# Cloud-Scale Runtime Verification of Serverless Applications

Kalev Alpernas*
Tel Aviv University
kalev.alp@gmail.com

Aurojit Panda
NYU

Leonid Ryzhyk
VMware Research

Mooly Sagiv*
Tel Aviv University

## Abstract

Serverless platforms aim to simplify the deployment, scaling, and management of cloud applications. Serverless applications are inherently distributed, and are executed using short-lived ephemeral processes. The use of short-lived ephemeral processes simplifies application scaling and management, but also means that existing approaches to monitoring distributed systems and detecting bugs cannot be applied to serverless applications. In this paper we propose Watchtower, a framework that enables runtime monitoring of serverless applications. Watchtower takes program properties as inputs, and can detect cases where applications violate these properties. We design Watchtower to minimize application changes, and to scale at the same rate as the application. We achieve the former by instrumenting libraries rather than application code, and the latter by structuring Watchtower as a serverless application. Once a bug is found, developers can use the Watchtower debugger to identify and address the root cause of the bug.

**ACM Reference Format:**
Kalev Alpernas, Aurojit Panda, Leonid Ryzhyk, and Mooly Sagiv. 2021. Cloud-Scale Runtime Verification of Serverless Applications. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3472883.3486977

## 1 Introduction

Serverless, or Functions-as-a-Service (FaaS) is a recent architecture for building elastically scalable applications that can be deployed on the cloud. This architecture is now widely supported, and serverless applications can be deployed on most public clouds [23, 51, 72], on shared edge infrastructure [37, 43], and in private datacenters [14]. The serverless architecture provides elastic scaling by allocating application resources in response to requests, which in turn ensures that resources allocated to an application are *proportional* to the number of requests being processed. Ensuring that resource consumption is proportional to workload is particularly important for applications where demand varies widely over time, *e.g.,* web applications that need to deal with diurnal pattern [82, 87, 94]. There is evidence that deployed serverless applications experience a similar variance; a recent study from Microsoft Azure [90] observed that request rates in applications can vary by up to 8 orders of magnitude. The use of the serverless architecture thus reduces deployment costs for such applications without impacting performance or utility.

Serverless platforms achieve their scalability by using ephemeral processes to run application code. Serverless applications are split into functions which are *triggered* by external events and run in ephemeral processes that can execute for a finite amount of time, generally ranging from several milliseconds [37] to an hour or more [73]. A single application might have several concurrent processes executing at a time. The use of ephemeral executors has significant impact on how applications are designed: first, the presence of concurrent processes means that all applications are inherently distributed; second, ephemeral processes mean that applications rely on cloud provider services (*e.g.,* S3 and DynamoDB) to store application state; and third, the number of processes varies over time and application logic cannot make assumptions about the number of processes running at a time, nor about their placement. We discuss these implications in greater detail in §2.1, but note here that all serverless applications are *distributed systems* comprised of functions with differing functionality.

Ensuring correctness (§2.2) for distributed applications, and even detecting buggy behavior is challenging. This is because correctness properties can require reasoning about the behavior of several isolated processes. Several research projects including D3S [67], Pip [84], X-Trace [46], and DInv [53] have proposed techniques for reasoning about the correctness of deployed distributed systems. Many of these techniques have

*Work done partially while at VMware Research.

been adopted in practice [81] and incorporated in tools such as Dapper [92] and Zipkin [96]. Unfortunately, applying these existing tools and techniques to the serverless context is challenging. This is because existing techniques either rely on vector clocks in messages to construct a causally consistent log of the application's runtime, or rely on consistent snapshot algorithms such as Chandy-Lamport [34]. As we describe later in §2.3, using vector clocks in serverless applications is challenging due to the pervasive use of cloud-provider services and also poses a scalability challenge due to the use of ephemeral executors. On the other hand, computing a consistent snapshot is also challenging because application code is unaware of the set of concurrent executors running at any point in time and executors cannot directly communicate with each other.

In this paper we develop *Watchtower*, a runtime verification tool that can be used to identify bugs in serverless applications.

Watchtower accepts as input one or more safety properties (§2.2), and then monitors and analyzes the application at runtime to detect violations. Similar to Pip and others, Watchtower analyzes application logs in order to detect property violations. This requires that applications be instrumented to produce logs. However, adding instrumentation can be a hindrance to adoption. We observe that most serverless applications use external cloud provider services to store program state, and access these services through a SDK that serves as a *narrow waist* for these applications: the 2020 Serverless Community Survey [38] found that 49.91% of respondents use AWS DynamoDB and 14.8% use AWS Aurora for application state. Both services are accessed through the AWS SDK. We make use of this insight by providing a mechanism (§3.3) that allows us to instrument SDK calls without requiring changes to the SDK library. This instrumentation can be reused across applications, reducing the instrumentation burden and facilitating easier adoption. Our current implementation includes instrumentation for the AWS SDK, as well as several other popular SDKs. Applications using these SDKs do not need to be manually instrumented. Application developers can also add additional application-specific logging. In our evaluation (§6) we did not need to add any application-specific logging.

Next, given a set of logs, Watchtower needs to reconstruct a single causally ordered log. Our approach for recovering causal ordering between application events during monitoring uses synchronized *wall-clock time* rather than *logical clocks*. While wall-clock time is insufficient for ordering events in most distributed systems, we show that it is sufficient for serverless applications. This is because processes (functions) in a serverless application communicate through the use of storage or queuing services which introduce several milliseconds of latency for message delivery, and NTP synchronization provides sufficient synchronization accuracy at these time scales. In our evaluation (§6) we encountered no scenario where events could not be ordered using wall-clock time. Nevertheless, in case two events do occur in such close temporal proximity that they cannot be ordered using wall-clock time, Watchtower can overcome this issue by considering all possible interleavings between such events. We discuss this in greater detail in §3.5.

Finally, Watchtower analyzes the causally-ordered log to check for property violations. Scaling property checking is one of the key challenges faced by systems like Watchtower, which are designed to work with serverless applications. Our approach to doing so is to build monitoring as a serverless application, thus allowing Watchtower to inherit the same scaling behavior as the application which uses it. We evaluate this approach in §7 using 6 open-source serverless applications, a variety of correctness properties and request rates. Our evaluation demonstrates that Watchtower can correctly identify instances where programmer-specified properties are violated and can scale to match application load *with no degradation of performance as scale increases* to rates of 1000 requests per second[1].

While identifying violations is useful, addressing them requires programmers to identify a root-cause. Watchtower aids with this process by providing a replay-debugger (§4). Print and log-based debugging is the main tool available to serverless developers today, and thus Watchtower's debugger is useful independent of its property checking capabilities.

## 2 Background

We begin by providing some background that explains the serverless computing model, the type of program properties we consider, and challenges with using existing techniques for monitoring distributed systems in the serverless context.

### 2.1 Serverless computing

First, we provide a brief background about where serverless has been used, and present some of the limits and features of this architecture. Serverless was originally designed as a way to build event-driven applications for deployment on cloud environments, and since then has been used for a variety of uses including group chat and mail services [4], trading platforms [9], e-commerce [2], and blogging services [6, 8]. Additionally, several recent academic efforts have shown how to use serverless in application domains including numeric computing (PyWren [62], NumpyWren [91], etc.), distributed builds (gg [47]), and video recompression (ExCamera [48]). Many of these uses, *e.g.,* PyWren and NumpyWren, go beyond the event-driven applications that serverless originally targeted. This is due to the advantages offered by the serverless architecture, which enables automatic *demand-driven resource provisioning* and fine-grained pay-as-you-go billing, thus greatly simplifying the task of deploying an application.

[1]Our experiments were restricted by the AWS Lambda global limit of 1000 concurrent function invocations.

Most of the benefits of the serverless architecture result from its adoption of an execution model based on **ephemeral processes**. In this execution model application logic is implemented as several independent functions, and executed in several short-lived containers. Functions are triggered by events including web requests, file system writes, and changes to a database, and each function once triggered executes for a bounded amount of time ranging from a few milliseconds [37] to a few hours [73], with AWS Lambda providing an execution bound of 15 minutes. Additionally, applications cannot make assumptions about placement, *i.e.,* an application's correctness cannot depend on assumptions about *where* a function is executed.[2] The use of ephemeral processes imposes several additional constraints on how serverless applications are written. First, the absence of long lived processes means that applications must rely on external services such as blob stores (AWS S3 [22] or Azure Blob Storage [71]), or databases (AWS DynamoDB [21], AWS Aurora [20], or GCP Cloud Spanner [52]) for stable storage, on services such as API gateways (AWS API Gateway [19], or Azure API Management [70]) to implement user accessible endpoints, etc. As a result, the use of cloud provider services is pervasive in serverless applications. Second, the degree of concurrency in these applications is proportional to the workload. This impacts both how applications are written, and the complexity of reasoning about their correctness. In terms of how applications are written, application logic is normally unaware of the degree of concurrency, or their network address and thus all communication between functions in a serverless application is performed by *reading* and *writing* from stable storage.[3] As we explain next the varying degree of concurrency also makes it harder to monitor and reason about application correctness.

## 2.2 Program properties

In this paper we consider the problem of detecting *safety property* violations in serverless applications. In particular we focus on properties whose violation can be observed in a causal trace of the application's execution. We consider both properties encapsulating correctness requirements for the application, and legal requirements for deployment. To make this more concrete, below we list three example properties:

(1) A simple serverless mailing list application (similar to Moonmail [4]) consisting of three functions: one that subscribes users to a list, a second that unsubscribes users from a list, and a third that sends e-mails to all subscribed users. A correct implementation of this application must ensure that e-mails sent to a list are only delivered to subscribed users, which translates to ensuring that e-mails are only delivered to users who have subscribed to a list, and have not subsequently unsubscribed. This property is a safety property, we can evaluate it each time an e-mail is sent, and requires reasoning about the order in which the three functions are called.
(2) Applications often use audit logs as a way to track access to sensitive data. Maintaining an audit log requires ensuring that the program logs any access to sensitive data, in a serverless application this can require changing all functions. A bug or missing log in any function can invalidate the audit log. Checking that accesses are logged is a safety property, and Watchtower can notify developers of violations.
(3) Several recent laws including GDPR [1] and CCPA [66] require the application to obtain user consent before processing user data. When implementing this requirement in a serverless application, developers generally add a function that is responsible for recording whether or not the user consents to data use. In this case Watchtower can be used to monitor and detect cases where a function accesses data without consent. This requires looking at causal logs across at least two functions: the function that records consent and function(s) accessing data.

Watchtower is designed to detect violations of general safety properties similar to the ones we described above. In order to do so it needs to reason about events occurring in different functions and potentially over wide time spans, which it does by reconstructing an ordered log, and checking for violations in this log.

**Threat model** Watchtower assumes a non-adversarial setting where functions correctly execute code provided by the developer, log messages are not lost, and no log messages are deleted until explicitly garbage collected by Watchtower.

## 2.3 Monitoring Distributed Applications

Detecting correctness and performance problems in distributed systems has been studied extensively. This area of research has led to the development of tools and standards including Dapper [92], Zipkin [96], Canopy [63] and OpenTracing [80] which are in use at large internet companies (including Facebook, Google, and Twitter) and are incorporated into widely used libraries such as GRPC. In this section, we briefly discuss the challenges with directly applying these existing approaches to serverless applications.

Existing work has taken two approaches to monitoring distributed systems for bugs: The first, are techniques that collect causally ordered logs and analyze them (either automatically

[2]Many implementation, including ones from Amazon, Azure and Google reuse containers across functions, and some applications use temporary storage in reused containers to improve performance. However, for correctness these applications must also handle the case where a container is not reused.
[3]There are a few notable exceptions such as gg [47] where application logic is responsible for triggering all concurrent functions and hence can make assumptions about the degree of concurrency and use NAT hole punching to allow direct communication between functions.

or manually) in order to detect bugs. These techniques have been used widely in research in efforts including Pip [84], X-Trace [46], DISTALYZER [76], and X-Ray [16]; and have also seen deployment as a part of systems including Dapper, Zipkin and OpenTracing. All of these techniques rely on vector clocks [44, 68] attached to logged events in order to reconstruct a causal sequence of events. As Fidge [44] noted in his original work on vector clocks, they can be extended to accommodate new processes by growing the vector when a new process joins and assuming an implicit value of 0 for timestamps before this occurs. One cannot, however, safely *remove* an element from the vector. Serverless applications comprise short-lived processes launched in response to requests, and several concurrent processes can execute the same function. This means that we need to track causality at the level of an ever-growing set of processes, which in turn results in vector clocks of unbounded size, which poses a practical challenge with adopting this technique. Techniques such as resettable vector clocks [15] that bound vector clock size can only be used when applications meet specific semantic conditions (*e.g.,* bounded histories and frequent all-to-all communication for RVCs), and cannot be applied more generally.

The second line of work builds on the use of consistent snapshots [34] which embody state at all processes in a distribute system. Prior work such as D3S [67] use the Chandy-Lamport algorithm to compute a distributed state snapshot and evaluate properties on process states. This approach does not require logging, and hence does not require the use of vector clocks. However, collecting consistent snapshots in a serverless application is challenging because (a) processes do not directly communicate with each other; (b) the number of processes in the system changes rapidly; and (c) the application does not control the rate of process creation nor can it pause process creation. As a result snapshot based approaches also cannot be directly applied to the serverless context.

In our work we chose to use the log-based approach since the properties we considered were more easily expressed in terms of events as we highlight next. To apply this approach to serverless application we eschew the use of vector clocks and instead use wall-clock time.

## 3 Design

Figure 1 shows Watchtower's overall architecture and workflow. Our current implementation targets serverless applications running on AWS Lambda, and written in JavaScript. For ease of exposition we use AWS services when describing some of our design choices, and focus on functions run under the JavaScript execution model [56]. Our techniques can be naturally applied to other cloud providers and languages.

Watchtower takes as input a set of properties (§3.2) and an application. The application needs to be instrumented (§3.3) so that each function produces a sequence of structured events (§3.1) that are logged to a *logging service.* In our current implementation we support using AWS Kinesis [18] and AWS Cloudwatch [17] for logging. The addition of structured events to the logging service triggers Watchtower's **Event Ingest** (§3.4) function. The Event Ingest function is responsible for reading events and adding them to a database which maintains an *index* to allow for efficient event retrieval when checking properties. The addition of *terminal events* (defined in §3.5) to the database triggers the **Property Checker** (§3.5) function which reads previously indexed events in order to determine whether or not a property violation has occurred. Watchtower generates a notification if a violation is found, and developers can use Watchtower's replay debugger (§4) to identify and address its root-cause.

Running event ingest and property checking in separate functions which are outside of the application's critical path allows Watchtower to minimize the impact property checking has on application performance. Implementing this functionality using serverless functions ensures scalability, and minimizes deployment cost. These choices improve performance and scalability for Watchtower. However, they also mean that Watchtower cannot *prevent* or mitigate violations.

### 3.1 Structured events

Watchtower analyzes logs produced by the functions comprising an application in order to detect property violations. We require that these function logs be structured as sequences of *structured events*, and be written to a logging service such as AWS Kinesis or AWS Cloudwatch. We neither require logs to be shared across functions, nor do we assume any consistency guarantees when reading events from different logs.

Each *structured event* specifies and event ID, a set of event specific parameters (§3.3), and a timestamp. The structured events in a log can be produced either from Watchtower's SDK instrumentation (§3.3) or from logging statements embedded in application code. In both cases we assume that the timestamp associated with each structured event denotes when the event occurred, and that timestamps are synchronized across the application (using NTP). Finally, as we discuss in §5, we assume that an event once logged is visible to Watchtower within a bounded amount of time, and that this bound can be empirically determined. This assumption holds due to the observation that datacenter networks are partially synchronous [41] and SLAs provided by AWS Kinesis and similar services ensure visibility in bounded times.

### 3.2 Property specification

Program properties (§2.2) in Watchtower are specified as a *family* of *finite state machines* whose *transitions* are specified in terms of IDs and parameters for structured events. The family of state machines is parameterized using structure event parameters, and a property instance (a single state

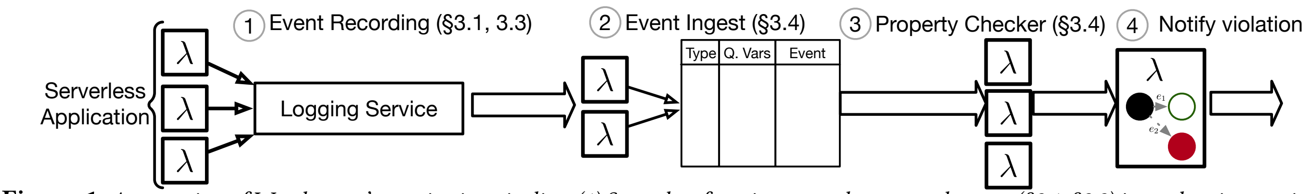

**Figure 1:** *An overview of Watchtower's monitoring pipeline: (1) Serverless functions record structured events (§3.1, §3.3) into a logging service. (2) The logging service triggers an ingest function (§3.4); the ingest function adds events to a database that enables quick retrieval of events associated with a property instance (§3.2). (3) When a* terminal event *is added to the database, a function is triggered to check the property instance (§3.5); the function waits for a fixed period to ensure log visibility, and then reads events from the database and determines if the property has been violated. (4) In the event of a property violation we notify the operator.*

```
1  {name: 'promotional',
2   quantifiedVariables: ['user_id', 'email_subject'],
3   states: ['consented'],
4   stateMachine: {
5     'CONSENT': {
6       params: ['user_id'],
7       'INITIAL' : {
8         to: 'consented'
9       },
10    },
11    'SENT_EMAIL': {
12      params: ['user_id', 'email_subject'],
13      // Starts with 'PROMOTION'
14      guard: (email_subject)
15              => email_subject.match(/^PROMOTION:/)
16      'INITIAL': {
17         to: 'FAILURE'
18      },
19      'consented' : {
20         to: 'SUCCESS'
21      }}}}
```

**Listing 1:** *Example policy for an e-mail application.*

machine) is instantiated for every unique combination of parameters. Property instances start out in the `INITIAL` state, and can terminate in either a `FAILURE` state, indicating that the property has been violated, or a `SUCCESS` state, indicating that no subsequent actions can ever violate the property.

Developers specify properties in JSON-based property language (see e.g., Listing 1), and provide the following: (a) property name; (b) property parameters (`quantifiedVariables`); (c) a set of states; and (d) a transition function.

Listing 1 shows an example program property named `promotional`. This property is parameterized by `user_id` and `email_subject` (l2), and a new property instance is instantiated for each unique combination of `user_id` and `email_subject`, *i.e.,* whenever a promotional e-mail (uniquely identified by a subject) is about to be sent to a user. The property instance begins in a designated `INITIAL` state. The transition function says that if the property instance is currently in the `INITIAL` state, and a `CONSENT` event whose `user_id` parameter equals the instance's user ID is logged, then the property instance should transition to the `consented` state (l7-l9). Similarly, if the property instance is in the `INITIAL` state, and a `SENT_EMAIL` event is logged where the `user_id` matches the instance's user ID, and where e-mail subject begins with the string "PROMOTION:", the property instance transitions to the `FAILURE` state indicating a violation (l16-l18). A similar `SENT_EMAIL` event occurring in the `consented` state results in the property instance transitioning to the `SUCCESS` state (l19-l21). Guards (l14-15) allow us to further restrict the set of events that trigger a transition. The order in which transitions are performed dictates whether the property instance arrives at the success or failure state.

**Property expressiveness and limitations** Watchtower can only be used to monitor safety properties that are expressible as finite state machines. Additionally, the transition function cannot read application program state (*e.g.,* local variables), and must infer state from past events. Watchtower's property checking algorithm also requires that all property parameters appear in any terminal event, this information is used to identify the property instance that should be evaluated (§3.5). Despite these constraints a wide variety of practical properties can be checked with Watchtower (§6).

## 3.3 Event recording

Our approach to event recording, which we introduced in §1, is to instrument SDK libraries used by developers when accessing cloud provider services including storage services like S3 and DynamoDB, and API gateways. The use of these SDKs is pervasive in serverless applications, and thus instrumenting the libraries implementing these SDKs automates logging for most serverless applications. This is similar to the approach adopted by DTrace [54] and Linux's eBPF tracing tools [33], where system-call instrumentation is used to enable debugging and analysis of a wide range of applications.

A naïve approach to instrumenting SDK libraries would be to manually change them to add calls that log events. Unfortunately this instrumentation would need to be redone every time the library is updated; in August and September 2020 a new AWS SDK library version was released several times a week [25]. The use of updated library versions is often mandated by organizational security policies, and changes to the library itself are thus infeasible.

Thus, instead of manual instrumentation, we use JavaScript's proxy mechanism [61] to add logging *without* changes to library code. JavaScript proxies are a mechanism by which programs can register callbacks that are invoked whenever a method is called on an object. In server-side JavaScript, libraries are loaded as objects. Proxies can thus be used to record the call, all parameters passed to the call, and return values. We use VM2 [93], a Node.js library, to attach proxies when loading libraries.

In order to write a proxy, one only needs to know function names and signatures, and these external interfaces tend to be more stable than the library itself. Watchtower currently provides proxies for a few libraries including the AWS SDK. Adding proxies for additional libraries merely requires emitting a structured event whose event ID encode the external service and API, and whose parameters record arguments passed into the call and values returned from the call.

While library proxies automate application instrumentation, the resulting events may be hard to work with when writing properties. For example, the `CONSENT` event in Listing 1 corresponds to one or more database writes from the application. These application-specific events can thus enable the use of more compact and easily understood policies. Therefore, Watchtower allows application developers to provide application-specific semantic translations, that turn SDK events into application-specific events. Consequently, program properties can use a combination of application-specific and SDK events.

Finally, application developers can also manually produce structured log events from the application code, to record events that do not correspond to SDK or library interactions. In our evaluation (§6) we did not require the use of manual logging.

## 3.4 Event ingest

In order to check a program property Watchtower needs to correlate and order logs from different functions. For example, checking the property in Listing 1 requires determining the relative order between a user giving consent and an e-mail being sent. Each of these actions occurs in a different function, and is potentially logged in different locations. Watchtower's *event ingest* function is responsible for collecting events from different logs and adding them to an indexed database, which then allows events to be easily correlated and ordered when checking properties (§3.5).

The event ingest function (which is a serverless function) is triggered when new events are written to any function's log; when triggered it reads logged events and adds them to an event database. Watchtower creates indices for this event database using the set of property parameters (`quantifiedVariables`) provided by users of Watchtower. This index allows quick retrieval of events corresponding to a property instance, thus speeding up property checking. For example, the property in Listing 1 is parameterized by user ID and e-mail subject, and Watchtower creates an index based on these parameters. Observe that some events, *e.g.,*`CONSENT`, that are used by the property state machine, do not include all fields (`email_subject` in this case). Watchtower indexes these properties using the subset of parameters available (`user_id`), and we call partial indices *projections*.

In our implementation the event ingest function writes events to DynamoDB, and later in §5 we explain how we can do so using a single table and two indices (one for the property type, and another for the parameters recorded). Furthermore, for efficiency we implement the ingest function so it reads all new log entries in a batch, and the use of batching improves resource efficiency (by decreasing the number of function invoked, etc.) and results in better performance. Our use of serverless functions ensures that event ingest can scale to keep up with the rate at which events are logged.

## 3.5 Property checking

The *property checker* is a serverless function responsible for processing ingested events stored in the database and determining whether or not a program property has been violated.

The property checker is triggered whenever a *terminal event*, i.e., an event that *can* transition a property instance to a *success* or *failure* state, is added to the database by the event ingest function. When triggered, the property checker function receives the terminal event as input, whose parameters uniquely identify a property instance (§3.1). We require that terminal events include all property parameters.

The property checker function is *provided by* the Watchtower implementation and *does not* need to be specialized to a particular application or property. During deployment Watchtower includes the property specification JSON file (§3.2) in the function's deployment package [24], and the property checker reads the specification when triggered.

Thus, when triggered, the property checker knows the property instance that it needs to check, and the state machine corresponding to this property instance. To check whether or not the property instance has been violated, the checker retrieves from the database all events belonging to this property instance, orders them by timestamp, and then transitions through the state machine in event order, and notifies the user if the state machine reaches the FAILURE state.

Structured events driving a property state machine can be produced by *different serverless functions*, each of which logs independently. Events from these different logs can be indexed by different instances of the event ingest function, and Watchtower cannot guarantee the order in which events are added to the database. As a result Watchtower cannot ensure that all events affecting a property instance are already in the database when property checking is triggered; missing events can result in false positives or negatives. We address this problem by delaying property checking for a fixed length of time to allow all events to be added to the database. As previously stated in §3.1, we assume that datacenter networks are partially synchronous, logged events are visible after a finite delay, and this *logging delay* can be determined (either empirically or from documentation). Delaying execution of the property checking algorithm by the *logging delay* guarantees that all events the occurred *before* the terminal event have been written before the checking algorithm runs.

Watchtower's correctness and performance depend on the choice of logging delay. Too short a delay can result in false positives and negatives, while too long a delay increases time before a violation is noticed. At present we use a conservative delay estimate computed using documented SLAs for the logging service and distributed database. In practice this resulted in a logging delay of 7 seconds. In the future we envision a more dynamic approach, where starting from a conservative estimate, Watchtower uses empirical measurements to progressively reduce logging delays.

Similar to the event ingest function, implementing property checking as a serverless function allows Watchtower to scale property checking in response to changing rates of event production. Additionally, several property checker functions can be safely invoked in parallel, and this ensures that Watchtower can scale to keep up with an increasing number of properties. Finally, we note that for many properties a single event can occur several times in the transition function. This in turn means that a terminal event may not necessarily result in a state machine transitioning to a terminal state. However, a property checker is invoked *whenever* a terminal event occurs. In order to reduce duplicated efforts, we save the current state of a property instance in a database whenever the property checking algorithm is run but does not reach a terminal state. Subsequent invocations of the property checking algorithm for this instance begin execution from the saved state.

**Checker run example** Consider a mailing list application, the property in Listing 1, and the following sequence of events: (a) at time t = 0, Alice, with e-mail address `a@me.com`, signs up to receive promotional e-mails, resulting in the application logging a `CONSENT` event with the e-mail address as a parameter; (b) at time t = 1, the application sends Alice an e-mail with the subject "PROMOTION: Your first coupon", which results in the application logging a `SENT_EMAIL` event with the subject and e-mail address. Each of these logs trigger the event ingest function, which adds them to the database. The `SENT_EMAIL` function is a terminal event and adding it to the database triggers the property checker function. When first triggered the property checker function waits for logging delay to elapse before running the checking algorithm. The checking algorithm creates the property instance, searches the database for a `CONSENT` event matching Alice's e-mail address, and when found checks to make sure that the consent event's timestamp predates the `SENT_EMAIL` timestamp, which causes the property instance to transition to the `consented` state. Next, the checker transitions using the `SENT_EMAIL` event and arrives at the `SUCCESS` state, showing the property is not violated.

**Dealing with clock skew** Watchtower relies on accurate timestamps to determine event order. Current serverless implementation including ones offered by Amazon, Azure and Google use NTP for time synchronization. Prior studies [65, 78] have shown that NTP within a datacenter can synchronize clocks to within a few microseconds and can provide millisecond level accuracy in the wide area. As a result Watchtower must rely on alternate mechanisms when reasoning about the order of events that occur within a small time interval.

Our current implementation assumes that ordering events using timestamps is insufficient if two events occur within 10ms of each other. In this case rather than trying to order these events, Watchtower considers all possible permutations of such events when checking properties, and decides that a property has been violated if *any* permutation would result in a violation. Watchtower uses dynamic partial order reduction (DPOR) [45] to reduce the number of permuted schedules it needs to consider. DPOR uses the observation that many events are commutative, *i.e.,* their order has no impact on the results of the property checking algorithm, to eliminate schedules which are equivalent to another schedule.Needing to consider multiple schedules also complicates the task of saving a property instance's state, because a property instance can be in multiple states simultaneously. Watchtower handles this by instead saving a *vector* of states for each property instance.

While the design above is sufficient for ensuring correctness, it potentially carries a performance penalty because of the additional schedules that must be checked. However, in practice we found that when checking properties, Watchtower generally only needs to consider a small number of schedules, often one. This is because the property checker only considers the order of events that correspond to a single property instance (§3.2), a small subset of all events produced by an application. Additionally, in our experience property instances typically relate events triggered by a few (often one) users. In this case human reaction time limits the rate at which events corresponding to a single instance occur, and thus increases the likelihood that timestamps suffice when ordering events. In our case studies (§6) we found no instances where we needed to consider permuted resources. In §7 we use synthetic microbenchmarks to evaluate the performance of our techniques for addressing clock skew.

Finally, considering *all* permutations and reporting an error when *any* of them leads to an error can result in false-positives where Watchtower will report an error in the absence of an actual violation. As a result Watchtower is *sound* (*i.e.,* all actual violations are reported), but not *complete* (*i.e.,* the system might report an error even when none occurred in reality).

## 3.6 Garbage collection

The Watchtower ingest function (§3.4) appends new events to the database table, but none of the mechanisms described so far subsequently delete these events. This can result in unbounded growth in table size. Cloud databases, including DynamoDB, charge by the volume of data contained in a table, and this growth can result in high costs. In order to address this growth, Watchtower includes a garbage collection algorithm that deletes events which are guaranteed to *never* be accessed by the property checking function in the future. These are events which do not require the use of projection, and for which a terminal event occurred sufficiently in the past (such that log delay has elapsed). We *cannot* safely collect events that require the use of projections: a single such event can correspond to an unbounded set of property instances, and we can never determine that no subsequent terminal event will correspond to such an instance. We also analyze application properties to determine cases where there is a cycle between two events (*e.g.,* a consent and withdraw consent event), in which case we must merely keep the latest event in the cycle. Garbage collection is triggered periodically, and operates asynchronously and thus does not affect detection latency. By utilizing TTL features of the cloud database, garbage collection does not incur any additional cost (§5).

## 3.7 Retroactive checking

So far we have assumed that program properties are known when an application is executed. However, the set of correctness properties a program needs to meet can change over time, with new properties added due to new regulatory or organizational changes, or the addition of new correctness requirements. Watchtower includes a retroactive checking mechanisms that can determine whether an application has *previously* violated a newly added property.

Retroactive checking in Watchtower builds on the observation that logging SDK calls (§3.1) is sufficient to capture most program events, and can only be used if a property does not rely on new events added to the application. Additionally, retroactive checking requires access to all events logged by the system. In order to cheaply enable access to historic events, Watchtower configures the log service so that in addition to triggering the event ingest function, the log service also writes logged events to Amazon S3, a cheap blob store. Retroactive checking (and debugging as we explain below) reads events logged to the blob store when required.

When a new application property is added, Watchtower runs retroactive checking by reading events from the blob store and running the ingest process on these events. This retroactive-ingest process writes to a new database, separate from the one used for runtime checking, and this new database is configured to never trigger property checking. Once retroactive-ingest is complete, Watchtower identifies all ingested terminal events that belong to the new property and for which the logging delay has expired. Watchtower then triggers property checking for these identified events, and notifies the user of any violations identified. After this property checking step has been completed, Watchtower deletes the database created by the retroactive-ingest process. The use of a different database means that application execution does not need to be paused during retroactive checking.

## 4 Debugging

Once a property violation is detected, developers need to debug the program in order to determine the cause of the violation. The cause of a violation might lie in the past (*e.g.,* in a previous user interaction), and currently this analysis necessitates manual log inspection to identify an hypothesis for what might have caused a violation. Confirming or rejecting these hypotheses is equally challenging: existing tools neither provide mechanisms for developers to recreate previous execution points in serverless applications, nor do they provide mechanisms to insert breakpoints or inspect the state of an application during execution. The absence of such mechanism makes it challenging to address violations.

Watchtower includes a record-and-replay debugger [12, 57] to simplify this task. This debugger can recreate application state and re-execute any function previously executed by the serverless application on a developer's local machine. Developers can then use existing tools, *e.g.,* Node Inspector [77], to inspect its state, step through its execution, etc.

To implement record-and-replay debugging, Watchtower logs parameters and return values for all non-deterministic calls including calls to external services, calls that read time, and ones that generate random numbers. We observe that Watchtower's SDK logging (§3.3) already records calls, arguments and return values for all interactions with external services, covering a large fraction of non-deterministic calls. Watchtower uses the same proxy mechanism to log structured events for the remaining non-deterministic function calls. In contrast to works like ODR [12], Watchtower *does not* need to account for non-dereminism resulting from scheduling and data-races due to preemptive multi-threading thanks to our focus on JavaScript application which are inherently single threaded.[4] On the other hand, asynchronous calls (through callbacks or promises) are widely used in JavaScript applications, and Watchtower does need to record the order in which asynchronous calls return. In order to do so Watchtower also

[4]However, we believe that this is not a fundamental limitation. Previous works, such as RR [79], have shown how record-and-replay debugging can be applied to multithreaded programs.

emits a structured event every time an asynchronous call returns, and this allows the debugger to determine the order in which callbacks should be invoked during replay. As we previously described in §3.7, Watchtower configures the logging service to write all structured event to a durable blob store.

During replay, Watchtower reads event logs for the functions the developer wants to reproduce from the blob store. The replay debugger then creates a process for each function that the developer wants to inspect. The replay mechanism uses VM2 (also used in §3.3) to replace SDK calls and non-deterministic calls with proxies that replay recorded values.

In some cases, the root-cause of a violation might lie in a function executed in the past. In these cases developers need to identify the causal chain that lead to the violation. We observe that functions in a serverless application communicate using external services such as S3 or SQS (§2.1). Watchtower's SDK instrumentation records all accesses to these services, and hence Watchtower can reconstruct causal links between functions using this information, allowing developers to follow the causal chain to the root cause of the violation.

The Watchtower debugger thus reduces developer effort when debugging serverless applications, and allows developers to both identify and address the cause of application violations. Since the debugger largely reuses the same instrumentation used by Watchtower for monitoring, the record-and-replay debugging capabilities of Watchtower add *negligible additional* overheads on top of those already used to store the history used for allowing retroactive checking (§3.7). Additionally, the debugger can be used to investigate application problems beyond program property violations, and Watchtower's debugging capabilities are useful independent of its monitoring capabilities.

## 5 Implementation

We have implemented Watchtower in JavaScript (ES6). The development of Watchtower required roughly one person-year of effort. In addition to implementing the core components of Watchtower, we have also instrumented several libraries and built-in Node.js modules used by the applications we evaluate (§6) including: `aws-sdk` APIs for S3, DynamoDB, and Kinesis; `node-fetch` for HTTP access; `twit` for interacting with twitter; `sendgrid` for sending emails; and `fs` for file system access. We did not need to make any modification to the application code in order to facilitate instrumentation.

Our implementation supports two logging back-ends: AWS Kinesis and AWS CloudWatch. Both services can be configured to trigger a Lambda function when new log events arrive; we use this mechanism to trigger the ingestion function (§3.4) and feed the event logs into it.

The ingestion function of Watchtower stores events in a DynamoDB table. DynamoDB compound keys consist of two fields, a *partition key* and a *sort key*. DynamoDB uses the partition key to determine which partition the data is stored on, and `query` calls on DynamoDB tables rely on partitions to efficiently return sets of data. Watchtower constructs an *instance string* for each property instance by concatenating the property name, and the sequence of parameter names and values. This guarantees that each property instance has a unique string, and this string is then used as the partition key. The keys of projection events are created using the properties that are available in the projection. The sort key of each event is a unique identifier, guaranteeing an overall unique compound key per stored event.

The checker function (§3.5) uses the same process to construct event keys when querying DynamoDB. The checker then makes a `query` call to the events table per projection, reading all of the events of the property. The partition locality of data improves the efficiency of the querying process.

When the checker function has processed all recorded events for a property instance (which might not be sufficient to decide whether a violation occurred or not) it *checkpoints* the state of the instance in a DynamoDB table. The checker begins subsequent checks for this property instance by reading the checkpoint, minimizing queries to the events table.

When a property instance reaches a terminal event, or when the instance is checkpointed, instance events can be safely deleted. The Watchtower garbage collection mechanism utilizes DynamoDB's TTL (Time to Live) feature to mark events for deletion by adding an expiration field set to the current time. DynamoDB periodically runs a background process that deletes expired items in the table.

Our implementation targets Node.js-based applications running on AWS Lambda. However, these limitations are not fundamental to the Watchtower architecture, and Watchtower can be modified to run on a variety of cloud platforms. Watchtower relies on the following AWS services: Kinesis for recording events and instance checker notifications, DynamoDB and S3 for event storage, AWS Lambda for running the ingestion and checker components, StepFunctions for orchestrating the delayed checker execution flow, and SES for notifying the user of violations. Other providers offer similar services with nearly identical semantics and APIs. Porting Watchtower to run on a different cloud platform would consist mainly of changing cloud service API calls.

Watchtower requires that the monitored application produce structured log events (§3.1). We achieve this by instrumenting the application (§3.3), using the `vm2` sandbox and ES6 proxies to inject event emitting code into the application without modifying any of its code. However, the production of structured logged events used by Watchtower does not necessitate the use of either serverless applications or JavaScript. Similar black-box recording behavior can be achieved by using AOP tools such as AspectJ [64] for JVM-based languages, as well as other metaprogramming

**Table 1:** *Serverless applications used to evaluate Watchtower.*

| Application | Description |
|---|---|
| RealWorld[8] | a blogging platform similar to Medium |
| Bitwarden[30] | a cloud-hosted password manager similar to LastPass |
| pingbot[7] | a bot that monitors the health of website |
| YoYo[10] | an embeddable comment engine similar to disqus |
| Nietzsche[5] | a web scraper and twitter bot for literary quotes |
| Serverless Video Analysis[11] | A web service for processing and analyzing videos |

mechanisms available in various programming languages. Finally, log events can even be produced explicitly by manually annotating the code, extending the applicability of our approach to any programming language and runtime.

Watchtower's source code is available at https://github.com/kalevalp/watchtower.

## 6 Case study

We evaluate Watchtower's generality by using it to monitor the correctness of six open-source serverless applications (Table 1). In this section we provide a detailed description of the most complex one: RealWorld. We detail the correctness properties we monitored for this application, and performance measurements from the deployment of Watchtower alongside RealWorld. A detailed evaluation of Watchtower using synthetic microbenchmarks is discussed in §7.

RealWorld [8] is a family of interoperable full-stack implementations of a blogging platform (a Medium.com clone app) written using different programming languages and technology stacks. RealWorld DynamoDB Lambda [3] is a cloud native implementation of the RealWorld back-end, using AWS Lambda for compute and AWS DynamoDB for storage. We extended the application with lambda functions that grant and revoke user data processing consent.

### 6.1 Correctness properties for RealWorld

We ran the RealWorld application with 10 correctness properties. One of the correctness properties expresses compliance with GDPR article 7. The other 9 properties were derived by generalizing existing system tests.

**GDPR Article 7 compliance** Article 7 of the GDPR requires that "[w]here processing is based on consent, the controller shall be able to demonstrate that the data subject has consented to processing of his or her personal data" [1]. In order to monitor compliance with this requirement, we defined a correctness property that is violated whenever data is processed without consent (either because none was given, or because it was revoked). The application emits `CONSENTED`, and `REVOKED_CONSENT` events when the functions for granting and revoking consent are called, and `PROCESSING_DATA` events when accessing user data.

**Correctness properties derived from tests** The RealWorld application contains 57 integration tests. These tests serve as a description of what the correct behavior of the system is, as software developers are expected to cover all of the important and probable system flows with dedicated tests. We used these system tests as a guide for specifying correctness properties. We defined 9 correctness properties, based on 31 of the 57 tests. The tests we chose describe system behaviors across several different serverless functions, and involving non-trivial causality. The resulting properties are global, application-wide, distributed properties. They cannot be enforced by assertions in the code, and require a distributed monitoring system such as Watchtower. The remaining 26 tests described local function behavior (e.g., a test that publishes an article, and checks that the operation was successful). While such properties can be monitored by Watchtower, we chose to focus on properties that require a distributed monitoring system, and cannot be addressed by simpler mechanisms.

**Table 2:** *Correctness properties in the RealWorld case study.*

| Property | Correctness Criterion |
|---|---|
| Article Created | login by author |
| Article Retrieved | article created |
| Article Deleted | article created, login by author |
| Article Fave | article created, login by reader |
| Article Listed | article created, login by reader |
| Article in Feed | article created, reader follows author |
| Comment Created | article created, login by reader |
| Comment Deleted | comment created, login by comment author |
| Comment Retrieved | comment created |

Table 2 lists the correctness properties and gives a brief summary of the correctness criterion of each property. All properties require that several events, emitted by different parts of the application, occur in a specific order. For example, the 'Comment Deleted' property (row 8) requires that if some user deletes a comment from an article, then the comment had previously been created by the user on the article, the comment has not been deleted since, the article has not been deleted since the comment was created, the user had previously logged in, and has not logged out since then. Each of these events is emitted by a different Lambda function.

### 6.2 Performance when monitoring RealWorld

To evaluate Watchtower's monitoring performance, we ran a test scenario, invoking all components of the RealWorld back-end. The scenario consisted of 130 application requests. We repeated the scenario 25 times. We measured end-to-end execution times of each request, running with and without Watchtower. The mean request running time without Watchtower recording was 76.93ms ($\sigma = 83.89$), whereas with Watchtower recording the mean increased by 25.29ms to 102.22ms ($\sigma = 122.64$), an increase of 32%, representing an additional \$0.000000422 in compute expenses[5]. The 99th

[5]All reported price figures assume running 1024MB lambda instances in the AWS region us-east-1. Price information is up-to-date as of September 2021.

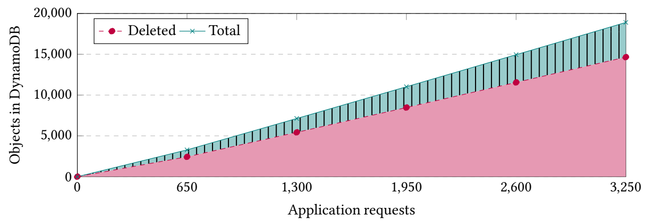

**Figure 2:** *Objects stored in the events table on DynamoDB.*

**Table 3:** *Serverless applications used in microbenchmarks.*

| Application | Description |
|---|---|
| *simple* | emit a single event |
| *event+sleep* | emit a single event, then sleep for several milliseconds |
| *sleep+event* | sleep for several milliseconds, then emit a single event |
| *loop* | emit a sequence of events, separated by periods of sleep |

percentile running time without Watchtower recording is 292.53ms, whereas with Watchtower recording the 99th percentile is 547.02ms, an increase of 254.49ms, or 87%.

Additionally, we measured the running times of the ingestion and the checker functions. The average amount of ingest Lambda compute time performed per application request was 53.45ms, and the average amount of checker Lambda compute time was 70.99ms per request, adding an additional 124.44ms of Lambda compute resources per request performed. This represents an added expense of $0.000002478 per application request. Recall that the ingestion and checker functions run out-of-band, separately from the monitored application, and these running times have no effect on the application execution.

In order to evaluate the effectiveness of our garbage collection optimization (§3.6) we recorded the number of event objects stored in the DynamoDB events table during the run of the scenario. Figure 2 shows the number of stored events as a function of the number of application requests. The solid line represents the total number of event objects written to DynamoDB. The dashed line represents the number of event objects deleted by the garbage collection optimization. The area between the two lines (marked by vertical lines) represents event objects written to DynamoDB and not deleted by the garbage collection optimization. Objects not deleted by the garbage collection are either projection events that cannot be safely deleted, or full-instance events whose property instances have not reached a final state yet.

## 7 Detailed Evaluation

In this section we present our evaluation of Watchtower performance. We designed several simple synthetic applications, listed in Table 3, to perform a detailed evaluation of the performance aspects of Watchtower. Applications were executed on AWS Lambda; we used a `t3a.medium` EC2 instance, deployed to the same region as the applications, to launch applications and record measurements.

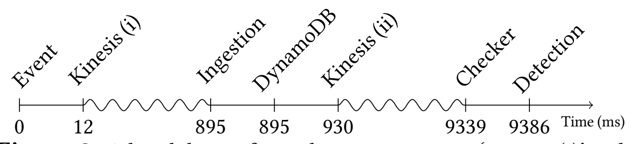

**Figure 3:** *A breakdown of Watchtower execution. 'Kinesis (i)' is the Kinesis arrival time of the structured log message; 'Ingestion' is the start time of the ingestion function; 'DynamoDB' is the time the processed event was written to DynamoDB; 'Kinesis (ii)' is the Kinesis arrival time of the property instance checker notification; 'Checker' is the start time of the checker function; and 'Detection' is the violation detection time.*

**Table 4:** *Overheads of structured logging.*

| | Baseline | | | CloudWatch | | | Kinesis | | |
|---|---|---|---|---|---|---|---|---|---|
| Application | Mean | $\sigma$ | $\Delta$ | Mean | $\sigma$ | $\Delta$ | Mean | $\sigma$ | $\Delta$ |
| Sleep+event | 101.77 | 0.77 | 0 | 102.47 | 1.23 | 0.69 | 124.05 | 10.57 | 22.27 |
| Event+sleep | 101.77 | 1.02 | 0 | 102.34 | 1.06 | 0.56 | 104.45 | 5.92 | 2.67 |

### 7.1 Performance

Figure 3 shows the average time spent (in milliseconds) in each stage of the Watchtower pipeline when detecting a violation. We evaluate this using the *loop* application. We logged events to Kinesis and used 7 seconds as our logging delay (§3.5). The total time required to detect and notify a user of a violation (the *detection latency*) was 9386ms. We observe that a majority (99%) of the time is spent before the Checker stage while waiting for Kinesis to invoke the ingest and checker functions, and for the logging delay to expire. Below we evaluate the runtime overheads of event logging, and evaluate the trade-off between application overhead and detection latency.

**Overhead of structured logging** As noted in §3.3, structured event logging is performed by application code, and hence overheads in logging can affect the application's fast path. In order to evaluate this impact we measured logging overhead when using AWS CloudWatch and AWS Kinesis. CloudWatch batches log writes, thus amortizing the cost of logging. This amortization however comes at the cost of increased delays before a property can be checked. Kinesis, by contrast, contacts the log service immediately, thus increasing logging overhead but allowing faster property checking. Our implementation supports both, and in both cases we use non-blocking asynchronous calls for logging in order to minimize application overheads.

We measure overheads using the *sleep+event*, and *event+sleep* applications. In the second case our use of asynchronous calls allows us to overlap the time spent logging with application logic. This is not the case for the first benchmark, where we expect that logging overhead will impact the function execution duration. In Table 4 we show latency (measured as function execution time) for these applications without structured logging (baseline), when logs are written to CloudWatch and when logs are written

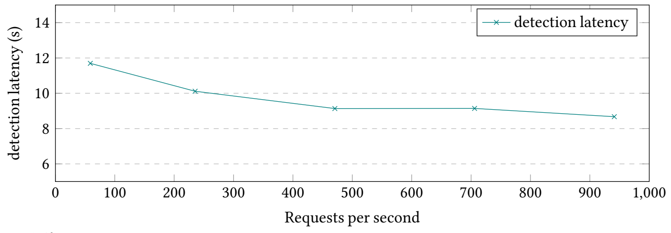

**Figure 4:** *Detection latency as a function of application load.*

to Kinesis. As expected, CloudWatch logging adds minimal overhead (1-2ms), whereas Kinesis logging adds a more significant overhead of ~20ms. In both cases the asynchronous calls overlap with the *event+sleep* execution, and have a negligible impact on the execution time of that application.

**Logging latency** Next, we measure the delay between an application emitting a structured log event and the Watchtower ingest function being triggered. We do so using the *simple* application. We measured logging latencies using both AWS Kinesis and Cloudwatch. When using Kinesis, the logging latency is 944ms on average, while for Cloudwatch it was 5194ms on average. However, as we previously observed, logging to Kinesis imposes a larger overhead on application performance, and developers must trade-off application overheads and detection latency in choosing between these options.

## 7.2 Scalability

In this section we investigate the scaling behavior of Watchtower as we vary (a) application load; (b) the complexity of the correctness properties monitored; and (c) the number of correctness properties monitored.

**Scaling with Workload** Next, we look at Watchtower's behavior as a function of workload changes. Recall that we implement ingest and property checking using serverless functions in order to ensure that Watchtower's performance does not degrade with changes in workload. In order to observe scaling behavior, we use the *loop* application, and scale the application call request rate from 60 invocations/s to 950 invocations/s[6]. In Fig. 4 we measure the detection latency as a function of the request rate. We find that detection time does not increase with increased request rates, thus demonstrating that Watchtower can scale to match the application's scaling behavior.

**Property Complexity** In Watchtower the size of a property (*i.e.,* number of states) closely corresponds to its complexity. Therefore we evaluate Watchtower's effectiveness in analyzing complex properties by varying property size and measuring run times of the checker and ingest functions. Since both functions perform local operations on the property state machine, we do not expect the size of the property to impact the running times. Indeed, as Fig. 5 shows, the size of the property does not affect execution times.

[6]Our experiments were restricted by the AWS Lambda global limit of 1000 concurrent function invocations.

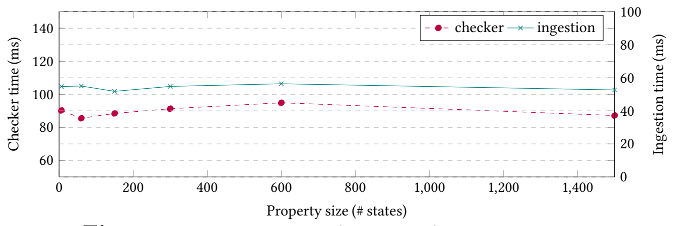

**Figure 5:** *Run times as a function of property size.*

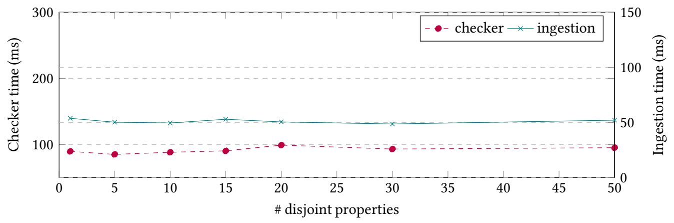

**Figure 6:** *Run times per number of disjoint properties.*

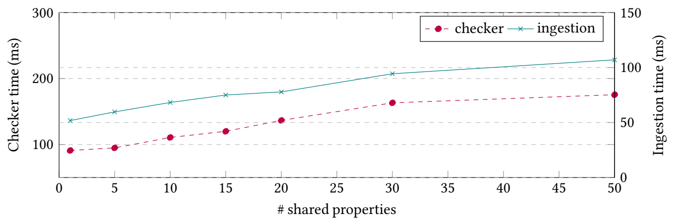

**Figure 7:** *Run times per number of shared properties.*

**Number of properties** We measure the impact of the number of properties being monitored on Watchtower performance in two different cases—(a) the properties share events, and (b) the properties have disjoint sets of events. When properties share events, we expect the checker and ingestion to execute longer as we add more properties. This is because events may lead to indexing and checking multiple different properties, causing increased workload per event. Indeed, Fig. 7 shows that the execution times increase as we add properties that share events. However, we are encouraged by the fact that the growth is mild—having increased the number of properties fifty-fold, we see a twofold increase in the average running time, from 91ms to 175ms in the checker, and from 52ms to 107ms in the ingestion function. When properties do not share events, we do not expect an increase in the running times of the checker and the ingestion, and indeed Fig. 6 illustrates that.

## 7.3 Effectiveness of the DPOR Optimization

We investigate the effectiveness of our reordering mechanism and dynamic partial order reduction optimization (§3.5) in handling events that occur within the clock-skew window. We run a variant of the *simple* application, in which all events belong to the same property instance, guaranteeing that if

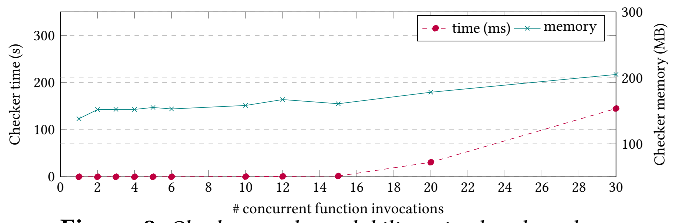

**Figure 8:** *Checker reorder scalability microbenchmark.*

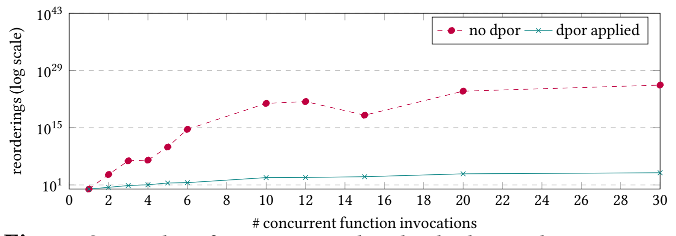

**Figure 9:** *Number of permutations that the checker needs to process with and without the DPOR optimization.*

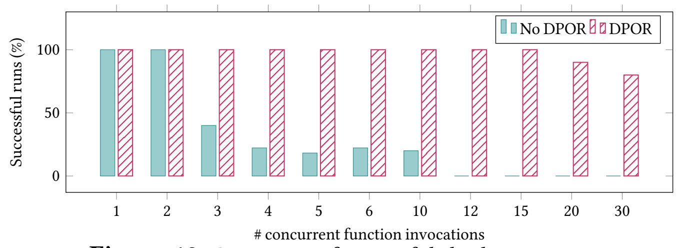

**Figure 10:** *Percentage of successful checker runs.*

two event timestamps are within 10ms of each other, the checker will have to consider both orders. We increase the rate of concurrency from 1 (no concurrency) to 30 concurrent function invocations. We ensure that the overall number of events remains the same for every rate of concurrency, so that the checker always processes a log of the same size.

Fig. 8 shows the running times (in ms) and memory usage of the checker as a function of the concurrency of the function. As expected, we can see an increase in both time and memory usage as the rate of concurrency increases. Fig. 9 shows the number of different permutations that arise as a result of the increase in contention (dashed line), compared to the number of permutations traversed in practice as a result of the DPOR optimization. Fig. 10 shows the percentage of checker runs that ended successfully (*i.e.*, did not fail due to a timeout or an out of memory error).

## 8 Related work

In §2.3 we discussed the related literature on distributed systems monitoring. Here we briefly discuss the related work on runtime-verification and record and replay debugging.

**Runtime verification of distributed systems** Runtime verification (RV) tools detect violations of correctness properties during the execution of the application. RV research has covered various aspects of verification systems, property definition languages, and monitored applications [39, 42]. Runtime verification of distributed system [31, 35, 36, 40, 55, 74, 75, 89, 95] can be classified as either distributed [26, 28, 29, 49, 50, 58, 60] or centralized [27, 28] monitoring systems.

In distributed monitoring systems, every executing process has a local monitor and the local states of processes are sent to all other processes. In centralized monitoring systems, every process in the system sends its local state to a dedicated monitor process. Watchtower takes a hybrid approach of a logically centralized monitor, i.e., all processes in our system send their local state to a central service, but running it in a physically distributed way, taking advantage of the natural parallelization inherent to monitoring parameterized properties.

**Properties defined as quantified automata** Reger et al. [83] had previously used quantified state machine to define properties for monitoring Java applications, similar to the parameterized automata in Watchtower. When events in the system are executed with specific values for the transition parameters, they instantiate a quantifier-free automaton, and check that the execution does not violate it.

**Record and replay debugging** The area of record and replay debugging had been extensively studied, with works focusing on different runtime environments [59, 69, 79, 88], and language agnostic recording [13, 32, 85, 86]. The single-threaded nature of the JavaScript runtime removes the need to record scheduling information, making the recording process a natural extension of the recording mechanism used to record application events. This is similar to the approach taken by the initial versions of Jalangi [88].

## 9 Conclusion

Detecting and debugging correctness violations in serverless applications is an essential step for wider adoption of this paradigm. In this paper we described Watchtower, which to the best of our knowledge, is the first system to address this growing requirement. Using Watchtower requires little or no modification to the application, and the only effort required is for expressing correctness properties as finite automata. Watchtower imposes minimal overheads on the application, and its detection mechanisms can elastically scale with the application. As a result Watchtower provides a practical path to detecting bugs in running serverless applications.

## Acknowledgments

We thank the anonymous referees for their insightful comments. This research was partially supported by the Israel Science Foundation (ISF) grant No. 1810/18, and by Len Blavatnik and the Blavatnik Family foundation.

## References

[1] 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation). *OJ* L 119 (2016), 1–88.

[2] 2019. *Building a truly global e-commerce platform — part 2: architecture & technology.* https://blog.labdigital.nl/building-a-truly-global-e-commerce-platform-part-2-architecture-technology-4a3f1afd5616.

[3] 2019. *λ serverless backend implementation for RealWorld using AWS DynamoDB + Lambda.* https://github.com/anishkny/realworld-dynamodb-lambda.

[4] 2019. *MoonMail - Email marketing platform for bulk emailing via Amazon SES.* https://moonmail.io/.

[5] 2019. *Nietzsche - Scrap quotes from Goodreads and schedule random tweets.* https://github.com/rpidanny/Nietzsche.

[6] 2019. *Noiiice - a serverless blog built on NuxtJS, AWS, serverless framework, and irrational exuberance.* https://github.com/DylanAllen/noiiice.

[7] 2019. *pingbot - A website monitoring/health-checking tool based on serverless architecture.* https://github.com/toricls/pingbot.

[8] 2019. *RealWorld.* https://github.com/gothinkster/realworld.

[9] 2019. *Why we use serverless architecture at Freetrade.* https://blog.freetrade.io/why-we-use-serverless-architecture-at-freetrade-e668c7bf5d42.

[10] 2019. *YoYo - A dead simple comment engine built on top of AWS lambda and React, alternative comment service to Disqus.* https://github.com/metrue/YoYo.

[11] 2020. *Serverless Video Preview and Analysis Service.* https://github.com/laardee/video-preview-and-analysis-service.

[12] Gautam Altekar and Ion Stoica. 2009. ODR: output-deterministic replay for multicore debugging. In *SOSP*.

[13] Silviu Andrica and George Candea. 2011. WaRR: A tool for high-fidelity web application record and replay. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN).* IEEE, 403–410.

[14] Apache. 2021. *OpenWhisk.* https://github.com/apache/openwhisk.

[15] Anish Arora, Sandeep Kulkarni, and Murat Demirbas. 2000. Resettable vector clocks. In *PODC*.

[16] Mona Attariyan, MIchael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *OSDI*.

[17] AWS. 2019. *Amazon CloudWatch.* https://aws.amazon.com/cloudwatch/.

[18] AWS. 2019. *Amazon Kinesis.* https://aws.amazon.com/kinesis/.

[19] AWS. 2020. *Amazon API Gateway.* https://aws.amazon.com/api-gateway/.

[20] AWS. 2020. *Amazon Aurora.* https://aws.amazon.com/rds/aurora/.

[21] AWS. 2020. *Amazon DynamoDB.* https://aws.amazon.com/dynamodb/.

[22] AWS. 2020. *Amazon S3.* https://aws.amazon.com/s3/.

[23] AWS. 2020. *AWS Lambda.* https://aws.amazon.com/lambda/.

[24] AWS lambda dep 2020. AWS Lambda deployment package in Node.js. https://docs.aws.amazon.com/lambda/latest/dg/nodejs-package.html.

[25] AWS SDK Release 2020. aws-sdk-js: History for CHANGELOG.md. https://github.com/aws/aws-sdk-js/commits/master/CHANGELOG.md.

[26] Özalp Babaoğlu, Eddy Fromentin, and Michel Raynal. 1995. Debugging Distributed Executions by Using Language Recognition. In *ICPP (2).* 55–62.

[27] Özalp Babaoğlu, Eddy Fromentin, and Michel Raynal. 1996. A unified framework for the specification and run-time detection of dynamic properties in distributed computations. *Journal of Systems and Software* 33, 3 (1996), 287–298.

[28] Özalp Babaoğlu and Keith Marzullo. 1993. Consistent global states of distributed systems: Fundamental concepts and mechanisms. *Distributed Systems* 53 (1993).

[29] Özalp Babaoğlu and Michel Raynal. 1995. Specification and verification of dynamic properties in distributed computations. *J. Parallel and Distrib. Comput.* 28, 2 (1995), 173–185.

[30] Bitwarden 2019. *Bitwarden.* https://bitwarden.com/.

[31] Bernd Bruegge, Tim Gottschalk, and Bin Luo. 1993. A framework for dynamic program analyzers. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications.* 65–82.

[32] Brian Burg, Richard Bailey, Andrew J Ko, and Michael D Ernst. 2013. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology.* ACM, 473–484.

[33] David Calavera and Lorenzo Fontana. 2019. *Linux Observability with BPF: Advanced Programming for Performance Analysis and Networking.* O'Reilly Media.

[34] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[35] Himanshu Chauhan, Vijay K Garg, Aravind Natarajan, and Neeraj Mittal. 2013. A distributed abstraction algorithm for online predicate detection. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems.* IEEE, 101–110.

[36] Sarah E Chodrow and Mohamed G Gouda. 1995. Implementation of the sentry system. *Software: Practice and Experience* 25, 4 (1995), 373–387.

[37] Cloudflare. 2020. *Cloudflare Workers.* https://workers.cloudflare.com/.

[38] Jeremy Daly. 2020. *Serverless Community Survey 2020.* https://github.com/jeremydaly/serverless-community-survey-2020.

[39] Nelly Delgado, Ann Q Gates, and Steve Roach. 2004. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on software Engineering* 30, 12 (2004), 859–872.

[40] Christian Drabek and Gereon Weiss. 2017. DANA-Description and Analysis of Networked Applications.. In *RV-CuBES.* 71–80.

[41] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[42] Yliès Falcone, Srđan Krstić, Giles Reger, and Dmitriy Traytel. 2018. A taxonomy for classifying runtime verification tools. In *International Conference on Runtime Verification.* Springer, 241–262.

[43] Fastly. 2020. *Fastly Compute@Edge.* https://www.fastly.com/products/edge-compute/serverless.

[44] Colin J Fidge. 1988. Partial orders for parallel debugging. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed debugging.* 183–194.

[45] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. *ACM Sigplan Notices* 40, 1 (2005), 110–121.

[46] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. 2007. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation.* USENIX Association, 20–20.

[47] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19).* 475–488.

[48] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th*

*USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 363–376.

[49] Eddy Fromentin, Michel Raynal, Vijay K Garg, and Alex Tomlinson. 1994. On the fly testing of regular patterns in distributed computations. In *1994 Internatonal Conference on Parallel Processing Vol. 2*, Vol. 2. IEEE, 73–76.

[50] Ornan Gerstel, Shmuel Zaks, Michel Hurfin, Noël Plouzeau, and Michel Raynal. 1994. On-the-fly replay: a practical paradigm and its implementation for distributed debugging. In *Proceedings of 1994 6th IEEE Symposium on Parallel and Distributed Processing*. IEEE, 266–272.

[51] Google. 2020. *Cloud Functions*. https://cloud.google.com/functions.

[52] Google. 2020. *Cloud Spanner*. https://cloud.google.com/spanner.

[53] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. 2018. Inferring and asserting distributed system invariants. In *ICSE*. 1149–1159.

[54] Brendan Gregg and Jim Mauro. 2011. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional.

[55] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, Jeffrey Vetter, and Nirupama Mallavarupu. 1995. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Proceedings Frontiers' 95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*. IEEE, 422–429.

[56] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The essence of JavaScript. In *ECOOP*. 126–150.

[57] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay. In *OSDI*.

[58] Michel Hurfin, Masaaki Mizuno, Michel Raynal, and Mukesh Singhal. 1998. Efficient distributed detection of conjunctions of local predicates. *IEEE Transactions on Software Engineering* 24, 8 (1998), 664–677.

[59] Michel Hurfin, Noël Plouzeau, and Michel Raynal. 1993. Debugging tool for distributed Estelle programs. *computer communications* 16, 5 (1993), 328–333.

[60] Michel Hurfin, Noël Plouzeau, and Michel Raynal. 1993. Detecting atomic sequences of predicates in distributed computations. In *Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging*. 32–42.

[61] Ecma International. 2015. *ECMAScript 2015 Language Specification* (6th ed.). Geneva. http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf.

[62] 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, 445–451.

[63] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An end-to-end performance tracing and analysis system. In *SOSP*.

[64] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. 2001. An overview of AspectJ. In *European Conference on Object-Oriented Programming*. Springer, 327–354.

[65] Ki-Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. 2016. Globally Synchronized Time via Datacenter Networks. *Proceedings of the 2016 ACM SIGCOMM Conference* (2016).

[66] California State Legislature. 2019. *California Consumer Privacy Act of 2018 (Assembly Bill No. 375)*. https://leginfo.legislature.ca.gov/faces/billTextClient.xhtml?bill_id=201720180AB375.

[67] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. D3S: Debugging Deployed Distributed Systems. In *NSDI*.

[68] Friedemann Mattern. 1988. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*.

[69] James W Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for JavaScript Applications.. In *NSDI*, Vol. 10. 159–174.

[70] Microsoft. 2020. *Azure API Management*. https://azure.microsoft.com/en-us/services/api-management/.

[71] Microsoft. 2020. *Azure Blob Storage*. https://azure.microsoft.com/en-us/services/storage/blobs/.

[72] Microsoft. 2020. *Azure Functions*. https://azure.microsoft.com/en-us/services/functions/.

[73] Microsoft Azure. 2020. Azure Functions scale and hosting. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale.

[74] Aloysius K Mok and Guangtian Liu. 1997. Efficient Run-Time Monitoring of Timing Constraints.. In *IEEE Real Time Technology and Applications Symposium*. 252–262.

[75] Menna Mostafa and Borzoo Bonakdarpour. 2015. Decentralized runtime verification of LTL specifications in distributed systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 494–503.

[76] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *NSDI*.

[77] Node.js. 2019. *Node.js v12.13.0 Documentation — Debugger*. https://nodejs.org/docs/latest-v12.x/api/debugger.html.

[78] Oleg Obleukhov. 2020. Building a more accurate time service at Facebook scale. https://engineering.fb.com/production-engineering/ntp-service/.

[79] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 377–389.

[80] OpenTracing. 2020. . https://opentracing.io/.

[81] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. 2020. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media.

[82] Lin Quan, John Heidemann, and Yuri Pradkin. 2014. When the Internet sleeps: Correlating diurnal networks with external factors. In *IMC*.

[83] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. 2015. MarQ: monitoring at runtime with QEA. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 596–610.

[84] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. 2006. Pip: Detecting the Unexpected in Distributed Systems.. In *NSDI*, Vol. 6. 9–9.

[85] Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS)* 17, 2 (1999), 133–152.

[86] Yasushi Saito. 2005. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*. ACM, 69–76.

[87] Brandon Schlinker, Italo Cunha, Yi-Ching Chiu, Srikanth Sundaresan, and Ethan Katz-Bassett. 2019. Internet Performance from Facebook's Edge. In *IMC*.

[88] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 488–498.

[89] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. 2004. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 418–427.

[90] 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 *ATC*.

[91] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).

[92] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[93] Patrik Simek. 2019. *vm2*. https://github.com/patriksimek/vm2.

[94] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J Franklin, Michael I Jordan, and David A Patterson. 2011. The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements.. In *FAST*, Vol. 11. 163–176.

[95] Jeffrey J. P. Tsai, K-Y Fang, H-Y Chen, and Y-D Bi. 1990. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Transactions on Software Engineering* 16, 8 (1990), 897–916.

[96] Zipkin. 2019. . https://zipkin.io/.