Lego-Serverless Platform
Serverless on Disaggregated Resources: Event handling and function creation

Hailin Qin
haqin@eng.ucsd.edu

Tianrui Chen
tic016@eng.ucsd.edu

Tianyu Sun
t9sun@eng.ucsd.edu

Chunnien Chan
chc030@eng.ucsd.edu

1 Introduction

Event handling and function creation module is a key component in modern serverless platforms like AWS Lmabda [1] [2] and OpenWhisk [3]. In this project, we implement the Lego-Serverless Platform, an event handling and function creation platform for modern serverless service. Currently, Lego-Serverless runs on Linux-based clusters. However, it can be easily modified to work for emerging disaggregated clusters like LegoOS-based clusters.

Lego-Serverless provides RESTful API for function and event CRUD. Additional management functions like user authentication and function authorization are supported too. We also design an event scheduling algorithm that provides load balancing and event warm starting. Lego-Serverless is designed for scalability and reliability. A highly available distributed database, CouchDB [4], is employed to provide fault-tolerance. Through our experiment, Lego-Serverless achieves a 2000 queries-per-second throughput under a single-node baseline model.

2 Background

There are many solutions to serverless services. Some of them are developed by cloud providers, while some others are maintained by open source communities.

AWS Lambda [1] [2] is a service provided by Amazon Web Service. It provides a number of built-in language runtimes so that users can directly run a function on its service. It supports an HTTP/REST runtime API, making it easy to upload or invoke functions in any programming language. Functions run in individual sandboxes and are configured with some limits, including memory or running time.

Fig.1 shows the overall architecture of AWS Lambda. Requests are invoked via REST API, and the frontend component will authenticate these requests, check for authorization, and load the metadata of these requests. The execution of code will happen on the Lambda worker fleet, and events of a single function will only be executed within a few workers, which

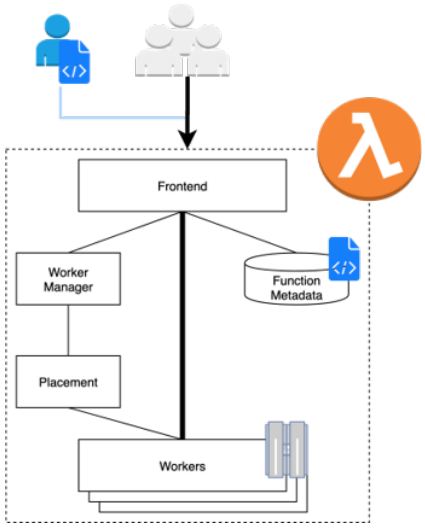

Figure 1: AWS Lambda architecture

can help improve cache locality, enable connection re-use and eliminate the cost of loading and moving the code. This feature is realized by the worker manager, a custom stateful router with high-volume and low-latency. Lambda workers offer several slots, and each slot is only ever used for one function. When a slot is available, the worker manager will inform the frontend. If no slots are available, it will call the placement service to request a new slot. The placement service tries to optimize the placement of slots in order to minimize the resource utilization of each individual worker. It is also responsible for limiting the lifetime of slots, terminate idle slots, or update the software.

Openwhisk [3] is another serverless platform by Apache. It was initially developed by IBM, and donated to open source communities in 2016. IBM Cloud uses this platform to provide their Function-as-a-Service (FaaS) service.

OpenLambda [5] is an research based open-source serverless platform. It is implemented in Golang and based on Linux containers.

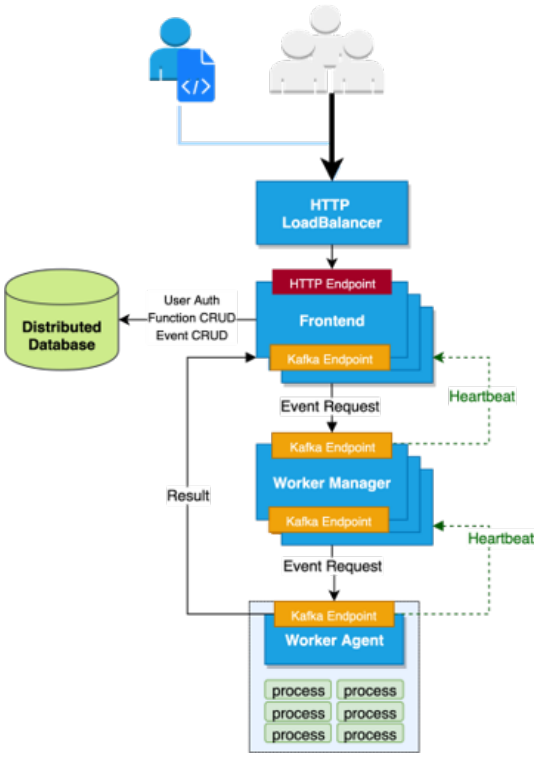

Figure 2: Overall architecture

3 Lego-Serverless Architecture

The overall architecture of Lego-Serverless Platform is shown in Fig.2. It consists of three components, **Frontend**, **Worker Managers** and **Worker Agents**. The components communicate to each other through Kafka [6], an unified, high-throughput and low-latency streaming platform. All the components can be dynamically scale-out, which means our platform is highly flexible, scalable and fault-tolerant. The sequence diagram of our platform is shown in Fig.3.

3.1 Frontend

The frontend component provides users an interface to access our serverless platform. It is written in Python with Flask, with CouchDB [4] as the distributed database. The architecture is shown in Fig.4. It provides a complete RESTful APIs for users to manipulate function metadata and invoke functions. The RESTful APIs are also documented in OpenAPI specification and published on Swagger [7]. The frontend component is important for user experience and overall performance since it is responsible for handling all the HTTP requests and database manipulations. It is also essential for the separation of concerns. The frontend component does request authorization and authentication and resolves the complicated user configurations and annotations of a function. With the help of frontend, worker managers and worker agents can focus on executing events without worrying about where the event comes from, how to get the configurations, and how to present the result to the user. What follows is the terms used in our platform:

- **Function**: functions are what users create and run on the Lego-Serverless Platform. Every function is associated with a set of configurations, including resource limits (CPU, memory, and timeout) and annotations. When a user creates a function, he or she must provide a unique function name along with an executable binary. Users can also make the function public for anonymous invocation or set a whitelist to allow individual user to invoke it. The function will be invoked to create events later.

- **Event**: every time when a function is invoked, a new event is created. The event instance contains information about when the function is invoked and ended and the result (*stdout* and *stderr*) of the execution. An event can come along with a list of parameters and configurations (like timeout for a single event only) that overwrite the configurations of the target function. The event information will then be prepared and sent to the worker manager for execution.

To communicate with worker managers and worker agents, frontend is implemented with a set of Kafka endpoints. Frontend handles periodic heartbeat signals from worker managers via Kafka. The heartbeat signal is to dynamically add and remove managers. When an event is ready to be dispatched to the worker manager for execution, the frontend will uniformly randomly choose a worker manager from the available manager set to handle the event. This mechanism can be treated as a basic load balancer for the worker managers. Since we assume that every manager monitor all the workers and the duplication of worker manager is only for redundancy, uniform dispatch does not affect the performance of the load balancer in the worker manager.

3.2 Worker Manager

The goal of worker managers is to provide load-balanced event dispatcher and worker agents monitor. Fig.5 shows the architecture of the worker manager. The worker managers communicate with both the frontend components and the worker agents through Kafka. Upon receiving the event invocation request from the frontend component, the worker manager chooses a worker for the event execution and sends the request to the corresponding agent on the assigned worker. Besides, the worker manager sends heartbeat messages to the frontend to inform their liveliness. It also accepts the heartbeat from worker agents to keep track of the liveliness of workers.

The event assignment is performed according to a hashing-based load-balancing algorithm. Listing 1 is the pseudo-code or our load-balancing algorithm. At first, the worker manager

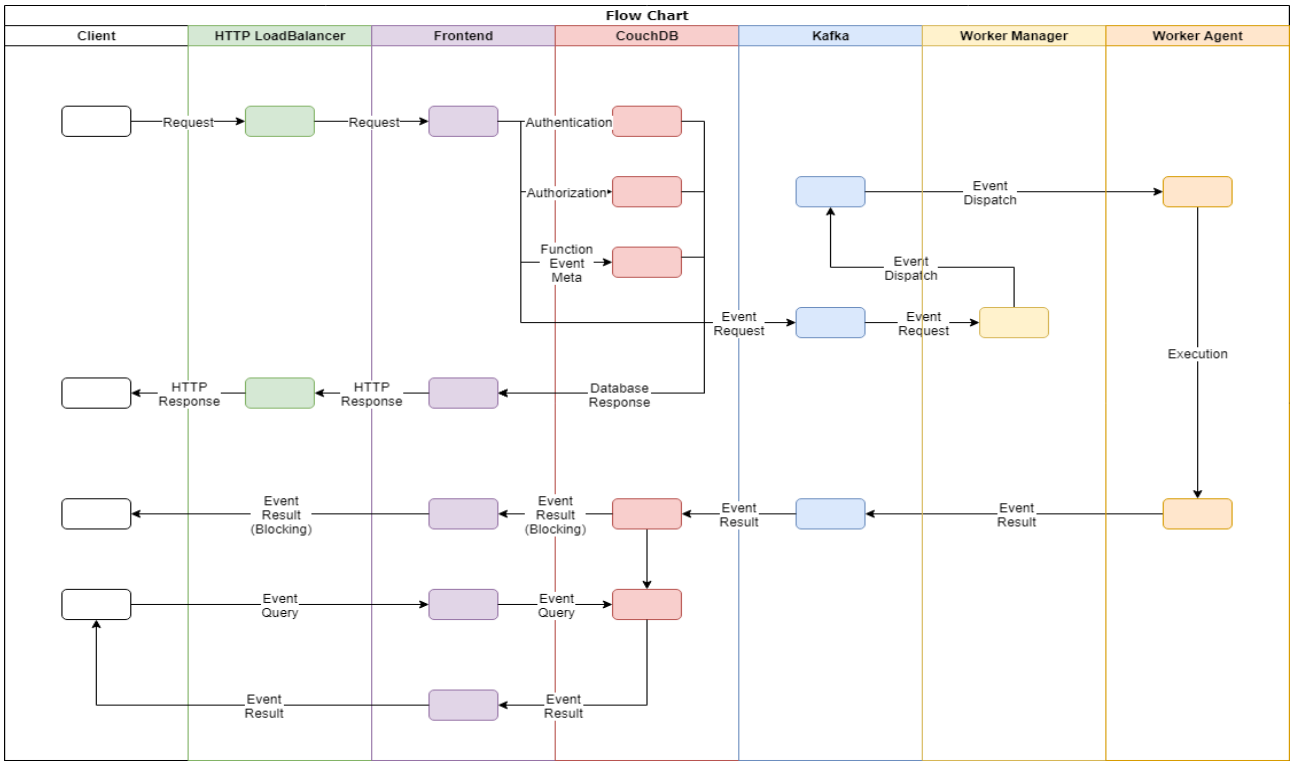

Figure 3: Sequence diagram

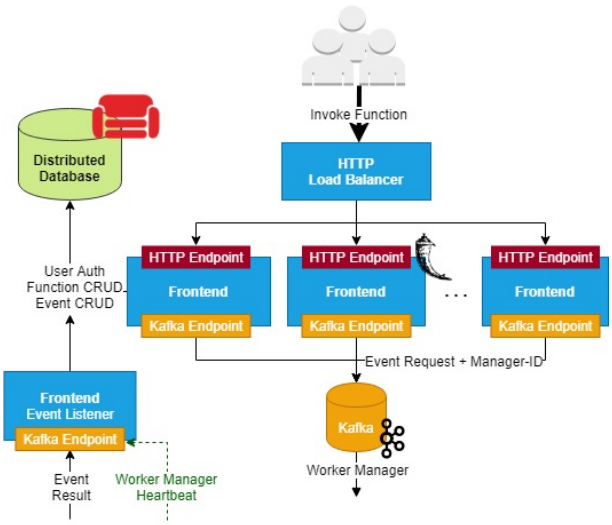

Figure 4: Frontend architecture

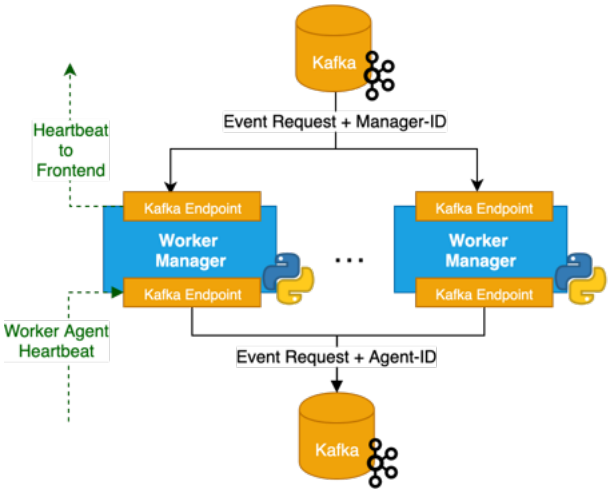

Figure 5: Worker Manager architecture

hashes the *functionID* of the event and use the generated hash code to determine the primary worker and step size for the function. To scatter events among workers, we use a *stepSizeList* to assign different step size for different functions. The *stepSizeList* is a list of all prime numbers that is less than *numWorker* and co-prime with *numWorker*. Then, starting from the primary worker, it would check if the current worker fulfills the capacity requirement, mainly memory requirement, for the event. If not, it moves on to the next worker with the step size.

By hashing the *functionID*, different events corresponding to the same function tend to be dispatched to the same sets of workers. Therefore, we can provide warm starting for the events to improve startup time. Besides, our algorithm scatter events of different functions among the workers by using different step sizes. Worker capacity is also taken into account to avoid overloading workers.

```
def assignWorker(funcId, memLimit, memCapacity,
    numWorker, stepSizeList):
    # funcId: function ID for the event
    # memLimit: memory limit for the event
    # memCapacity: list of available memory on
    workers
    # numWorker: total number of workers
    # stepSizeList: list of available step sizes
    hashCode = hash(funcId)
    primeIdx = hashCode % numWorker
    stepSize = stepSizeList[hashCode %
    stepSizeList.size()]

    idx = primeIdx
    availMem = memCapacity[idx]
    while(availMem < memLimit):
        idx = (idx + step) % numWorker
        availMem = memCapacity[idx]
    return idx
```

Listing 1: Load-balancing Algorithm

Worker manager collects worker capacity information through heartbeats. Worker agents send the worker capacity information along with heartbeat messages. Managers can monitor the liveliness of all the workers and update the capacity information at the same time.

3.3 Worker Agent

The main responsibility of worker agents is to execute the events on workers, monitor their execution, and return their results. The architecture of this component is shown in Fig.6.

When a worker agent is running, it will continuously send a heartbeat to managers through Kafka. Manager nodes should use the agent ID field in heartbeats to monitor whether there is a new agent, or whether an agent has crashed. Heartbeats also contain the capacity information of the current node, including CPU usage, memory usage, etc. worker managers can use this information to do a better load-balance.

Worker agents receive event requests from worker managers, including the binary image of the function (encoded

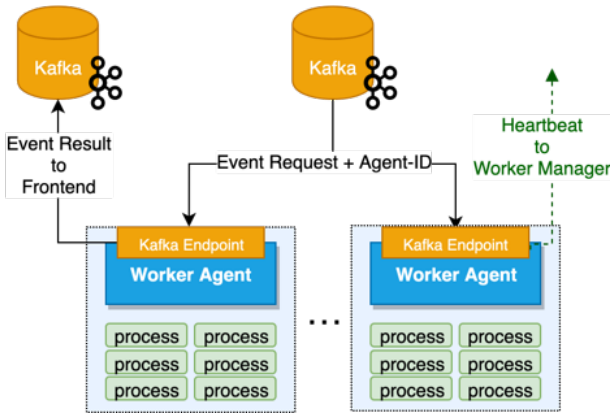

Figure 6: Worker Agent architecture

in Base64), parameters, standard input, time limit, and so on. The function would be executed according to these instructions, and after it finishes, agents would collect results of this execution, including return value, standard output, standard error output, and execution time. This information would then be sent back to the frontend, and some of them would be provided to the users.

Currently, this component is implemented to be hosted on a normal Linux platform. Although the frontend component and worker manager are independent of the worker environment, agents may have to be implemented to support different operating systems and machines.

4 Lego-CLI

Lego-CLI is the command line interface for Lego-Serverless Platform. Similar to other command line interfaces for cloud platforms, like openwhisk-cli [8] and firebase CLI [9], Lego-CLI is a wrapper of HTTP client and provide users a convenient and efficient approach to access our platform. Currently, it supports all the features in our platform. Users can create and update function meta data, invoke function with parameters, and view event result with Lego-CLI.

5 Evaluation

We developed an evaluation baseline model, where we deploy a single node for each layer. An automated function invoker is developed based on Lego-CLI. For the function invoker, we create multiple threads to maximize the requests we send per second. Theoritically, the frontend component is optimized by the WSGI server, which should not be the bottleneck. So the main components we need to evaluate for the performance of our platform is our worker manager, worker agent, and message queue.

For our message queue, Kafka is designed for large-scale messaging, which is generally expected to achieve 5,000 to

50,000 queries-per-second. For the worker agent, since every event would need time to be computed, it is hard to find a metric that makes sense for the evaluation. We cannot measure what the worker agent returns directly as the performance. So all we need to evaluate is the message queue and worker manager. Actually, creating a real-time query rate at more than 5,000 per second level is hard for us. Our solution is to accumulate requests when the worker manager is down and start the worker manager afterward to generate a huge peak. We observed a peak in the worker manager sender at 9,149 queries in 5 seconds, which is approximately 2,000 queries-per-second. The peak rate is way lower than what we could expect from Kafka, which implies our worker manager may be the bottleneck.

6 Discussion

Currently, due to the limitation of time and testing environment, we haven't done much detailed or precise experiments. In the future, we decide to test the throughput of Lego-Serverless with multiple nodes in each component. Also, generating testing requests through REST API is very inefficient, and it can't put enough pressure on our platform. So we may try to modify the code and directly generate testing requests in the frontend component.

In addition to the experiments, more features can be introduced to our platform and supported by the frontend component to provide better user experience. For example, Apache OpenWhisk introduces the idea of "trigger" and "rule," which provides users a better way to invoke the function and manage function annotations and events. Moreover, the support of multiple language runtimes is a future direction to complete our platform. Currently, we only support binary executable and use stdout and stderr as the event result, which is general but not efficient and convenient for developers. For example, like other serverless platforms, the support of Python runtime may enable users to deploy their Python functions and use the function return as the result. This feature will not only make our platform more complete but also allow future improvement on runtime level optimization or even programming language level optimization.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 419–434.

[2] Amazon Web Service, "Aws lambda – serverless compute - amazon web services," *URL https://aws.amazon.com/lambda/*.

[3] Apache, "Apache openwhisk is a serverless, open source cloud platform," *URL https://openwhisk.apache.org/*.

[4] Apache CouchDB, "Apache couchdb," *URL https://couchdb. apache. org*.

[5] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau, "Serverless computation with openlambda," in *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[6] Apache Kafka, "Apache kafka," *URL https://kafka.apache.org/*.

[7] "Lego-serverless api," *URL https://app.swaggerhub.com/apis-docs/jevancc/lego-serverless$_api$/1.0.0/*.

[8] Apache, "apache/openwhisk-cli," *URL https://github.com/apache/openwhisk-cli*.

[9] Google, "Firebase cli reference," *URL https://firebase.google.com/docs/cli*.