Multi-Event Triggers for Serverless Computing

Valentin Carl
TU Berlin
Berlin, Germany
vc@3s.tu-berlin.de

Trever Schirmer
TU Berlin
Berlin, Germany
ts@3s.tu-berlin.de

Niklas Kowallik
TU Berlin
Berlin, Germany
nk@3s.tu-berlin.de

Joshua Adamek
TU Dortmund
Dortmund, Germany
joshua.adamek@tu-dortmund.de

Tobias Pfandzelter
TU Berlin
Dortmund, Germany
tp@3s.tu-berlin.de

Sergio Lucia
TU Dortmund
Dortmund, Germany
sergio.lucia@tu-dortmund.de

David Bermbach
TU Berlin
Berlin, Germany
db@3s.tu-berlin.de

ABSTRACT

Function-as-a-Service (FaaS) is an event-driven serverless cloud computing model in which small, stateless functions are invoked in response to events, such as HTTP requests, new database entries, or messages. Current FaaS platform assume that each function invocation corresponds to a single event. However, from an application perspective, it is desirable to invoke functions in response to a collection of events of different types or only with every n^{th} event. To implement this today, a function would need additional state management, e.g., in a database, and custom logic to determine whether its trigger condition is fulfilled and the actual application code should run. In such an implementation, most function invocations would be rendered essentially useless, leading to unnecessarily high resource usage, latency, and cost for applications. In this paper, we introduce multi-event triggers, through which complex conditions for function invocations can be specified. Specifically, we introduce abstractions for invoking functions based on a set of n events and joins of multiple events of different types. This enables application developers to define intricate conditions for function invocations, workflow steps, and complex event processing. Our evaluation with a proof-of-concept prototype shows that this reduces event–invocation latency by 62.5% in an incident detection use-case and that our system can handle more than 300,000 requests per second on limited hardware, which is sufficient load for implementation in large FaaS platforms.

KEYWORDS

function-as-a-service, function triggers, serverless workflows

1 INTRODUCTION

FaaS is a serverless cloud computing model that enables developers to focus on implementing their business logic in small, stateless functions, while the platform handles scaling and manages the execution environment [5]. Such functions are triggered by events, which can be HTTP requests, pub/sub messages, changes in databases, or events created periodically at certain times of day, among others. FaaS follows a pay-per-use model, in which providers only charge developers for the resources their application actually consumes.

FaaS has been applied to different contexts and at different scales, ranging from traditional cloud platforms in large data centers, e.g., Amazon Web Services (AWS) Lambda or Google Cloud Platform (GCP) Functions, to edge settings in which FaaS functions are deployed to smaller devices, such as Raspberry Pis and microcontrollers [14, 26]. FaaS can serve as the central level of abstraction for building entire applications with multiple functions chained together to create serverless workflows [10]. Applications of serverless computing include data processing pipelines, machine learning, video processing, scientific computing, among others [7, 8, 10, 13, 17, 24, 28].

Implementing such complex applications requires functions to invoke other functions, e.g., subsequent workflow steps. This is trivial when a function invokes a single or multiple other functions. However, invoking a single function with the results of multiple functions or other services-in essence a join or merge operation-cannot be built using today's abstractions. We refer to the concept of triggering a single function with multiple events as *multi-event triggers*.

To implement multi-event triggers today, a function has to keep state and, hence, needs access to a database for storing

individual events and manually determining whether its trigger condition is fulfilled and the actual function logic should be executed. Depending on the condition's complexity, the amount of times a function is executed to only update its event database can exceed the amount of times it executes its application logic by orders of magnitude. Due to pay-per-use billing, these additional function invocations result in high costs for application developers. This is exacerbated with expensive function cold-starts when events arrive in irregular intervals. Most importantly, handling events manually goes against the FaaS model, where the platform itself is supposed to handle the arrival and distribution of events as well as the eventual invocation of a function [1].

In this paper, we attempt to close this gap by extending the event-based trigger mechanism in serverless computing to support multi-event triggers. We aim to retain the benefits of the FaaS programming model, e.g., loose coupling, automatic scalability, and ease of development, by integrating multi-event trigger logic into the FaaS platform's function trigger mechanism. Specifically, we make the following contributions:

- We propose multi-event triggers as a novel extension of the FaaS function trigger mechanism, along with types of multi-event trigger rules useful to build FaaS applications (§3).
- We introduce the design of a multi-event trigger engine to process events and trigger rules for FaaS platforms (§4).
- We develop a prototype implementation of our multi-event trigger engine to demonstrate the feasibility of our design (§5).
- We evaluate the potential of multi-event triggers in an incident detection use case (§6).

2 FAAS TRIGGER MECHANISMS

Most FaaS platforms offer request-response patterns, pub/sub-subscriptions, or messages queues for function invocation, as well as custom—usually platform-specific—solutions for triggering FaaS functions. Virtually all FaaS platforms support synchronous triggers that use a request-response pattern, such as a RESTful interface through HTTP(S) [11, 31]. Of course, functions can also use these direct communication mechanisms to trigger other functions to build larger workflows.

Similarly, functions can be invoked asynchronously through message queues or pub/sub systems. AWS's event source mapping allows functions to process collections of events in a single invocation, resulting in a model adjacent to stream-processing.

Lastly, serverless functions can be invoked in response to events originating from other cloud services. AWS and

GCP, for example, allow the connection between FaaS functions and a variety of events their respective other services, ranging from changes to databases, at specific times, log messages, or infrastructure changes to invoking functions using email. This wide collection of different trigger mechanisms allows FaaS to be used in a large variety of contexts—as glue code for responding to or handling infrastructure changes, for processing streams of data, or for building applications in distributed and heterogeneous edge environments.

The mechanics of trigger mechanisms are essential for serverless workflow execution, as they determine when a workflow is invoked and how functions communicate. While $1 : 1$ and $1 : n$ relationships between workflow steps are trivial to implement, e.g., one function sends multiple HTTP requests to other functions, creating a join pattern, i.e., $n : 1$, is more challenging. Current approaches involve either creating an additional function that is responsible for the join step or communicating through databases [2]. In the first case, the function spends most of the invocation in an idle state, which leads to double-billing, increasing cost [1]. In the second case, additional data storage is required, which was not intended to be used for communication between functions and also increases cost [12]. In addition to higher cost, building such complex orchestration mechanisms defeats the purpose of the high levels of abstractions in FaaS platforms.

3 MULTI-EVENT TRIGGERS

With current serverless workflows, application developers still need to know a lot about how the platforms work internally to understand what kind of workloads are even possible to implement. As this is against the spirit of serverless, we propose to make it easier to write workflows by introducing multi-event triggers. These triggers make it trivial for application developers to implement a fan-in or join mechanism by specifying a *set* of events that need to be collected before a function is triggered.

While different kinds of multi-event triggers are conceivable, we focus on event counts, *ANDs*, and *ORs* for multi-event trigger rules. A necessary precondition for such rules is the existence of different event types. In general, a multi-event trigger rule is specified by one or more event types and their respective quantity. An event type can be, for example, the kind of sensor or client device that produces the event, or a more granular application-specific event type.

A multi-event trigger rule is made up of *AND* and *OR* conditions, which each contain two *amount:event type* pairs or, recursively, another rule. A formal description is shown in ?? 1.

The simplest form of such a rule is a simple count-based trigger, where every n^{th} event of type t results in a function

```
<rule>       ::= <count> ":" <type> |
    <condition> "(" <rule> "," <rule> ")"
<condition> ::= "AND" | "OR"
<count>      ::= "regexp:[0-9]+"
<type>       ::= "regexp:[a-zA-Z]+"
```

Listing 1: A formal specification of the structure of multi-event trigger rules.

```
OR(
    AND(6:temperature,6:wind),
    AND(1:temperature,1:motion)
)
```

Listing 2: An example of a trigger rule for a merge-based MET in a smart home setting. The trigger rule is fulfilled if either there a six temperature and wind measurements each or if there is one temperature measurement and a motion sensor detects a person entering the apartment.

call. Consider the following example: Temperature readings (events) are produced periodically by a sensor, e.g., once per minute. Once per hour, we want to trigger a function that uses a single measurement to store and analyze. Hence, the trigger is defined by the triple of event type *temperature*, the interval *60 min*, and the function's URL.

More complex multi-event trigger rules combine different rules. Consider the following smart home example. Assume there are three sensors in an apartment: a temperature sensor, a wind sensor, and a motion sensor that produces an event when someone enters the apartment. The wind and temperature sensors produce readings every 10 minutes. There is a function that checks—depending on the value of its inputs—whether the heater should be turned on or off. A trigger rule that implements this application is shown in **??** 2. In this example, the trigger rule is fulfilled once per hour (as both the wind and temperature sensors will each have produced six events) *or* once there is a temperature reading and the motion sensor was triggered. Once one of the two conditions is fulfilled, the function is invoked with the set of events that caused the condition to be fulfilled. The function can use them to determine whether to turn on the heater.

Internally, each trigger has one *trigger set* for each event-type that is part of its *trigger rule*. These sets grow as events arrive, and—once the trigger rule is fulfilled—events are removed from the trigger sets and sent to the function.

There are limits as to which conditions are possible. For instance, *NOT* conditions are not possible, as we cannot guarantee while triggering a function that an instance of a certain event type has not been received. This problem is exaggerated in the case of distributed, FaaS-based applications, in the case of network partitions or due to communication delays.

4 THE MET ENGINE

We present an architecture for native platform support for multi-event triggers, the MET (multi-event trigger) engine. The MET engine is a component that can be placed in front of existing FaaS platforms or that platforms can use to extend their function trigger mechanism, inhabiting a similar place to AWS's API Gateway, for example. This architecture and our prototype (§5) are designed for a multi-node, single rack setting, although the principles can be adapted to distributed or edge settings.

Figure 1 shows an overview of the MET engine's architecture. The system comprises two main components that can be scaled out individually: the dispatcher and the invoker. In addition, a load balancer serves as central entry point to the system and distributes events between dispatchers.

The dispatcher receives events from the load balancer and forwards them to the appropriate invoker, which handles trigger logic. For each MET, a new *trigger handler* is created. The trigger-handler contains a *trigger set* for each event-type in which events are stored until the function is invoked. For each event a trigger handler receives, it checks whether a trigger rule is fulfilled and, if that is the case, pulls the events from the trigger sets and invokes the function accordingly.

Dispatchers and invokers communicate using brokerless pub/sub messaging. An invoker subscribes to events at the dispatchers based on the event types it has trigger rules for. Consequently, the component handling the trigger logic (invoker) does not need to be concerned with handling events of types that do not match any of its triggers. The dispatcher can be scaled out to different nodes (e.g., other VMs) in case load on the whole system increases to the point at which a single dispatcher cannot handle it alone.

Scalability is a key concern in designing MET engine. The stateless nature of FaaS platforms allows distributed gateway designs, e.g., the frontend in AWS Lambda [5]. Introducing state management, as in our trigger sets, complicates the design of MET engine. We design the system so that it can handle the load realistically expected at a large FaaS platform. As an example, consider AWS Lambda, which receives 10s of trillions of requests per month globally. Assuming equal distribution across time and the 99 AZs, we can estimate the average load of individual AZs for Lambda to be in the high 10,000s or the low 100,000s requests per second[4]. We design the MET engine with a modular design as a distributed system to achieve these load parameters: By deploying additional invokers, the amount of triggers that can be handled
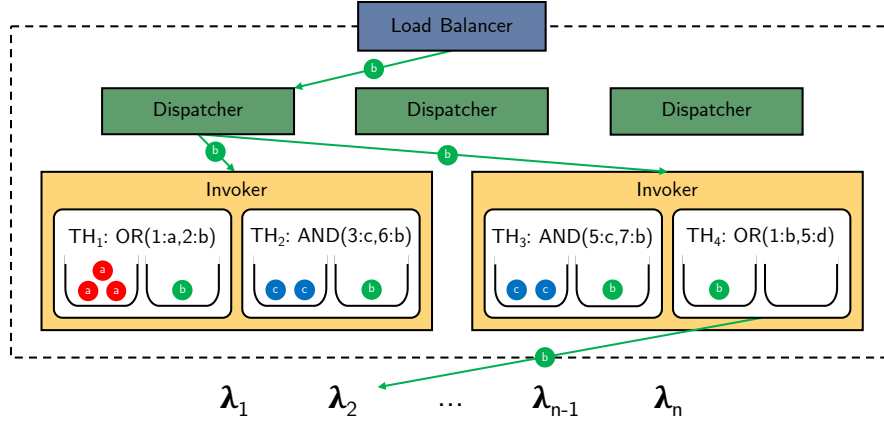
Valentin Carl, Trever Schirmer, Niklas Kowallik, Joshua Adamek, Tobias Pfandzelter, Sergio Lucia, and David Bermbach

Figure 1: An overview of the MET engine's architecture. Different client devices produce events and send them to the MET engine. A load balancer distributes events between dispatchers, which process the events by forwarding them to the invokers that handle triggers using that event type. Each invoker handles a set of triggers, stores them, checks whether the trigger condition is fulfilled, and triggers the functions accordingly.

increases. While our design currently imposes a single invoker as limit for a single trigger, we can circumvent this issue through *trigger partitioning*. By purposefully partitioning a MET into independent replicas, the amount of traffic it can handle increases even further. This is made possible because these replicas do not need to communicate events, as long as the load-balancing dispatchers send an event to only one of the replicas. While this impact the composition of event groups functions receive, this issue is negligible in cases where the order of incoming event only needs to be approximately kept. Overall, the MET engine follows a *multi-node, single rack* setting; meaning we distribute replicas of the dispatchers and invokers in the same datacenter with VMs.

5 PROTOTYPE IMPLEMENTATION

To demonstrate and evaluate our system, we implement an open-source proof-of-concept prototype.[1]

5.1 Context

In designing an architecture for handling METs at scale, we need to be able to handle the typical load large FaaS platforms experience. In this context, we use multiple nodes in the same data center, as communication between servers in the same data center is efficient compared to settings where nodes are distributed geographically. Simultaneously, this means that the maximum expected traffic for a single MET in our system is substantially less, requiring at most a single node.

5.2 Infrastructure

The dispatcher and invoker are written in Go. We deploy them to a Kubernetes[2] cluster, which we run on Google Kubernetes Engine. Doing so allows us to easily adjust the number of dispatchers and invokers we want to use. Each dispatcher and invoker runs in a separate pod. The dispatchers send events to the invokers using ZeroMQ, a popular messaging library.[3] Using a Kubernetes service of type *LoadBalancer*, we distribute events between dispatchers. The event producers publish their events by sending HTTP requests. As individual triggers are registered at an invoker by providing a trigger rule and a function URL, our prototype can trigger arbitrary FaaS functions on every platform that supports HTTP.

5.3 Trigger Rules

Invokers receive trigger rules as specified in §3, which they use to create a set of binary trees as internal representation of the MET. It is noteworthy that an invoker needs to know which part of the trigger rules caused the rule to be fulfilled, as trigger rules might contain *OR* conditions and invokers pull events from their trigger sets once the trigger rule is fulfilled. In our prototype, we implement this by creating a binary tree for each case a trigger rule contains and checking them individually as a new event arrives.

6 EVALUATION

To evaluate our proof-of-concept prototype, we perform multiple experiments. In the first experiment in §6.1, we measure

---

[1]https://github.com/valentin-carl/MultiEventTriggers

[2]https://kubernetes.io/
[3]https://zeromq.org/

```
OR(
    AND(5:packetLoss,1:temperature),
    1:powerConsumption
)
```

Listing 3: Multi-event trigger rule used in our example use case.

the event–invocation latency compared to a baseline of manually handling event logic in the functions. In the second and third experiment in Sections 6.2 and 6.3, we measure the scalability of the MET engine when increasing concurrent requests and triggers, respectively.

6.1 Experiment 1: Latency

To compare the overhead of using the MET engine versus handling events manually in a function and using an external database for keeping state, we implement a simplified data center incident detection application.

Use case. We simulate three kinds of sensors that produce periodic events of types temperature, packetLoss, and powerConsumption. We use a FaaS function to analyze the values these sensors produce and detect whether an incident occurred that requires manual intervention in the data center. The trigger rule in **??** 3 defines which combinations of events trigger the function.

The packetLoss and powerConsumption sensors produce single floating point numbers, while the temperature sensor produces an array of 25 floating point numbers to simulate different values for different racks in the data center.

Baseline. To implement multi-event triggers in FaaS applications without a dedicated MET engine, we need a database for keeping state across invocations. To this end, we use the PostgreSQL database system and deploy it to a virtual machine running in the same zone as the FaaS function to be invoked by the multi-event trigger. The function invoked for each individual event and handles the event logic itself. Consequently, most function invocations do not actually perform the incident detection, instead only resulting in a database transaction storing the new event.

Experiment setup. We use the k6 load generator[4] to simulate the sensors. The packetLoss, temperature, and powerConsumption sensors use 20, 10, and 10 virtual users and produce 180, 36, and 18 events per minute, respectively. The load generator runs for 30 minutes in each experiment run. To compare the overhead different ways of handling complex event logic introduce, we are interested in *event–invocation latency*: the duration between the creation of the event that causes a
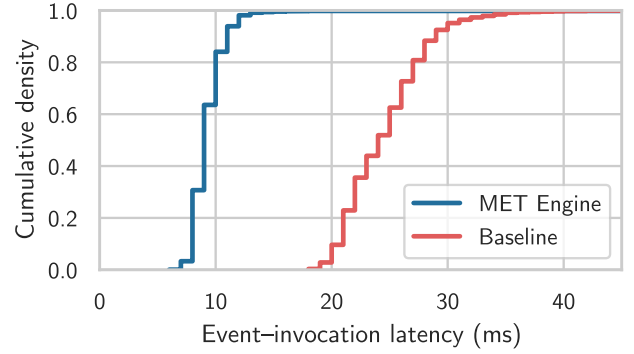
Figure 2: Cumulative distribution of latency values from the creation time of the last event that should trigger a function and the time at which the function logic actually starts. By using the MET engine over handling event logic within the function and coordinating database access across function instances, event–invocation latency is reduced by 62.5% in the median.
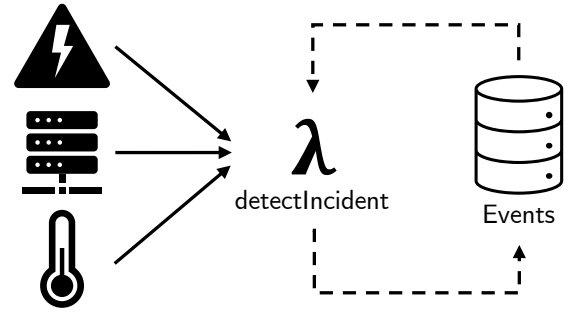

Figure 3: Baseline implementation for handling MET without the SUT. The load generator calls the function `detectIncident` for each event. The function handles the event logic in addition to the use case and uses a database for keeping state across executions.

trigger rule to be fulfilled and the start of the execution of the application logic within the triggered function. For the baseline, the function uses a timestamp created *after* it handled the trigger logic but before the application part is run. To ensure that we can compare timestamps from different sources, we deploy every virtual machine, Kubernetes cluster, and Cloud Run function in the same Google Cloud zone (europe-west10-a).

The system under test (SUT) is deployed to a Kubernetes cluster, which we run using Google Kubernetes Engine. As we only use one MET in this experiment, we only use one
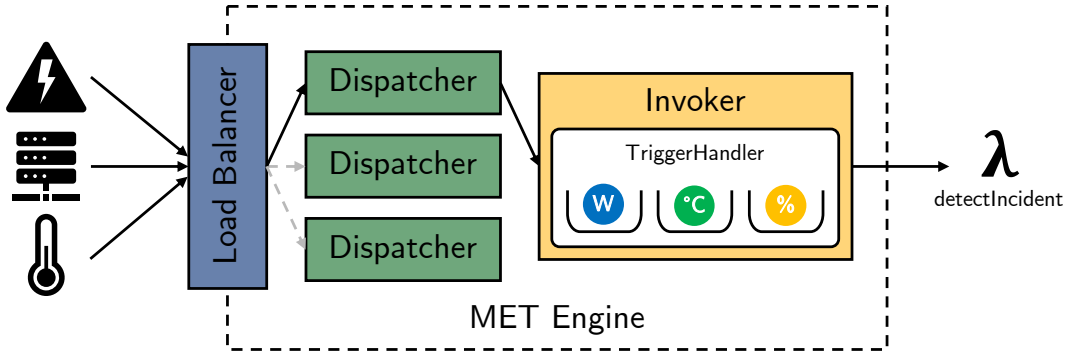
Figure 4: Experiment setup for measuring event–invocation latency with the SUT. The load generator sends requests to the MET engine's load balancer, which distributed events across three dispatchers. As we only use a single MET for the function, there is only one invoker. This invoker collects events until the trigger rule is fulfilled, after which the function `detectIncident` is called.

invoker and, hence, only one node of type `e2-standard-4`. The load balancer sends events to a load balancer, which forwards the request to one of three dispatchers. The application—comprising the `detectIncident` function—runs on Google Cloud Run functions. The baseline implementation is configured similarly, leaving out the Kubernetes cluster. The function is modified to include the event logic, and we add a virtual machine in the same zone running PostgreSQL to store events.

Results. Figure 2 shows the cumulative distribution of event–invocation latency values for both using the SUT and the baseline. In the median, using the MET engine reduces the event logic's overhead by 62.5%. Furthermore, the function is called once for every event in the baseline case, which constitutes a $4.\overline{3}$ fold increase compared to using the MET engine. This number depends on the trigger rule and the event arrival rates; it can only be calculated in cases where the event arrival rates are known.

6.2 Experiment 2: Concurrent Requests

After demonstrating the event–invocation improvements, the following experiments focus on evaluating the performance of the MET engine.

Experiment description. In the concurrent requests experiment, we evaluate how the MET engine scales with increasing amounts of concurrent requests with different cluster sizes. To this end, we deploy the system under test as a single node and in a four-node configuration. Each node is a `c7i.2xlarge` AWS EC2 virtual machine instance with 8 vCPUs and 16 GiB of memory, running in AWS's *eu-central-1* region. For load generation, we use a `c7i.16xlarge` EC2 instance with 64 vCPUs and 128 GiB of memory, running
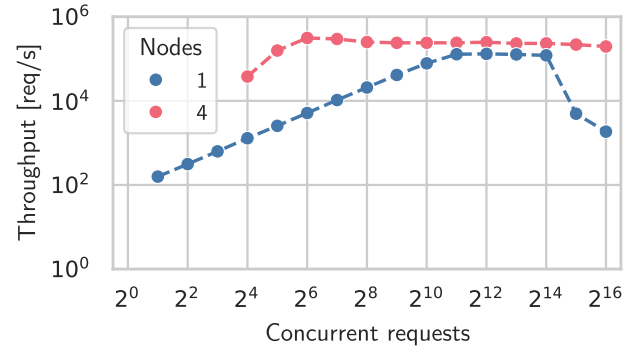

Figure 5: Throughput for different amounts of virtual users for a single-node and four-node cluster configuration. The single node reaches its maximum at 131,012.7 requests per second with 4,096 virtual users. With four nodes, maximum throughput increases to 313,154.81 requests per second.

in the same region. We create the trigger (3:a), distributed across all nodes in the cluster. As a result, a dummy function is invoked for every third event the trigger receives. In addition, each request to the MET engine contains a 1,024-byte payload of randomly generated characters. For both the configurations, we generate an increasing amount of concurrent requests. For each amount of concurrent requests and cluster size, we generate load for five minutes and calculate the throughput as the average amount of requests processed per second by the system under test. Between each such five-minute period, we leave sufficient time for the VMs to cool down.
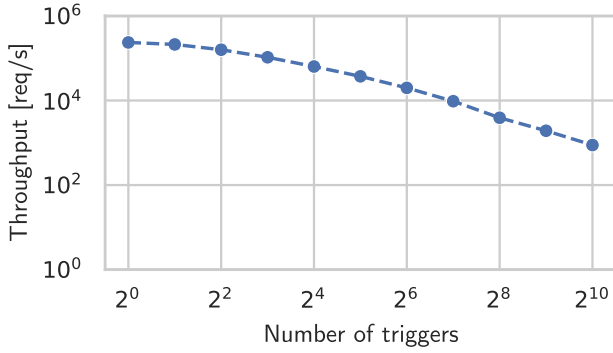
Figure 6: Throughput for increasing amount of concurrent triggers and a single invoker. While a single trigger with the trigger rule `AND(2:a,2:b)` reaches an average throughput of 236,601.77 requests per second, throughput decreases to 883.67 requests per second for 1,024 triggers on the same node.

Results. Figure 5 shows the results of the second experiment. We observe that throughput increases linearly with the amount of concurrent requests up to 2^{11}, at which point throughput is approximately 128,258.82 requests per second the single node scenario. The maximum is reached at an average of 131,012.7 requests per second for 2^{12} concurrent requests. Increasing the cluster size to 4 nodes increases maximum throughput to 313,154.81 requests per second, which is reached with 2^6 concurrent requests.

6.3 Experiment 3: Concurrent Triggers

The purpose of this experiment is to evaluate the effect of the amount of concurrent triggers on throughput within a single invoker.

Experiment description. We run one invoker only on a `c7i.large` AWS EC2 virtual machine instance with 4 vCPUs and 8 GiB of memory. For each step of the experiment, we run a constant number of copies of the same trigger for 5 minutes and measure the average throughput. After each step, we have a cooldown period for resetting the system and increase the number of triggers exponentially. To simulate having a variety of different triggers, we replicate the trigger `AND(2:a,2:b)` up to 1,024 times, each triggering a dummy function. For each configuration, we have 128 virtual users, split evenly between event types a and b. Each request contains an additional 1,024-byte payload of random numbers, similarly to the previous experiment.

Results. As expected, throughput decreases with increasing amounts of triggers running on the invoker. For a single trigger, the invoker achieves an average throughput of

236,601.77 requests per second. Increasing this to 2^3 and 2^4 results in 105,353.51 and 63,717.27 requests per second, respectively. At the end, the invoker was only able to handle 883.67 requests per second. After this point, the scheduling overhead was too large, and the system crashed. The amount of concurrent triggers on a single invoker is primarily CPU-bound. As more complex trigger rules require more computation for checking whether they are fulfilled, the invoker parallelized this process.

7 DISCUSSION AND FUTURE WORK

7.1 Platform Integration

Ideally, the MET engine would be tightly integrated into the FaaS platform itself. This enables integration into the other features of the platform, such as the component that load balances incoming function calls. We focus on a general implementation for our prototype of the MET engine instead, as our aim is demonstrating it viability for different use cases (and platforms).

7.2 Stream Processing

Stream processing and complex event processing are adjacent to multi-event triggers for FaaS. Taking a broader view, both are concerned with determining when, *and how often*, user-provided code is executed in response to incoming events. With FaaS platforms that do not use the MET engine, *every* event leads to an invocation. Stream processing platforms, however, use windowing techniques to group events, for which a user-defined operator is then executed. These engines do not include many features that developers depend on for serverless applications, such as scaling to zero, quickly adapting to bursty workloads, and support for diverse runtimes. Thus, the MET engine approaches a similar set of problems from another direction.

7.3 Geo Distribution

The current implementation of the MET engine focuses on a multi-node, single rack setting. Our evaluation shows that this setup scales enough to handle the load that AWS Lambda handles in a single availability zone. A limitation of these design decisions is that it currently does not work well for geographically distributed multi-event triggers, which require tracking trigger state over more than a single rack. In this case, the pub/sub component between invokers and dispatchers additionally needs to deal with network partitions and state drift between components. We plan to explore this in future work, using conflict-free replicated data types (CRDTs) to track incoming events.

7.4 Triggers

Multi-event triggers present a new challenge for application developers on how to design effective trigger rules. While the two triggers presented in this work (OR and AND) already enable complex use cases, we envision that application developers can also benefit from other types, such as XOR.

Another challenge is how to deal with synchronization and consistency in case of downtimes. For example, using the trigger rule from ?? 2 where six temperature and six wind measurements were required, what happens if the wind sensor skips sending one event? Currently, the MET engine would trigger the function with out-of-date temperature events (as it has to wait for an additional wind sensor event). One possible solution to this problem is giving events a time-to-live, after which they can not trigger a function anymore.

8 RELATED WORK

In most FaaS platforms, function invocations are stateless and correspond to single events. Consequently, triggering a function in response to a collection of events only requires maintaining state to determine when a trigger condition is met [25]. Several approaches have introduced state management to FaaS. *Crucial* [3] adds a distributed shared object layer, essentially treating function instances as threads at larger scale and persisting state via an external server and S3 storage. *Cloudburst* [33] integrates an autoscaling key-value store, co-locating data and functions to improve performance over public object storage. Other systems employ alternative strategies for state management: *Boki* [16] introduces a serverless runtime that enables functions to persist and share state via a log-based API. *Faasm* [32] leverages a WebAssembly runtime where function instances (*Faaslets*) can share memory regions. *Photons* [9] achieves state sharing by co-locating function instances within the same execution environment. *Enoki* [27] combines an edge-native FaaS platform with geo-distributed data management for local data access for every function. These stateful FaaS solutions could be adapted for multi-event triggers by maintaining state between function instances.

Our approach similarly uses an external component to keep state and store events that have not yet contributed to a function invocation. However, unlike stateful FaaS systems, which nonetheless invoke functions for every incoming event, our system invokes functions only when a complete trigger condition is met, reducing the amount of unnecessary executions. Additionally, using stateful FaaS would require a locking mechanism to prevent race conditions when function instances communicate via shared storage. Our design avoids this issue by avoiding redundant invocations and handling event logic in the MET engine as part of the trigger mechanism itself.

FaaS function triggers have found further interest in the literature. Scheuner et al. [30] benchmark FaaS triggers across multiple cloud platforms, finding that HTTP-based triggers outperform storage-based triggers, which suffer from higher latency. This is additionally supported by Quinn et al. [29], who observe significant delays when using object storage for workflow orchestration. These findings further highlight the need for alternative state management and communication methods between function instances, especially for creating complex condition for when the application logic should be executed.

Function triggers serve as a central communication mechanism in serverless environments [30]. Since serverless functions lack direct communication capabilities [15], triggers play a crucial role in orchestrating workflows. As outlined in §2, workflows are either managed by a centralized orchestrator or implemented as decentralized function choreographies, resulting in different workflow structures and execution patterns.

Taibi et al. [34] describe the fan-in/fan-out pattern, also known as fork/join, where a task is divided into parallel executions across multiple function instances. While most public FaaS platforms provide workflow engines (e.g., AWS Step Functions and Google Cloud Workflows), these solutions often exhibit inefficiencies for fork/join workflows [2]. Among public platforms, IBM Composer was identified as the most effective for this use case [2], but its deprecation alongside IBM Cloud Functions removed it as an option.

Several alternative approaches have been proposed, differing primarily in their scheduling mechanisms. *Trigger-Flow* [22] provides a framework for constructing custom workflow engines on Knative, where triggers are represented as user-defined Python callables, allowing for flexible trigger conditions. Other systems distribute workflow scheduling across FaaS worker nodes, as seen in *FaaSFlow* [19]. *DataFlower* [20] moves beyond control-flow-based orchestration by scheduling functions based on data availability, ensuring that functions execute only when all required inputs are ready, thereby reducing idle wait times.

In the domain of machine learning workflows, *Cirrus* [7] employs a client-side backend for scheduling, while *PyWren* [18] coordinates fork/join steps using external storage and virtual machines for parallel data processing. Alternatively, some approaches eliminate the need for a centralized orchestrator altogether, instead leveraging function-level libraries to enable choreography-based workflows [6, 21].

Serverless computing has also gained traction in scientific workflows due to its potential for high parallelism and scalability. Elshamy et al. [12] compare serverless and traditional serverful orchestration approaches, highlighting the necessity of additional storage layers in both models. They identify two strategies for implementing a reduce stage: (1) using a

dedicated function that periodically checks for completion and (2) embedding workflow logic within functions so that each instance verifies completion at the end of execution. Similarly, Malawski et al. [23] evaluate scientific workflows on public FaaS platforms and find comparable performance to serverful alternatives, further demonstrating the viability of serverless architectures in data-intensive applications.

A central problem serverless workflows face is that functions lack directly communication, which requires external storage or non-serverless components for coordination. By extending the function trigger mechanism, we propose multi-event triggers as an alternative.

9 CONCLUSION

In this paper, we introduced multi-event triggers as a novel extension to the event-based invocation mechanism in current FaaS platforms. By enabling functions to be triggered based on complex combinations of multiple events—rather than individual occurrences—MET addresses a fundamental limitation in the current design of serverless workflows, namely the lack of native fan-in support and the inefficiencies associated with manually implementing such behavior at the application level.

To realize this concept, we proposed the MET engine, which can be integrated with existing FaaS platforms and decouples the evaluation of complex trigger conditions from the function logic itself. Our evaluation demonstrates that this approach significantly reduces invocation latency—by 62.5% in an incident detection use case, while also lowering resource consumption by reducing the number of superfluous function invocations.

Furthermore, our scalability experiments show that the MET engine can sustain high throughput under load, reaching over 300,000 requests per second in a four-node configuration. These results indicate that the proposed architecture is suitable for deployment in large-scale cloud environments and can be integrated with minimal impact on existing systems.

In doing so, this work preserves core benefits of the serverless paradigm, such as loose coupling, automatic scalability, and minimal operational overhead, while enabling more expressive and efficient event-driven workflows. Future work will focus on extending support for geographically distributed deployments, extending the expressiveness of trigger rule specifications, and exploring techniques for improved synchronization, fault tolerance, and consistency across distributed trigger handlers.

REFERENCES

[1] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The serverless trilemma: function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, Vancouver BC Canada, 89–103. https://doi.org/10.1145/3133850.3133855

[2] Daniel Barcelona-Pons, Pedro García-López, Alvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. 2019. FaaS Orchestration of Parallel Workloads. In *Proceedings of the 5th International Workshop on Serverless Computing*. ACM, Davis CA USA, 25–30. https://doi.org/10.1145/3366623.3368137

[3] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference*. ACM, Davis CA USA, 41–54. https://doi.org/10.1145/3361525.3361535

[4] Jeff Barr. 2024. AWS Lambda turns ten – looking back and looking ahead. https://aws.amazon.com/blogs/aws/aws-lambda-turns-ten-the-first-decade-of-serverless-innovation/

[5] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. 2023. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 315–328.

[6] Valentin Carl, Trever Schirmer, Tobias Pfandzelter, and David Bermbach. 2024. GeoFF: Federated Serverless Workflows with Data Pre-Fetching. In *2024 IEEE International Conference on Cloud Engineering (IC2E)*. 144–151. https://doi.org/10.1109/IC2E61754.2024.00023 ISSN: 2694-0825.

[7] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: a Serverless Framework for End-to-end ML Workflows. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, Santa Cruz CA USA, 13–24. https://doi.org/10.1145/3357223.3362711

[8] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. 2020. funcX: A Federated Function Serving Fabric for Science. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, Stockholm Sweden, 65–76. https://doi.org/10.1145/3369583.3392683

[9] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. 2020. Photons: lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. ACM, Virtual Event USA, 45–59. https://doi.org/10.1145/3419111.3421297

[10] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2021. A Review of Serverless Use Cases and their Characteristics. https://doi.org/10.48550/arXiv.2008.11110 arXiv:2008.11110 [cs].

[11] Simon Eismann, Joel Scheuner, Erwin van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2022. The State of Serverless Applications: Collection, Characterization, and Community Consensus. *IEEE Transactions on Software Engineering* 48, 10 (Oct. 2022), 4152–4166. https://doi.org/10.1109/TSE.2021.3113940 Conference Name: IEEE Transactions on Software Engineering.

[12] Abdallah Elshamy, Ahmed Alquraan, and Samer Al-Kiswany. 2023. A Study of Orchestration Approaches for Scientific Workflows in Serverless Computing. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies*. ACM, Rome Italy, 34–40. https://doi.org/10.1145/3592533.3592809

[13] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Anirudh Sivaraman, and George Porter. [n. d.]. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. ([n. d.]).

[14] Gareth George, Fatih Bakir, Rich Wolski, and Chandra Krintz. 2020. NanoLambda: Implementing Functions as a Service at All Resource Scales for the Internet of Things. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. 220–231. https://doi.org/10.1109/SEC50012.2020.00035

[15] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. (2019).

[16] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. ACM, Virtual Event Germany, 691–707. https://doi.org/10.1145/3477132.3483541

[17] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the 2021 International Conference on Management of Data*. ACM, Virtual Event China, 857–871. https://doi.org/10.1145/3448016.3459240

[18] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, Santa Clara California, 445–451. https://doi.org/10.1145/3127479.3128601

[19] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. FaaSFlow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Lausanne Switzerland, 782–796. https://doi.org/10.1145/3503222.3507717

[20] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. 2023. DataFlower: Exploiting the Data-flow Paradigm for Serverless Workflow Orchestration. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. ACM, Vancouver BC Canada, 57–72. https://doi.org/10.1145/3623278.3624755

[21] David H Liu, Amit Levy, and Sebastian Burckhardt. 2023. Doing More with Less: Orchestrating Serverless Applications without an Orchestrator. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*. ACM, Boston.

[22] Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard. 2020. Triggerflow: trigger-based orchestration of serverless workflows. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. ACM, Montreal Quebec Canada, 3–14. https://doi.org/10.1145/3401025.3401731

[23] Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, and Kamil Figiela. 2020. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems* 110 (Sept. 2020), 502–514. https://doi.org/10.1016/j.future.2017.10.029

[24] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. ACM, Portland OR USA, 115–130. https://doi.org/10.1145/3318464.3389758

[25] Tobias Pfandzelter and David Bermbach. 2019. IoT Data Processing in the Fog: Functions, Streams, or Batch Processing?. In *2019 IEEE International Conference on Fog Computing (ICFC)*. 201–206. https://doi.org/10.1109/ICFC.2019.00033

[26] Tobias Pfandzelter and David Bermbach. 2020. tinyFaaS: A Lightweight FaaS Platform for Edge Environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*. 17–24. https://doi.org/10.1109/ICFC49376.2020.00011

[27] Tobias Pfandzelter and David Bermbach. 2023. Enoki: Stateful Distributed FaaS from Edge to Cloud. In *Proceedings of the 2nd International Workshop on Middleware for the Edge* (Bologna, Italy) (*MiddleWEdge '23*). ACM, New York, NY, USA, 19–24. https://doi.org/10.1145/3630180.3631203

[28] Qifan Pu and Ion Stoica. [n. d.]. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. ([n. d.]).

[29] Sterling Quinn, Robert Cordingly, and Wes Lloyd. 2021. Implications of Alternative Serverless Application Control Flow Methods. In *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*. ACM, Virtual Event Canada, 17–22. https://doi.org/10.1145/3493651.3493668

[30] Joel Scheuner, Marcus Bertilsson, Oskar Grönqvist, Henrik Tao, Henrik Lagergren, Jan-Philipp Steghöfer, and Philipp Leitner. 2022. TriggerBench: A Performance Benchmark for Serverless Function Triggers. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 96–103. https://doi.org/10.1109/IC2E55432.2022.00018

[31] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference*. ACM.

[32] Simon Shillaker and Peter Pietzuch. [n. d.]. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. ([n. d.]).

[33] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: stateful functions-as-a-service. *Proceedings of the VLDB Endowment* 13, 12 (Aug. 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[34] Davide Taibi, Nabil El Ioini, Claus Pahl, and Jan Niederkofler. 2020. Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review:. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, Prague, Czech Republic, 181–192. https://doi.org/10.5220/0009578501810192