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

Edge System Reading Group @ SEU

Haodong Tian

June 1, 2023.

Introduction – FaaS

- Real-world web environment stimulates demand for resource elasticity.
- Compared to others, FaaS provides:
  - Automatically scales applications in a finer granularity (functions)
  - Shorter reaction time
  - Pay-as-you-go model for cost-efficient computation

Introduction – FaaS

- Prior work has proposed to run various applications as FaaS functions, including video processing, software complication, micro-services, etc.
- FaaS encounters challenges when deploying traditional monolithic web applications.

Motivation – Tackling Request Bursts with FaaS

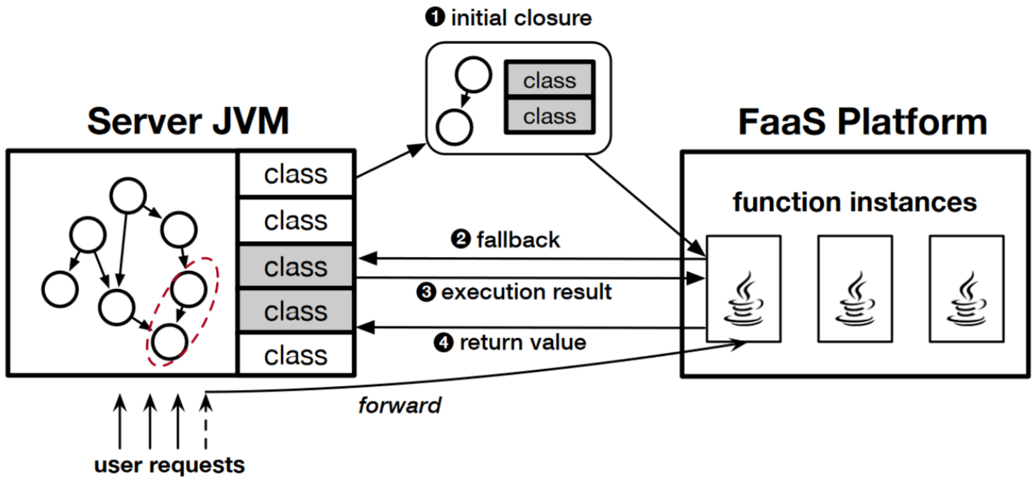
- Request bursts are long-term enemies for web applications.
- Cloud vendors (taking AWS as example) have provided various solutions for resource scaling:
  - Reserved Instance (RI) *high cost*
  - On-demand instance *long instance creation time*
  - Burstable instance *similar to RI but uses a different billing model*
  - Fargate *granularity and billing not so flexible*
  - Lambda (FaaS) *rapid, elastic, and automatic fashion*

Motivation – Initial Approaches of Applying FaaS to Web Applications

- It is not trivial to adapt existing web applications to FaaS platforms.
- Three different methods to migrate an enterprise-level web service into FaaS platform for execution:
  - Method 1: direct execution *violates stateless and lightweight assumptions*
  - Method 2: manual rewriting *too complicated*
  - Method 3: static code analysis *too dynamic to be statically analyzed*
- An execution model for web applications to leverage the power of FaaS should conform to the following principles:
  - Partial. Automatic. Dynamic.

System Design – Offloading-Based Semi-FaaS with BeeHive

- BeeHive mainly contains two parts: long-running servers and FaaS platforms.
- When facing request bursts, BeeHive controls servers to 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).



System Design – Reducing the Performance Overhead of Fallbacks

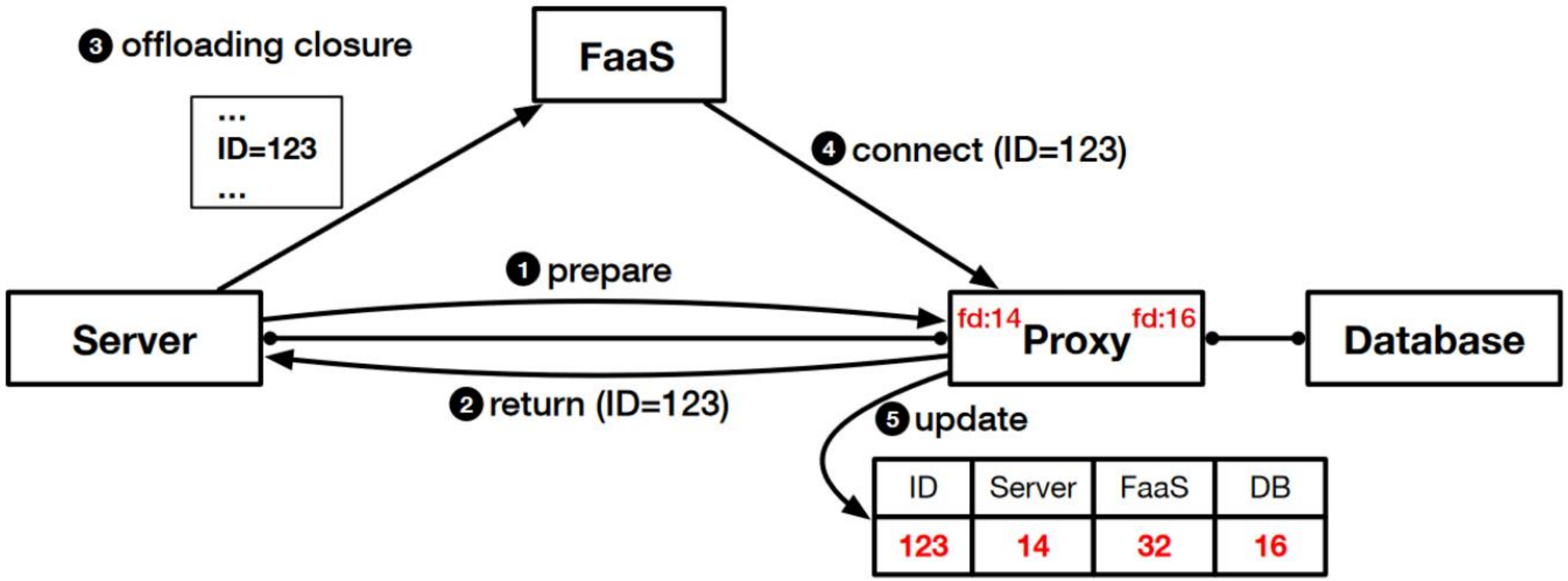
- **Native invocations**
- In web applications, native invocations are heavily used (System.arraycopy, Thread.currentThread, etc) but treated as not offloadable. Returning to servers for handling would cause prohibitive overhead.
- Native methods could be divided into four categories:
  - Pure on-heap operations. *can be directly executed on FaaS*
  - Hidden states. *marshal the hidden states into closure*
  - Network-related. *will be discussed later…*
  - Stateless. (such as *currentThread()*) *can be tagged and executed on FaaS*

System Design – Reducing the Performance Overhead of Fallbacks

- **Stateful connections**
- Web applications contain stateful connections with external services like databases. Those connections cannot be directly offloaded to FaaS platforms.
- The core idea of the proxy-based approach to manage stateful connections is to share a connection to external services between servers and FaaS functions.

System Design – Reducing the Performance Overhead of Fallbacks

- Stateful connections



System Design – Reducing the Performance Overhead of Fallbacks

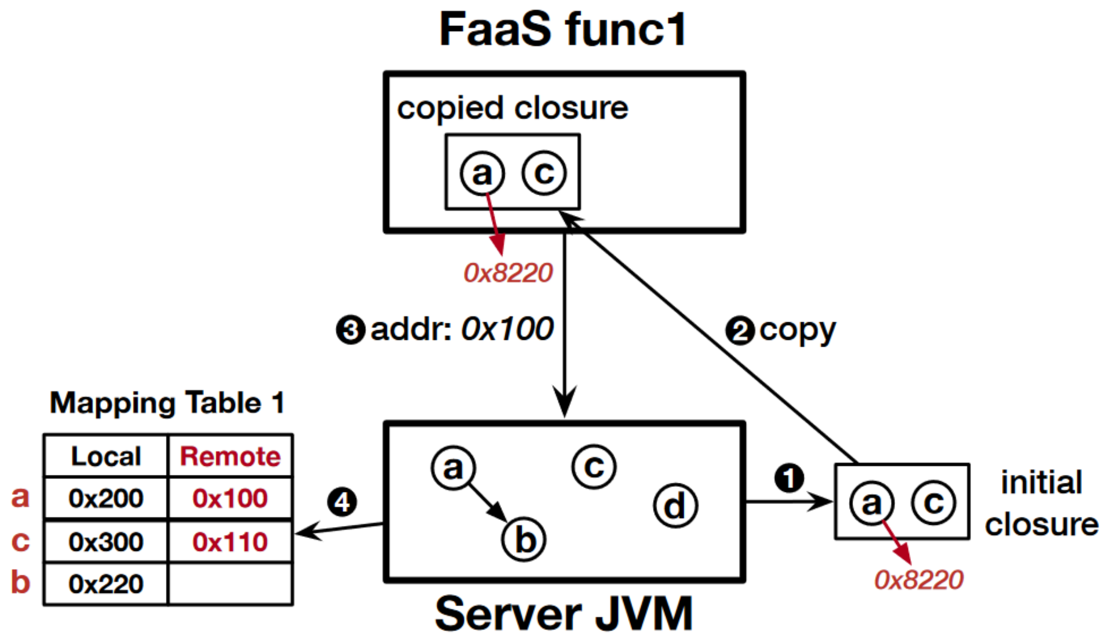
- **Warmup overhead**
- The number of fallbacks is large for the first execution on FaaS, and the FaaS platform needs to establish a new runtime environment for function execution.
- BeeHive proposes shadow execution to hide the warmup overhead from users, which is to process a duplicated user request without introducing observable state modifications.

System Design – 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
  - Memory management

System Design – The BeeHive Runtime System

- **Distributed object sharing**
- When a function is being launched, the server constructs the initial closure to include objects likely to be used by the offloaded function, and mark the references of remote objects as remote references.

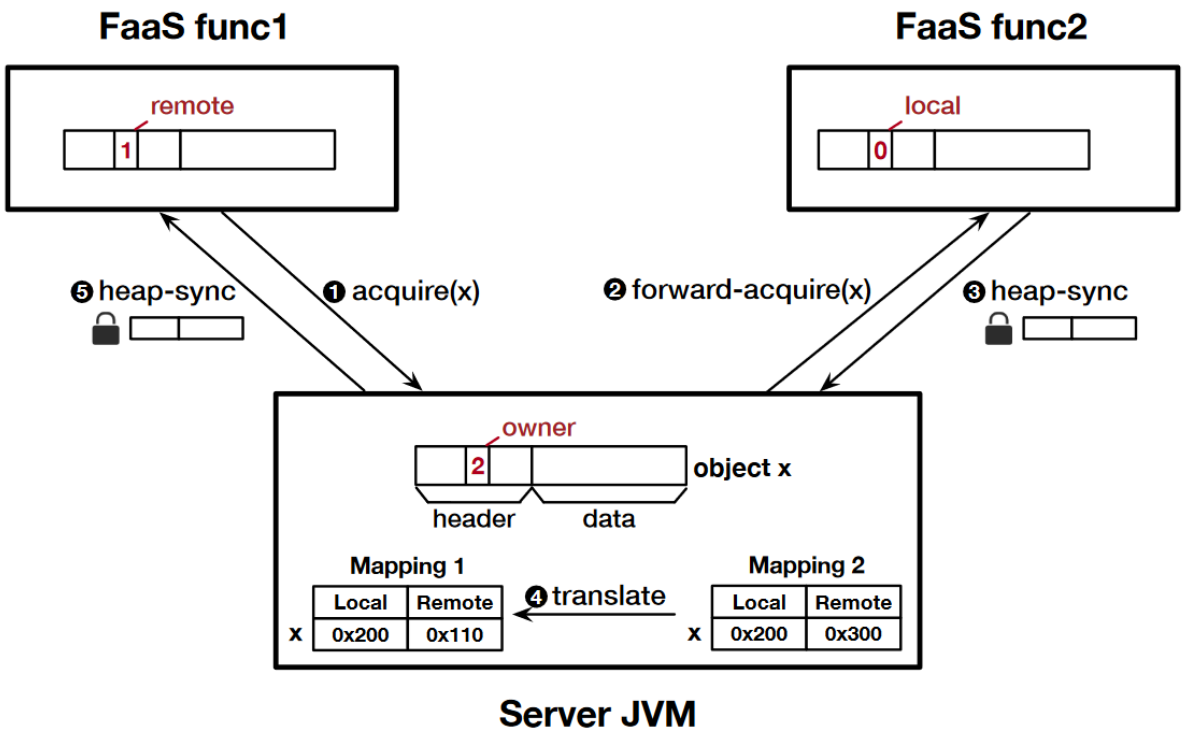


System Design – The BeeHive Runtime System

- **Shared state synchronization**
- BeeHive support state synchronization to ensure consistent execution for multiple FaaS endpoints.
- 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.

System Design – The BeeHive Runtime System

- Shared state synchronization



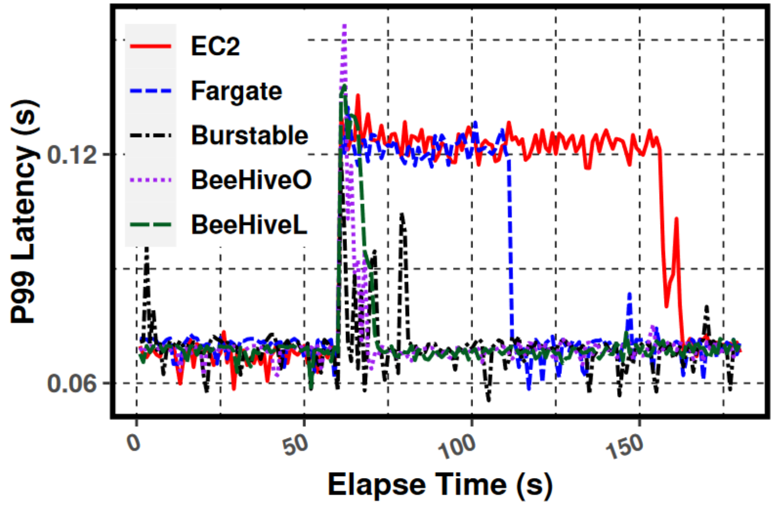
System Design – The BeeHive Runtime System

- **Root method selection**
- The initial closure for offloading is constructed from a root method.
- Method annotations can be used to distinguish user-provided methods from framework ones to avoid unsatisfying performance.

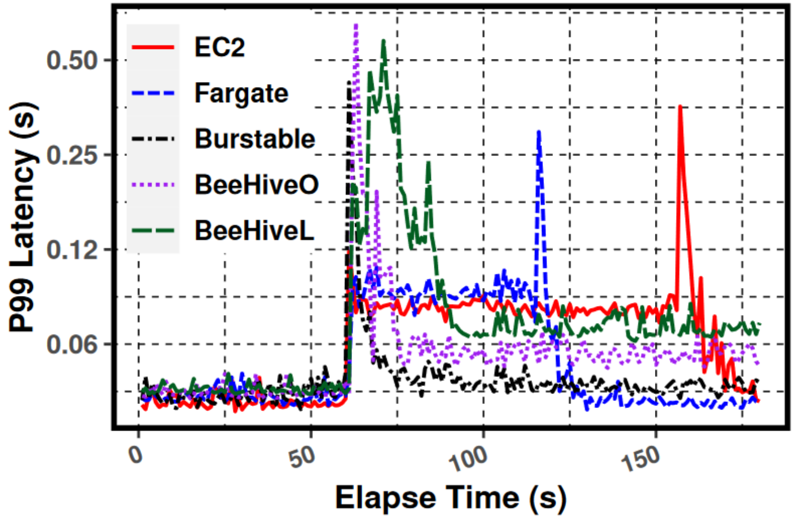
System Design – The BeeHive Runtime System

- **Memory management**
- **Failure Recovery**

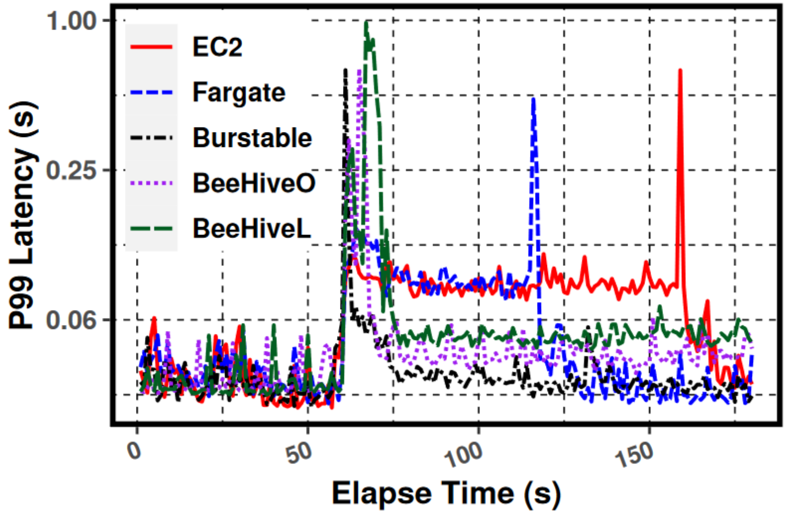
Evaluation



(a) thumbnail

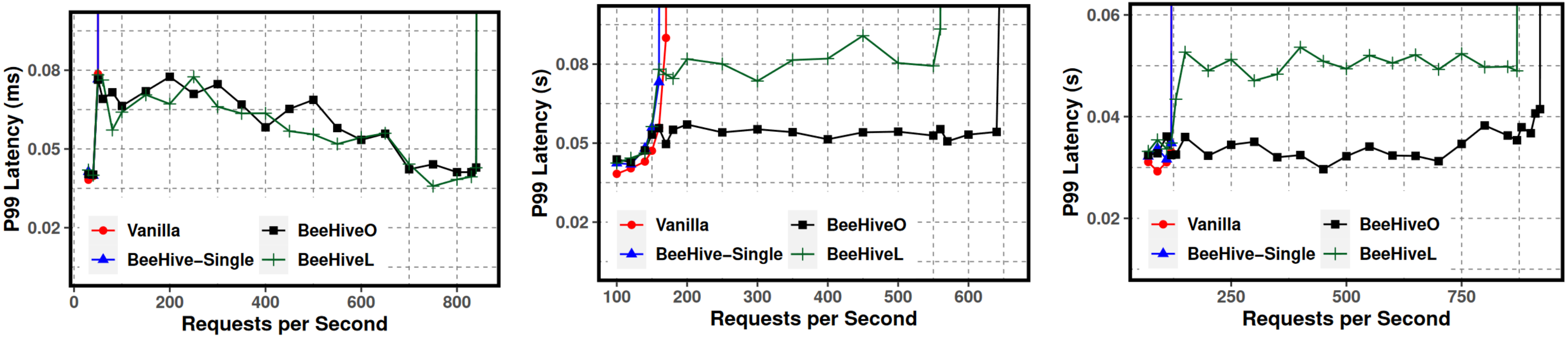


(b) pybbs



(c) blog

Evaluation



(a) thumbnail

(b) pybbs

(c) blog

Evaluation

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

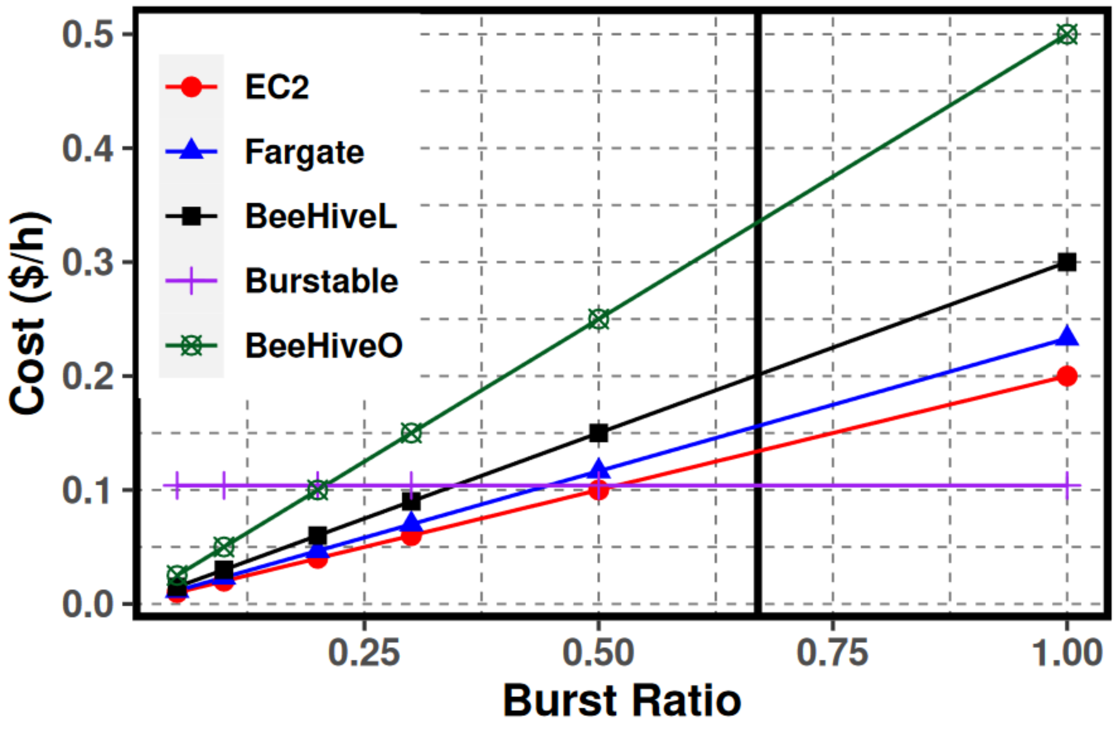


Figure 9: Cost with various burst ratios