Distributed FaaSRunner: Enabling Reproducible Multi-node, Multi-threaded Function-as-a-Service Endpoint Testing

Tomoki Kondo, Austin Bomhold, Robert Cordingly, Dongfang Zhao, Wes Lloyd
School of Engineering and Technology
University of Washington
Tacoma, Washington USA
tkondo2, abomhold, rcording, dzhao, wlloyd@uw.edu

***Abstract*—Today's cloud native applications are often built using a service-oriented architecture supported by many microservices hosted using serverless Function-as-a-Service (FaaS) platforms. The vast majority of serverless function benchmarks and tests are performed using a single-client machine to generate workloads against highly scalable serverless backends. However, single-client test engines can lack sufficient computational resources and network bandwidth to adequately stress serverless backends. Testing tools such as Apache JMeter support orchestrating tests using multiple client nodes, but lack the ability to orchestrate sophisticated tests with distributed workload patterns common with real-world serverless workloads. In this paper, we introduce Distributed FaaSRunner, a distributed test tool which supports the ability to reproduce multi-node serverless workloads using traces derived by ingesting serverless function log files and randomly generated workload traces. We test Distributed FaaSRunner's ability to precisely reproduce serverless function request dispatch and arrival time using various test cluster configurations. Using globally distributed clients, we predict latency to adjust workload trace event dispatch times to reproduce original request arrival latency. We demonstrate that Distributed FaaSRunner can reproduce both temporal and spatial characteristics of serverless workloads, enabling new capabilities to assess performance of FaaS platforms beyond traditional load testing.**

***Index Terms*—Function-as-a-Service; Serverless Computing; Load Testing; Multi-Node Testing; Reproducible Test;**

I. Introduction

Scalability testing of microservices is imperative to ensure that service-oriented applications can handle user demand within the required response times [1]. To ensure that service deployments meet scalability requirements and to compare the utility of different Function-as-a-Service (FaaS) host platforms, it is important that scalability tests are easily reproducible so new service deployments and configurations can be continuously validated. Generating load tests using a single client, particularly for short-lived microservices, is often insufficient. Test clients are often the bottleneck when having to generate thousands of concurrent requests in parallel against endpoints with millisecond response times. Test clients may lack adequate computational resources and network bandwidth to meet throughput goals.

Support for reproducible tests enables fair and repeatable comparisons of serverless application deployments made to different cloud providers and/or platforms with different configuration settings. By reproducing distributed multi-client workloads, we can recreate identical load and distribution characteristics, enabling the ability to infer which platforms and configurations work best for particular scenarios. Precisely reproducing the temporal and spatial characteristics of distributed workloads to compare alternate serverless application deployments and platforms is currently challenging and largely unsupported. While a few multi-node test tools exist, existing tools do not support orchestrating complex tests to precisely reproduce these characteristics of distributed workloads seen in real-world environments.

A fundamental challenge in reproducing distributed workloads lies in addressing the reproduction of arrival times recorded in event traces, while taking into account network latency. Since network latency varies across cloud regions and fluctuates over time due to congestion or route changes, when reproducing workload load traces without correction, arrival times may not faithfully reflect the original ordering of events. As a result, naively replaying such traces can violate the original event sequence and undermine temporal fidelity. To mitigate this discrepancy, reproducing traces based on dispatch times adjusted using network latency estimates can achieve higher fidelity in timing reproduction. Our approach aims to ensure that trace events match the original relative arrival times. These inter-event timings can be crucial for testing FaaS-specific behaviors such as cold start and reuse of runtime environments. For distributed testing, latency adjustments are key for reproducing temporal workload characteristics to enable accurate comparisons across FaaS platforms.

In this paper, we introduce Distributed FaaSRunner, a distributed load testing tool that precisely reproduces multi-user workloads against serverless as well as general REST endpoints [2]. To support distributed testing, we also provide a workload trace generator, the Serverless Event Trace Generator (SETGen), which can generate random or log derived workload traces. Distributed FaaSRunner orchestrates client nodes distributed across cloud regions, localized to a single region, or to a single availability zone (AZ). In addition, Distributed FaaSRunner supports adjusting scheduled event timings based on preobserved network latency. Users can specify each node's latency offset using the `timeAdjustmentMs` attribute in the test cluster's configuration file. This offset adjusts the scheduled event times at each client node to account for node-specific network latency. By accounting for network latency of distributed clients, our design facilitates more faithful reproduction of distributed event traces. We investigate our

ability to reproduce temporal characteristics of distributed workloads using diverse test clusters and evaluate the impact of latency-based schedule adjustments on the accuracy of request arrival timing. Supporting distributed load testing with reproducible workload traces offers a novel way to characterize the performance of FaaS platforms not currently available.

A. Research Questions

To evaluate our tool's ability to reproduce distributed workload traces, we investigate the following research questions:

(RQ-1 Workload Event Latency Characterization): What is our ability to reproduce distributed workload traces using distributed test clusters consisting of nodes disbursed across multiple cloud regions at continental vs. global levels? Without event scheduling adjustments, what request arrival latency is observed at the serverless endpoints?

(RQ-2 Adjustment Capability): By adjusting arrival times of scheduled events based on expected network latency, to what extent can we reduce request arrival latency when reproducing distributed event traces?

(RQ-3 Latency Prediction): What is the efficacy of alternate methods to forecast network latency to adjust scheduled event timings to reproduce distributed event traces at the continental and global level?

II. Background and Related Work

Scalability of web service and FaaS endpoints is often evaluated in terms of supported throughput (e.g. requests/second) and whether services return valid responses under increasing load. We review the capabilities and limitations of existing load testing and event replay tools.

A. Load Testing Tools and Workload Replay tools

Apache JMeter is a widely recognized multi-threaded load testing tool [3] , supporting distributed load testing by orchestrating multiple clients to support parallel load generation of service endpoints [4]. For multi-client load testing, however, users can only specify a throughput rate. JMeter does not support execution of workload traces to reproduce specific request arrival patterns or load distributions. **BlazeMeter**, a commercial SaaS platform built on top of JMeter, allows users to execute JMeter test plans in the cloud and scale load generation by specifying the number of concurrent virtual users [5]. This provides cloud-based scalability and ease of deployment, but inherits core limitations from JMeter. As a result, both JMeter and BlazeMeter are essentially limited to stress testing, as specific workload traces cannot be reproduced.

The CLI-based load testing tool, **k6**, emphasizes programmable tests using a lightweight high performance execution environment with tests defined using JavaScript [6]. Users can issue requests based on event traces or generate loads with specified throughput levels to multiple endpoints. K6 supports distributed tests via the k6-operator with Kubernetes [7]. This mechanism allows users to expand load testing to multiple Kubernetes Pods to execute tests in parallel. By using the commercial cloud service Grafana Cloud, users can perform distributed load tests with k6, and test results from each client can be aggregated into a unified visual report [8]. For distributed tests, however, the k6-operator supports only a single Kubernetes cluster, limiting clients to a single region. Grafana Cloud allows assigning a proportion of the total virtual users to predefined regions or zones, but individual events cannot be mapped to specific nodes as with Distributed FaaSRunner. Therefore, it is difficult to accurately reproduce spatial characteristics of distributed workloads.

Locust is a Python-based load testing tool that supports distributed execution via a master-worker architecture. Its programmability and native support for worker coordination over TCP enables flexible test scenarios. Although Locust does not provide built-in mechanisms for time synchronization or replay of workload traces across nodes, users can implement such functionality, including accurate reproduction of spatial and temporal characteristics of workload traces. However, doing so requires substantial coding effort.

FaaSRunner is a Python-based single-node test client designed to orchestrate tests of serverless FaaS functions [9]. FaaSRunner performs experiments and measures round-trip time, latency, and a number of other performance metrics. FaaSRunner works in combination with the Serverless Application Analytics Framework (SAAF) to characterize serverless function execution providing more than 48 distinct metrics (e.g. runtime, VM CPU type, and function state-*cold/warm*) [10]. FaaSRunner's implementation using Python threads has limited throughput due to Python's Global Interpreter Lock (GIL). FaaSRunner does not support distributed multi-node testing and is limited to using only a single node to stress endpoints.

Shahrad et al. [11] provided production serverless workload traces of Azure Functions [12], including per-function invocation counts, trigger types, execution times, and per-application memory usage. To reproduce these traces, they developed **FaaSProfiler**. They evaluated the function state (e.g., cold or warm) by measuring request frequency and referring to documentation published by cloud providers, without directly inspecting the internal state of the FaaS infrastructure. They analyzed the internal infrastructure state using logs from a locally hosted OpenWhisk [13] setup in a separate study [14], but this approach is not applicable to commercial FaaS platforms. Furthermore, FaaSProfiler does not support multi-node execution and lacks mechanisms to coordinate request dispatch across multiple nodes. As a result, FaaSProfiler is limited in its ability to reproduce workload traces that require higher throughput than what can be achieved by a single node.

B. Serverless Benchmarking Frameworks

Serverless benchmarking frameworks provide reusable libraries of serverless test cases, including individual functions and entire applications. **Serverless Benchmark Suite (SeBS)** provides serverless function and application use cases, including web applications, multimedia use cases, utilities, an inferencing example, and graph computational use cases [15]. Benchmarking frameworks, such as **ServiBench**, use proba-

bilistic state machines to generate dynamic user stories and trace complex serverless workflows [16]. **ServerlessBench**, is a framework designed to assess metrics in serverless computing, including communication latency and startup latency [17]. Serverless benchmarking frameworks provide reusable use cases for developers and practitioners to compare performance of serverless platforms and configurations.

While serverless benchmarking frameworks provide reusable serverless function and application use cases, these frameworks do not facilitate ***distributed testing***. They are not designed to reproduce temporal or spatial characteristics of distributed real-world serverless workloads or to characterize how platform performance varies over time, across different regions, and cloud providers.

III. Methodology

A. Architecture of Distributed FaaSRunner

The Distributed FaaSRunner developed in this study is a Java-based distributed load testing tool capable of generating high-throughput workloads targeting general HTTP endpoints. It achieves precise scheduling of request transmissions by leveraging multi-threaded parallelism and fine-grained timing control, enabling accurate reproduction of timing-sensitive scenarios. While the tool itself is not limited to FaaS platforms, it supports detailed performance analysis of cold starts, runtime variability, and resource utilization when the target endpoint is implemented as a FaaS function enabled by the Serverless Application Analytics Framework (SAAF) described in III-C. Additionally, Distributed FaaSRunner supports synchronous and asynchronous HTTP request execution, enabling flexible interaction with a broad range of deployment environments.

Node Structure and Scalability: Distributed FaaSRunner is designed to operate in a multi-node environment, enabling horizontal scalability to handle increasing loads. The architecture consists of two types of nodes: **controller nodes** and **worker nodes**. Worker nodes communicate via TCP and are deployed to distinct virtual machines (VMs) for scale-out. Our current implementation supports a single controller node per cluster. Supporting redundant controllers is left for future enhancement. All nodes leverage the Network Time Protocol (NTP) for clock synchronization. This architecture enables sustained load levels infeasible with a single client node, by orchestrating multiple nodes in a coordinated manner.

Execution Modes: The tool supports two operational modes, tailored for different testing scenarios: **Throughput Mode**: In this mode, the user specifies a target throughput and a request dispatch pattern (e.g. Poisson, random, or uniform distribution). Requests are generated and sent to a single endpoint. The generation logic for each pattern follows the same implementation as SETGen, described in Section III-B. Support for multiple request dispatch patterns enables users to produce different traffic profiles based on their testing objectives. **Event Trace Mode**: In Event Trace Mode, the system loads a predefined event trace from a JSON file and dispatches requests to multiple endpoints with precise timing. Event trace files are randomly generated or derived from parsing serverless function log files to enable reproduction of complex traffic patterns with specific request timings. Each event in the trace is characterized by the following five attributes: **`eventID`**: unique identifier for the event, **`time`**: scheduled request time, **`nodeID`**: identifier of the responsible worker node, **`endpoint`**: destination endpoint URL, and **`payload`**: data to be included in the request.

The Event Trace Mode operates as follows: **Event Loading**: a local client sends the full event trace file to the controller node, where each event specifies the responsible `nodeID` and scheduled event time. **Per-Node Trace Generation**: The controller node parses the original event trace to generate worker node specific event trace files. **Event Distribution:** The controller distributes event trace files to the corresponding worker nodes based on a configuration file attribute that maps `nodeID` to a public IP. These preprocessing stages alleviate potential bottlenecks in the distributed system. **Experiment Execution:** After all workers are ready, the controller sends a start signal. Each worker waits until the scheduled start time before generating any scheduled requests.

B. Serverless Event Trace Generator (SETGen)

Distributed FaaSRunner enables reproduction of event traces to replicate temporal and spatial characteristics enabling a new degree of test reproducibility beyond that of current load testing tools. Distributed FaaSRunner's temporal precision enables the reproduction of traces of highly concentrated event sequences consisting of hundreds of calls on a millisecond scale. SETGen supports creating event traces with reproducible event patterns. SETGen provides two modes of trace generation: log file-based and random. In the log file-based mode, SETGen parses a serverless endpoint's log files from AWS CloudWatch and composes them into a workload trace file for use by the Distributed FaaS Runner. If client IP addresses are present in the logs, SETGen uses them to assign events from the same IP to the same node in the resulting trace file. In random generation mode, users provide a statistical distribution pattern, request time intervals (in milliseconds), and a total number of events or a trace duration. Individual event times follow a specified statistical distribution: **Uniform**: fixed dispatch intervals. **Poisson**: request dispatch intervals follow an exponential distribution, $f(x;\lambda) = \lambda e^{-\lambda x}$, with $\lambda = 1/\mu$, where μ is the user-defined request time interval. **Random**: request dispatch intervals are selected from the continuous range $U(0, 2\times\frac{1000}{x})$[ms], with $x = 1/\mu$, where μ is the request time interval. **Sinusoidal**: request intervals oscillate sinusoidally according to the function $\mu + A\sin\left(\frac{2\pi t}{T}\right)$, where μ is the request time interval, A is the amplitude, and T is the period.

C. Serverless Application Analytics Framework (SAAF)

SAAF is a framework to profile the performance, resource utilization, and infrastructure of serverless FaaS functions [18]. Using SAAF, attributes including CPU type, running time, start time, etc. can be measured. SAAF also supports profiling serverless functions written in multiple languages

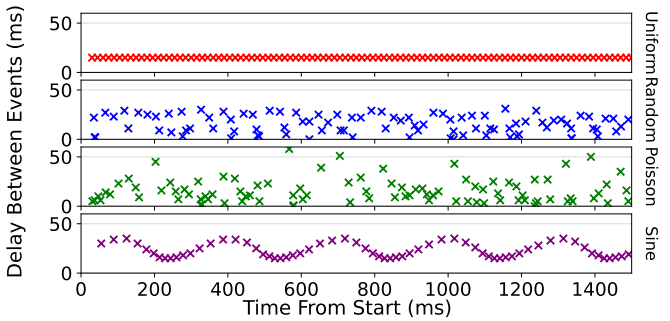


Fig. 1: SET Generator Event Distributions

including Python, Java, Go, and Node.js. Additionally, SAAF is supported on many cloud platforms including AWS Lambda, Google Cloud Functions, Digital Ocean Functions, IBM Cloud Code Engine, Azure Functions, and OpenFaaS. We use SAAF to assess whether a FaaS endpoint is cold or warm, and observe distinct function invocation times.

D. Experimental Design

We leverage two metrics to assess the precision of reproducing workload traces: **Client Dispatch Delay**: difference between worker node's actual request dispatch time and the event's scheduled time; and **Request Arrival Latency**: the difference between the client's request arrival time at the server and the scheduled event time. For the FaaS endpoint, we deployed SAAF's Python 'hello' template function to capture profiling metrics for a no-op function on AWS Lambda. The template function executes SAAF profiling in less than 10 ms, returning the function's execution start time in SAAF's inspection data. We treat the time as the request arrival time. We deploy separate AWS EC2 c7i.xlarge instances with 4 vCPUs and 8 GB RAM for the controller and worker nodes unless otherwise noted. Each experiment repeated a one minute workload trace eleven times to account for cold starts and temporal variability. Data from the first run are treated as a warm-up and are excluded from analysis. We used the asynchronous request mode of Distributed FaaSRunner to evaluate the tool's precision under high-throughput and independent dispatch conditions. Between workload trace runs, we waited 30 seconds to ensure AWS Lambda function instances remained warm [19]. We tested 3 types of distributions; **uniform**, **Poisson**, and **random**. These workload traces were generated by SETGen. We used the Amazon Time Sync Service (ATSS), an implementation of NTP to synchronize time on all nodes [20].

Ex. 1 (Single Node Performance Comparison): This experiment evaluates the precision and reproducibility of workload traces using three load generation tools: k6, FaaSProfiler, and Distributed FaaSRunner. The evaluation focused on how each tool replicates event timings with varying request patterns and throughput levels on a single client node. We deployed AWS EC2 instances and the AWS Lambda function in the same AZ (us-east-2c). We tested workload traces generated by SETGen with uniform, Poisson, and random distributions at 100, 200, 500, and 1000 requests per second (rps). All workload traces were converted into proper formats for each tool prior to the experiments using custom conversion scripts. Each tool preloaded traces into memory before execution, ensuring that no file I/O occurred during the request dispatch phase, avoiding any I/O impact on performance.

Timing Measurement: **k6**: We measure the request dispatch time by summing the start time with the connection latency, TLS handshake, and the time to write requests to the socket. We utilized 200 virtual users (VUs) for the experiments. **FaaSProfiler**: FaaSProfiler utilizes the requests-futures library's FuturesSession for asynchronous HTTP requests. Timestamps are recorded immediately after invoking the `post()` method, aligning with the tool's inherent design. We assume that the request has been dispatched when the asynchronous method call completes, and use the corresponding timestamp to record the request dispatch time. **Distributed FaaSRunner**: We record the timestamp immediately after submitting the request to the asynchronous HTTP client. We treat the timestamp as the request dispatch time.

As all dispatch timings are recorded at the application layer on the client side, they do not guarantee the exact time at which the request packet was sent on the network. To validate the actual request execution during experiments, we additionally measured the observed throughput at the target endpoint. Specifically, we used the START log entries automatically emitted by AWS Lambda in the CloudWatch Logs, which are recorded when the function begins execution, to verify that each request was successfully invoked [21].

Ex. 2 (Horizontal Scalability Evaluation): To assess the scalability of dispatch timing precision for multi-node experiments, we deployed five nodes to us-east-2c AZ as a test cluster. Each node executed the same trace concurrently. We tested uniform, Poisson, and random workload traces that performed sequential function calls with distinct event timings with an expected throughput of 500 rps per node. We used a backend Lambda function with a 'concurrent executions' quota of 3,000. This experiment evaluates horizontal scalability by investigating timing skew and reproducibility under concurrent execution with multiple nodes.

Ex. 3 (Parallel Trace Reproducibility): This experiment evaluates the tool's ability to reproduce traces with parallel requests. We consider three types of request patterns to evaluate different levels of concurrency: 1) **Sequential pattern**, the baseline, with uniform request rates at 100, 200, and 500 rps, with no concurrency. 2) **Small burst pattern**, where the base rate is fixed at 500 rps, and five requests are dispatched in parallel every 10 milliseconds. 3) **Large burst pattern**, which issues 10, 20, and 50 parallel requests at 100-ms intervals, for base rates of 100, 200, and 500 rps, respectively. We tested two configurations to investigate how the horizontal scaling of client nodes affects the accuracy of parallel request reproduction. In the single-node setting, one client instance in us-east-1a is responsible for issuing all requests in parallel. In the multi-node setting, five client instances in us-east-1a are used to distribute the same burst workload, with each node

responsible for a fraction of the requests scheduled at the same timestamp. For multi-node experiments on EC2, we tested EC2 placement groups: cluster (i.e., worker VMs grouped) and spread (i.e., worker VMs isolated) [22].

Ex. 4 (Impact of Test Cluster Configuration and Offset Correction on Request Arrival Time): This experiment investigates: (i) how the geographic distribution of worker nodes impacts the reproducibility of request arrival times at the server **(RQ-1)**, and (ii) the efficacy of applying static time offsets, determined by measurement observations to improve the reproducibility of arrival times **(RQ-2)**. We deployed one EC2 instance (c6i.xlarge, 4 vCPUs, 8 GB RAM) as a worker node in each of following regions: eu-central-1, ap-southeast-1, ap-northeast-1, sa-east-1, and af-south-1. C6i.xlarge instances were used in this experiment because c7i.xlarge instances were unavailable in af-south-1. Each node sent 100 rps for 60 seconds to an AWS Lambda endpoint deployed in us-east-1a, following a uniform distribution. We first ran the workload trace without any correction for 11 consecutive runs, serving two purposes: to quantify the effect of geographic dispersion on reproducibility **(RQ-1)**, and to obtain median arrival latencies for each region to calculate offsets, a process we refer to as the *measurement-based approach* (MBA). After a 10-minute pause, we replayed the same trace 11 times with the offset correction applied. In both uncorrected and corrected sets, the first run was treated as a warm-up and excluded from analysis. Finally, we compared the median request arrival latency of the corrected sets vs. the uncorrected sets to assess to what extent our measurement-based approach to offset correction improves temporal reproducibility **(RQ-2)**.

Ex. 5 (Request Arrival Latency Prediction): We evaluated how accurately the median request arrival latency can be estimated in advance, to enable adjustment of request timings for workload traces **(RQ-3)**. Our goal was to assess accuracy of alternative methods to predict event offsets to minimize request arrival latency when requests originate from different locations (i.e., cloud regions). Our objective is to adjust event timings up front without having to first profile event latencies for the entire workload trace.

We compared three methods to estimate event offsets against our baseline approach of profiling the entire workload up front to measure request arrival latencies and establish event offsets (our MBA). Our first method used AWS Network Manager's Infrastructure Performance feature, which provides round-trip time (RTT) data measured by AWS-managed internal probes [23]. These measurements are performed at 5-minute intervals between regions and are published as P50 metrics, offering reliable and consistent latency information. Our second method relied on CloudPing, a third-party latency monitoring tool that continuously measures TCP handshake RTTs (e.g., P50, P90) between AWS regions [24]. Our third approach constructed a simple linear regression model based on physical distances between regions and previously observed latencies. AWS Region coordinates (latitude and longitude) were sourced from publicly available data on GitHub [25], and although the exact data center locations are not officially disclosed, we assume that errors on the order of several tens of kilometers are inconsequential when inter-regional distances are typically on the order of thousands of kilometers. To construct the distance-based linear regression model, we first executed 11 runs of our workload trace under the same conditions with 10 regions (us-east-1, us-east-2, us-west-1, us-west-2, ca-central-1, eu-central-1, ap-northeast-1, ap-southeast-2, sa-east-1, and af-south-1) and used the results to train the model.

TABLE I: Endpoint throughput mean and standard deviation for each tool using our workload trace w/ uniform distribution

Tool	Target rate	Mean (rps)	Std
k6	100	100.036	0.007
k6	200	200.062	0.028
k6	500	500.080	0.040
k6	1000	1000.055	0.113
FaaSProfiler	100	100.047	0.012
FaaSProfiler	200	186.591	2.389
FaaSProfiler	500	188.807	1.783
FaaSProfiler	1000	188.020	1.336
Distributed FaaSRunner	100	100.031	0.006
Distributed FaaSRunner	200	200.049	0.033
Distributed FaaSRunner	500	500.093	0.030
Distributed FaaSRunner	1000	1000.153	0.151

We then performed a single run of our workload trace with 6,000 function calls to obtain the median request arrival latency which we used to calculate each method's accuracy. The first 30 seconds of the trace were used only to measure the actual median request arrival latency for each region, which also served as offset values for our MBA. For our evaluation, for the ground-truth, we used the median request arrival latency observed during the last half of the trace (30 seconds). To ensure temporal consistency, for our AWS Network Manager and CloudPing approaches, we used the most recent RTT values available immediately prior to this evaluation run. Since AWS Network Manager and CloudPing report RTT, we approximated one-way network latency by dividing the RTT values by two. In Figure 7, we compare six AWS regions from this experiment: us-east-1, eu-central-1, ap-northeast-1, ap-southeast-2, sa-east-1, and af-south-1. Prediction error was calculated by comparing the predicted values from each method with our ground-truth observations.

IV. Results

A. Ex. 1. Single Node Performance Comparison

We first report the observed endpoint throughput to ensure that all tools successfully executed the input traces and issued requests as expected. This step is essential to validate that the client-side dispatch timing measurements correspond to actual request executions. Table I summarizes the mean and standard deviation of endpoint throughput, binned at 1-second intervals, between the arrival of the first and last requests. While both k6 and Distributed FaaSRunner successfully reproduced the target throughput up to 1000 rps, FaaSProfiler plateaued at around 188 rps, failing to sustain higher throughput. This result indicates that FaaSProfiler's throughput was constrained to below 200 rps on a c7i.xlarge instance.

Figure 2 compares the client dispatch delay precision for k6, FaaSProfiler, and Distributed FaaSRunner based on the

client dispatch delay observed under workloads according to a uniform distribution at 100, 200, 500, and 1000 rps. Since FaaSProfiler failed to sustain 200 rps and above, only the 100 rps case is included. Both Distributed FaaSRunner and FaaSProfiler demonstrate high reproducibility, with nearly all requests dispatched within 0.5 milliseconds of their scheduled time. While k6 achieves high precision for approximately half of the requests (with delays under 0.1 ms), the remaining requests exhibit delays around 1 ms, revealing a bimodal dispatch pattern that limits its accuracy for reproducing fine-grained workload traces. The results of the client dispatch delay experiment are summarized in Table II. The table shows statistical measures of the client dispatch delay at each throughput level for Distributed FaaSRunner. Based on these results, we see that reproducing the workload trace features sub-ms dispatch delay with a mean of 0.18 ms across all distributions at 1000 rps. These results suggest that Distributed FaaSRunner offers the highest timing fidelity among the evaluated tools, making it well suited for experiments that require sub-ms precision in trace-driven workload reproduction.

TABLE II: Client dispatch delay (ms) for increasing asynchronous request rates for Distributed FaaSRunner. CV denotes the coefficient of variation.

Pattern	Rate	Min	Q1	Median	Q3	Max	Mean	SD	CV%
uniform	100	0.119	0.225	0.257	0.302	9.392	0.271	0.139	51.384
uniform	200	0.050	0.190	0.212	0.233	7.968	0.222	0.153	69.005
uniform	500	0.032	0.143	0.181	0.196	16.124	0.183	0.291	158.520
uniform	1000	0.027	0.086	0.093	0.098	39.707	0.123	0.508	414.312
Poisson	100	0.079	0.190	0.230	0.284	6.317	0.246	0.152	61.956
Poisson	200	0.077	0.166	0.205	0.239	12.324	0.213	0.171	80.353
Poisson	500	0.022	0.123	0.188	0.216	14.177	0.188	0.226	120.481
Poisson	1000	0.020	0.097	0.123	0.176	32.779	0.172	0.662	385.132
random	100	0.079	0.203	0.238	0.280	5.647	0.249	0.140	56.281
random	200	0.076	0.178	0.207	0.236	9.997	0.215	0.167	77.894
random	500	0.036	0.144	0.189	0.209	17.288	0.194	0.331	170.437
random	1000	0.022	0.105	0.155	0.198	29.394	0.182	0.564	310.126

B. Ex. 2. Horizontal Scalability Evaluation

Results of Ex. 2 are presented in Table III. Despite the increased total request volume in the multi-node setting (five nodes), the performance in terms of Client Dispatch Delay does not show notable degradation. The accuracy of request scheduling remains comparable to our single-node test configuration, indicating that Distributed FaaSRunner maintains its dispatch precision even with horizontally scaled workloads.

Figure 3 shows the client dispatch delay histograms for two configurations: a single node that runs the uniform trace at 500 rps and five nodes that execute the uniform trace at a combined throughput of 2500 rps (500 rps per node). The similar distributions demonstrate that Distributed FaaSRunner scales horizontally without compromising timing accuracy.

TABLE III: Distributed FaaSRunner Client dispatch delay (ms) (single vs. multi node). CV=coefficient of variation.

Distribution	Rate	Nodes	Min	Q1	Median	Q3	Max	Mean	SD	CV%
Uniform	500	1	0.063	0.181	0.290	0.334	55.243	0.292	0.518	177.126
Uniform	2500	5	0.053	0.164	0.264	0.325	82.574	0.394	1.729	438.818
Poisson	500	1	0.052	0.177	0.306	0.376	55.635	0.336	0.624	185.635
Poisson	2500	5	0.050	0.198	0.340	2.233	91.955	2.393	4.714	196.997
Random	500	1	0.054	0.174	0.296	0.373	28.972	0.320	0.414	129.569
Random	2500	5	0.047	0.187	0.309	0.543	71.341	1.392	3.234	232.301

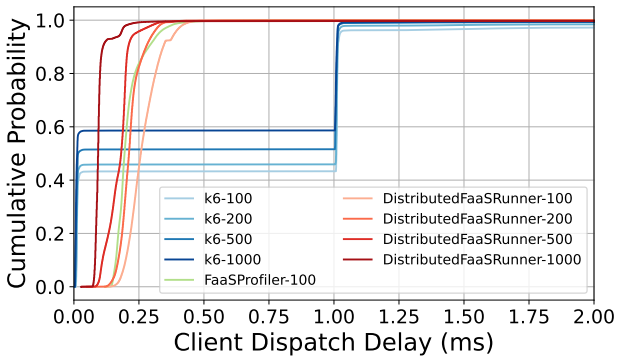


Fig. 2: Client dispatch delay comparison across tools . Uniform distribution with 100, 200, 500, and 1000 rps shown

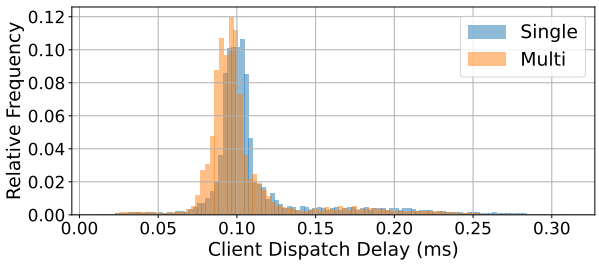


Fig. 3: Client dispatch delay for single and multi node (uniform 500 rps and 2500 rps). Shown 0th–99th percentile.

C. Ex. 3. Parallel Trace Reproducibility

Figure 4 illustrates results of reproducing traces using a single node or multiple nodes deployed using cluster or spread EC2 placement groups. Cluster placement consolidates VMs to a single rack or host to minimize network latency while spread placement distributes VMs across racks in an AZ to reduce risk of correlated hardware failures. For sequential traces (seq_100, seq_200, seq_500), a single-node reproduced requests with very high timing precision. A single node, however, had increasingly high dispatch latency for small and large parallel burst patterns and became a bottleneck.

In contrast, when the burst traces were distributed across five nodes, no notable degradation in timing precision was observed. This suggests that parallel burst patterns can be reproduced with high accuracy when the workload is horizontally distributed. No meaningful difference was observed between EC2 placement group types. Both cluster and spread placement yielded comparable client dispatch delay.

D. Ex. 4. Impact of Test Cluster Configuration and Offset Correction on Request Arrival Time

Results of Ex. 4 are presented in Figure 5 and Table IV. Figure 5 shows the request arrival latency distributions before and after correction, while Table IV reports the median and standard deviation for each region, with the uncorrected and corrected values on separate rows. Before correction, the median request arrival latency tended to increase with geographic

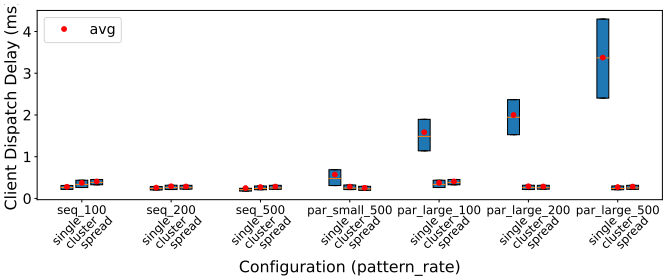


Fig. 4: Client dispatch delay: Sequential (seq) vs Parallel (par). Small and large denote small burst (10 ms interval) and large burst (100 ms interval) respectively

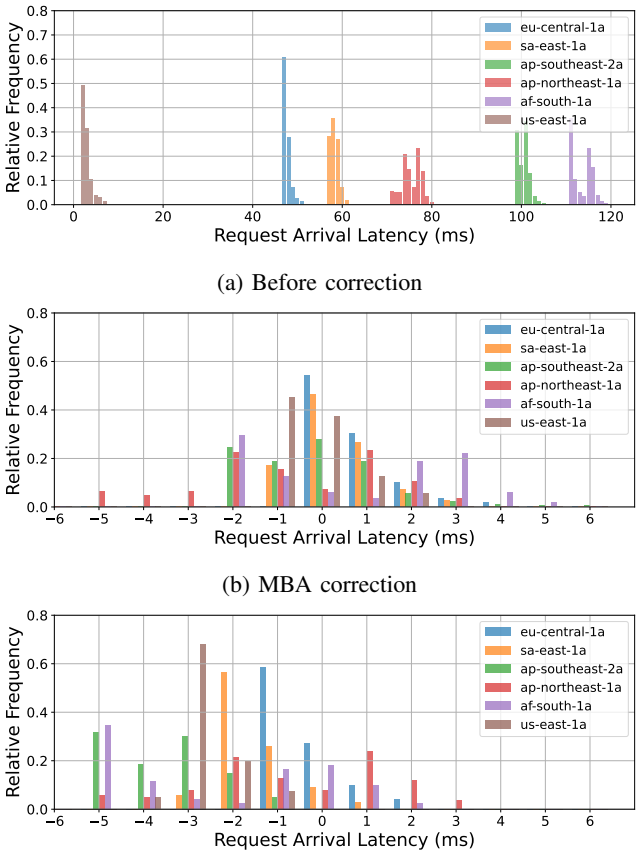
distance, often exceeding 50–100 ms (**RQ-1**). After applying static time offset correction based on the observed per-node median latency, the median arrival time for all regions fell to within a few milliseconds of zero. This correction notably improved the reproducibility of the trace in terms of request arrival timing, achieving an order-of-magnitude enhancement in temporal precision (**RQ-2**). These findings demonstrate that, even in geographically dispersed environments, static offset adjustments can substantially enhance the temporal fidelity of distributed workload trace reproduction, offering a promising approach for multi-node testing with high temporal accuracy and spatial fidelity.

TABLE IV: Effect of static time offset correction on request arrival latency (ms)

Region	Original Median	MBA Median	CP Median	Original SD	MBA SD	CP SD
eu-central-1a	47	0	-1	13.61	13.67	13.51
sa-east-1a	58	0	-2	18.11	18.35	17.94
ap-southeast-2a	101	0	-3	40.67	41.38	40.70
ap-northeast-1a	76	-1	-1	27.01	26.95	26.93
af-south-1a	113	2	-2	48.69	49.11	49.85
us-east-1a	3	0	-3	1.49	1.63	1.04

E. Ex. 5. Request Arrival Latency Prediction

In this experiment, we evaluated the efficacy of predicting inter-region request arrival latency to support adjusting event timings in advance without profiling as required in Ex. 4. As shown in Figure 6, a linear regression model based on geographical distance achieved an R^2 of 0.927 after excluding several exceptional region pairs — (af-south-1, sa-east-1), (af-south-1, ap-southeast-2), (ap-northeast-1, af-south-1), (sa-east-1, ap-southeast-2), and (eu-central-1, ap-northeast-1) — suggesting that distance is a key independent variable for predicting latency. Figure 7 presents the relative error (%) of predicted median request arrival latency for each region pair. Prediction using our linear regression model incurred from 10 to 27% relative error, and was unsuitable for correcting event timings in region pairs with high latency prediction error (**RQ-3**). This limitation stems in large part from temporal variability in inter-region network latency; this level of variation aligns with prior findings reporting daily fluctuations between 2% and 29% [26]. In contrast, RTT-based estimators such as CloudPing and AWS Network Manager consistently



(a) Before correction

(b) MBA correction

(c) CP correction

Fig. 5: Request arrival latency comparison with static time offset correction. MBA denotes measurement-based approach.

yielded relative error below 10% for most region pairs, making them promising alternatives (**RQ-3**). Among all methods, our MBA demonstrated the highest accuracy, maintaining relative errors within just a few percent for all region pairs (**RQ-3**). These results suggest that while the MBA is the most suitable approach when precise correction is required, RTT-based estimators like CloudPing and AWS Network Manager can also serve as sufficiently accurate alternatives to avoid the requirement of profiling workload traces in advance.

V. Conclusion and Future Work

In this paper, we showed that Distributed FaaSRunner can reproduce event traces with better precision than existing distributed testing tools. Distributed FaaSRunner supports distributed traces that reproduce events using geographically specified nodes to reproduce spatial workload traces, a feature missing from other tools. Our results confirm our ability to reproduce these distributed traces with both high throughput or bursty patterns. Combined with SAAF, Distributed FaaSRunner can report the actual arrival time of individual requests. By measuring drift between scheduled event times and the actual event times, workload request arrival timings can be corrected by modeling network latency to enable event timing adjustments to reproduce temporal precision of workloads with

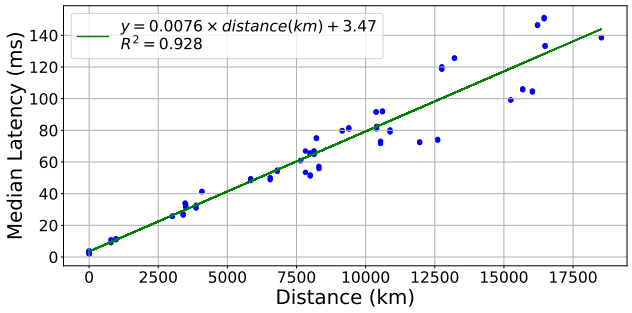


Fig. 6: Linear regression model for median latency based on distances. Following outlier pairs are excluded ((af-south-1, sa-east-1), (af-south-1, ap-southeast-2), (ap-northeast-1, af-south-1), (sa-east-1, ap-southeast-2), (eu-central-1,ap-northeast-1))

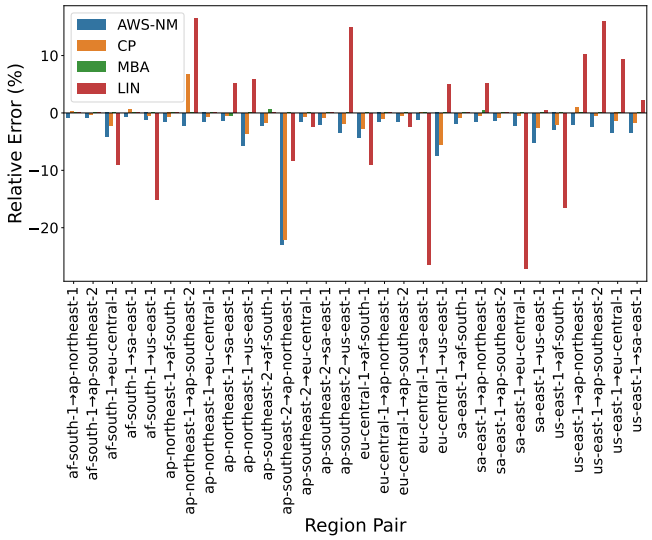


Fig. 7: Relative error (%) of predicted request arrival latency for each region pair. AWS-NM, CP, MBA, and LIN denote AWS Network Manager, CloudPing, measurement-based approach, and linear regression model. For LIN, region pairs excluded from the model training are plotted as zero.

dispersed clients. **(RQ-1 Reproducibility without Adjustment)**: Without any adjustment, we observe that the median delay between scheduled and actual request arrival time often exceeds 50–100 ms, especially between geographically distant regions, reflecting inherent network latency. Although Distributed FaaSRunner can issue requests with sub-millisecond accuracy, such end-to-end delays constrain the workload traces reproducibility at the server. **(RQ-2 Adjustment Capability)**: Applying static adjustments derived from the MBA consistently reduces the median request arrival latency to within a few milliseconds (e.g. 0-2 ms) across all tested regions. This demonstrates that such adjustments substantially improve reproducibility of scheduled event timings. **(RQ-3 Latency Prediction)**: Publicly available latency monitoring tools can predict request arrival latency with a typical relative error of less than 10% compared to ground-truth measurements. In contrast, latency estimates based solely on geographic distance using a linear regression model tend to suffer from higher prediction error (10–27%).

References

[1] M. Yan, H. Sun, X. Wang, and X. Liu, "Ws-taas: a testing as a service platform for web service load testing," in *2012 IEEE 18th Int. Conf. on Parallel and Distrib. Syst.*, 2012, pp. 456–463.

[2] T. Kondo, "Distributed FaaSRunner," https://github.com/wlloyduw/DistributedFaaSRunner.

[3] "apache jmeter," https://jmeter.apache.org.

[4] R. K. Lenka, M. Rani Dey, P. Bhanse, and R. K. Barik, "Performance and load testing: tools and challenges," in *2018 Int. Conf. on Recent Innovations in Electrical, Electronics Communication Eng. (ICRIEECE)*, 2018, pp. 2257–2261.

[5] P. Software, "Blazemeter," https://www.blazemeter.com.

[6] G. Labs, "K6: modern load testing tool for developers," https://k6.io, 2024.

[7] ——, "K6 operator," https://grafana.com/docs/k6/latest/cloud/kubernetes/operator/, 2024.

[8] ——, "Grafana cloud k6," https://grafana.com/products/cloud/k6/, 2024.

[9] R. Cordingly, N. Heydari, H. Yu, V. Hoang, Z. Sadeghi, and W. Lloyd, "Enhancing observability of serverless computing with the serverless application analytics framework," in *Companion of the ACM/SPEC Int. Conf. on Perform. Eng.*, ser. ICPE '21, 2021, p. 161–164.

[10] R. Cordingly, H. Yu, V. Hoang, Z. Sadeghi, D. Foster, D. Perez, R. Hatchett, and W. Lloyd, "The serverless application analytics framework: enabling design trade-off evaluation for serverless software," in *Proc. 2020 6th Int. Workshop on Serverless Comp.*, 2020, pp. 67–72.

[11] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conf. (USENIX ATC 20)*, July 2020, pp. 205–218, online: https://www.usenix.org/conference/atc20/presentation/shahrad.

[12] "azure public dataset," https://github.com/Azure/AzurePublicDataset.

[13] "openwhisk," https://openwhisk.apache.org/.

[14] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proc. of the 52nd Annual IEEE/ACM Int. Symp. on Microarchitecture*, ser. MICRO '52, 2019, p. 1063–1075.

[15] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: a serverless benchmark suite for function-as-a-service computing," in *Proc. of the 22nd Int. Middleware Conf.*, 2021, pp. 64–78.

[16] J. Scheuner, S. Eismann, S. Talluri, E. van Eyk, C. Abad, P. Leitner, and A. Iosup, "Let's trace it: fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications," 2022.

[17] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proc. of the 11th ACM Symp. on Cloud Comput.*, ser. SoCC '20, 2020, p. 30–44.

[18] W. J. Lloyd, "serverless application analytics framework(saaf)," https://github.com/wlloyduw/SAAF.

[19] W. Lloyd, M. Vu, B. Zhang, O. David, and G. Leavesley, "Improving application migration to serverless computing platforms: latency mitigation with keep-alive workloads," in *2018 IEEE/ACM Int. Conf. on Utility and Cloud Comput. Companion (UCC Companion)*, 2018, pp. 195–200.

[20] "amazon time sync service," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/configure-ec2-ntp.html.

[21] AWS, "viewing cloudwatch logs for lambda functions," https://docs.aws.amazon.com/lambda/latest/dg/monitoring-cloudwatchlogs-view.html.

[22] "placement groups for your amazon ec2 instances - amazon elastic compute cloud — docs.aws.amazon.com," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html.

[23] AWS, "how infrastructure performance for aws network manager works," https://docs.aws.amazon.com/network-manager/latest/infrastructure-performance/how-nmip-works.html.

[24] M. Adorjan, "cloudping," https://www.cloudping.co.

[25] A. Yachin, "aws regions geo locations," https://gist.github.com/atyachin/a011edf76df66c5aa1eac0cdca412ea9.

[26] R. Cordingly, J. Kaur, D. Dwivedi, and W. Lloyd, "Towards serverless sky computing: an investigation on global workload distribution to mitigate carbon intensity, network latency, and cost," in *2023 IEEE Int. Conf. on Cloud Eng. (IC2E)*, 2023, pp. 59–69.