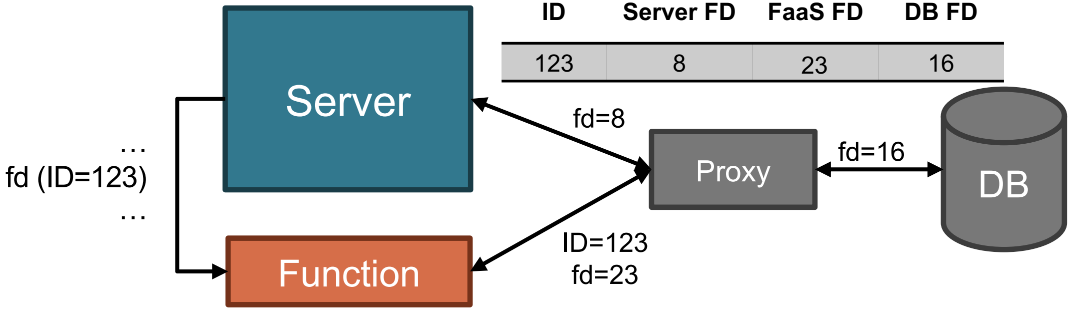
Handling Native Invoca

- **Web applications rely on native invocation heavily**
  - For reflection, access system resources, acceleration, etc.
  - E.g., a simple request can trigger 220k+ native invocation
  - Native invocations are not offloadable since they may rely on the hidden native state (e.g., JVM-internal state, OS-related state)
- **Packageable: a new interface to pack hidden native state**
  - Define how to pack/unpack hidden native state
  - Modify the JDK library to implement packageable interface for specific classes

Proxy-based Connection Management

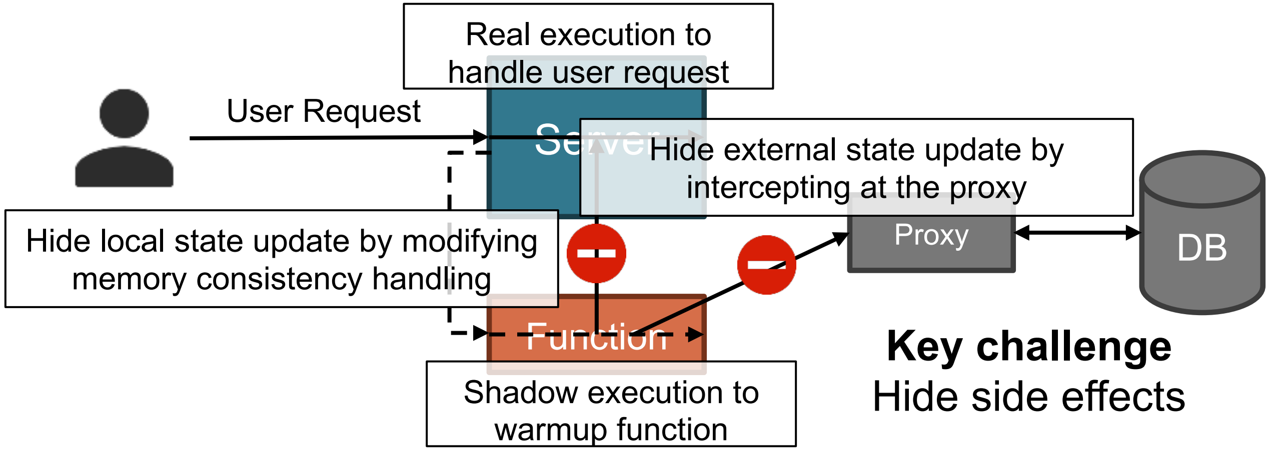
- **Network operation in web applications**
  - Web application depends on the network to access external services (e.g., DB)
  - A simple request requires 80+ DB access
  - Network connections are hard to migrate due coupled with OS
- **Proxy-based network connection migration**

ID	Server FD	FaaS FD	DB FD
123	8	23	16



Shadow Execution

- **Incomplete initial closure introduces frequent fallback**
  - Dynamic nature of web applications makes it hard to traverse all runnable code
  - Closure completes itself with fallbacks during execution
- **Shadow execution to hide overhead during warmup**



Key challenge
Hide side effects

The Beehive Runtime

- **A modified JVM supporting semi-FaaS execution, with**
  - Offload function selection based on runtime profiling data
  - Fallback detecting and handling
  - Memory consistent among endpoints following Java Memory Model
  - Memory management among endpoints
  - Optional fault tolerance mechani
- **Enables unmodified applicat**
  changing their underline JV


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

ABSTRACT

(ASPLOS '23), March 25 – March 29, 2023, Vancouver, Canada. ACM, New

Please checkout our paper :)

Experimental Setup

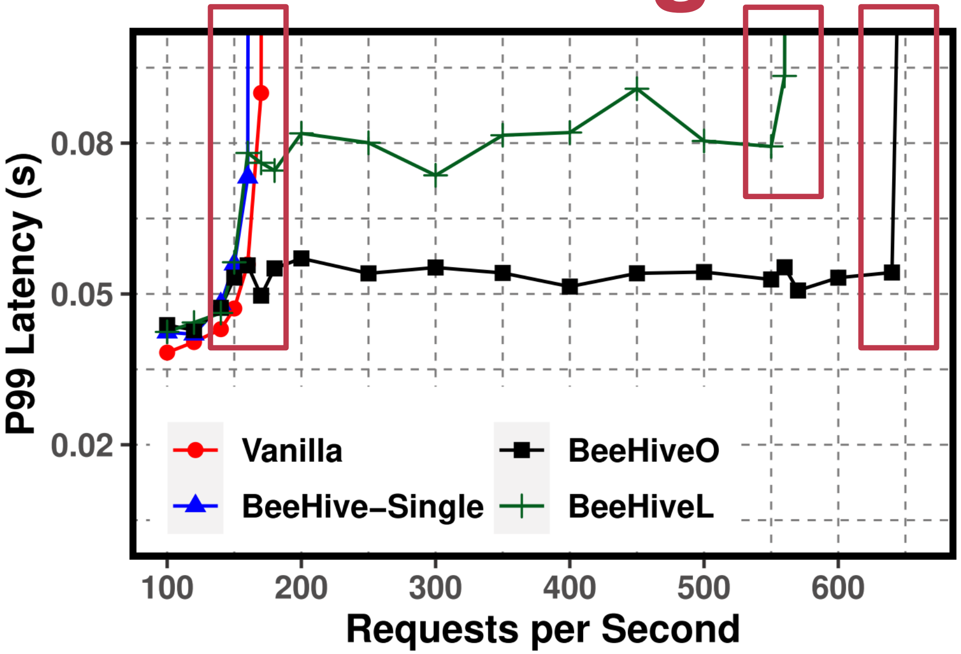
- **Environment: AWS cloud**
  - DB: m4.10xlarge (40 vCPUs/2.40GHz, 160GB DRAM) EC2 instance
  - Server: m4.xlarge (4 vCPUs/2.30GHz, 16GB DRAM) EC2 instance
- **Applications**
  - Thumbnail: micro-benchmark making thumbnail of images
  - Pybbs: open-source forum application with 24692 classes
  - Springblog: open-source blog application 18493 classes

(All mentioned data are average of all three applications by default)

Experimental Setup

- **Scaling methods**
  - **Burstable**: Reserved resource (reserved burstable EC2 instance)
  - **EC2**: On-demand VM (on-demand EC2 instance)
  - **Fargate**: On-demand container (AWS Fargate)
  - **BeehiveO**: Local FaaS platform (functions running on EC2 cluster managed by OpenWhisk)
  - **BeehiveL**: Commercial FaaS platform (functions running on AWS Lambda)

Auto-Scaling



Centralized server acting as dispatcher and state manager becomes the bottleneck

Vanilla & **Beehive-Single**: No scaling

BeehiveO: Scale with Functions running on sufficient m4.large instances managed by OpenWhisk

BeehiveL: Scale with Functions running on sufficient AWS Lambda with 1GB memory

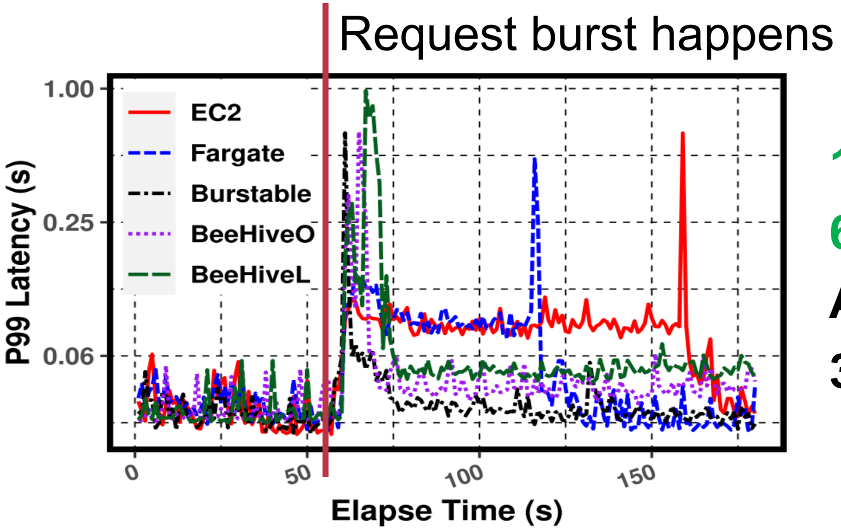
Beehive **atomically scales** to higher throughput **(9.41x (O) & 9.11x (L))**

Lower throughput (worst 7.14%) with the same resource due to management overhead

Sringblog case for better presentation, refer to our full paper for performance result of all applications

Fast Scaling

- **Beehive handles burst faster with acceptable overhead**



11.25x(O) & 6.43x(L) faster than scaling with **EC2**

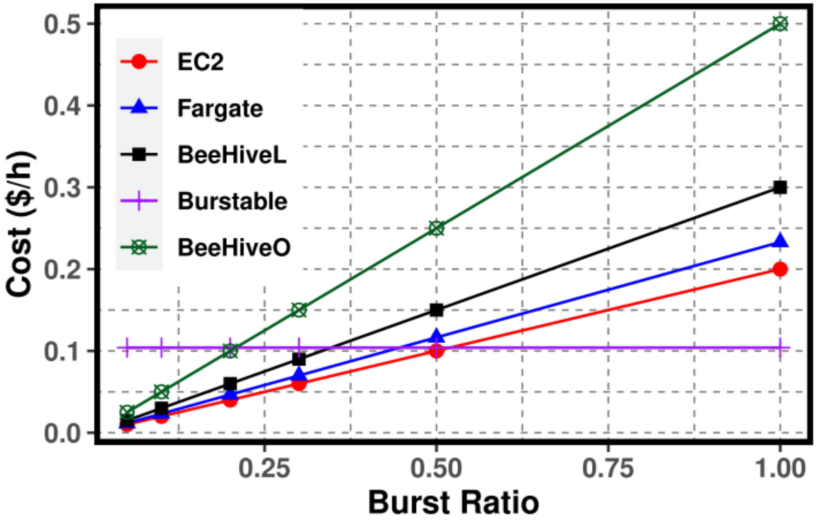
6.32x(O) & 3.61x(L) faster than scaling with **Fargate**

Acceptable tail-latency slowdown (**15.0%(O) & 31.0%(L)** compared with EC2)

Reach stabilized latency in **668.56ms** on AWS Lambda with **function cache** (**two orders of magnitude better**)

On-demand Cost

- **Beehive enjoys on-demand billing provided by FaaS**



When **burst infrequently**, Beehive **costs less** compared with **reserving resources** (**3.56x(L)** at a 10% burst rate)

Beehive **always costs more** compared with other **on-demand scaling methods** due to execution overhead, while reacting to burst faster

Conclusion

- **FaaS is suitable for web applications to handle request burst**
  - Challenging to leverage FaaS by direct execution or code refactoring
- **Semi-FaaS: Fallback-based automatic computation offload at runtime with FaaS**
  - Partial, automatic, dynamic way to leverage FaaS for computation offloading
  - Packageable, network proxy, shadow execution to eliminate performance overhead caused by fallbacks
- **Beehive: Runtime supporting Semi-FaaS execution model**
  - Automatically scale to higher throughput
  - Faster reaction to request burst compared to on-demand scaling
  - Lower cost compared to reserving idle resources in advance

Thanks!

SHANGHAI JIAO TONG UNIVERSITY 1896
IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS