Research Internship Report at Ecole Polytechnique

Audit Trailing Activities in a Resource Automation Service

Gasparini Fiuza do Nascimento, Henrique
Professor: Gilles Schaeffer
Supervisor: Angela Visser

February 7, 2018

Déclaration d'intégrité relative au plagiat

Je soussigné(e) GASPARINI FIUZA DO NASCIMENTO, Henrique certifie sur l'honneur:

1. Que les résultats décrits dans ce rapport sont l'aboutissement de mon travail.
2. Que je suis l'auteur de ce rapport.
3. Que je n'ai pas utilisé des sources ou résultats tiers sans clairement les citer et les réferencer selon es règles bibliographiques préconisées.

Mention à recopier

Je déclare que ce travail ne peut être suspecté de plagiat.

23 Août, 2017, Henrique Gasparini Fiuza do Nascimento

Abstract

When developing any large-scale serious project, one indispensable feature is a mechanism to track activities history efficiently, reliably, and securely, in order to provide useful insights to people using it.

In my internship, I joined one of Amazon EC2's resource automation teams, which also needed this feature. In addition, given its cell-based architecture, it proved to be an interesting case for tracking activities that occurred in specific elements of its structures. When implementing this feature, it was important to notice that operations performed on a specific object contained its identifier in its parameters or its output. So activities request and response objects actually were very meaningful.

Common approaches for audit trailing activities on a service vary a lot in their natures: private-owned and open-source logging analytics platforms, AWS dedicated services, simple storage services, and cloud database services are some of them. However, each of the approaches studied in this internship could be split in three main components: a log collector and shipper, a log indexer, and a log searcher and analyzer, implemented in numerous ways.

We wanted a log shipper that provided real-time logging and strong consistency, a log indexer that allowed multiple queries indexed by different kinds of objects, and a searching tool that helped understanding the activities history. A synchronous invocation interceptor, a combination of Amazon DynamoDB non-relational cloud database service and Amazon S3, and an customer API powered by an interactive web page proved to satisfy all the wanted requirements, both in theory and in practice.

We believe that this framework can be extended to similar use cases in which one needs to track activities by objects that can be identified as parameters in the arguments or outputs from their activities.

Therefore the main research topic explored in this internship was efficiently auditing activities in a service and using non-relational databases to store and query past activities data. This report presents the collected knowledge of the state-of-the-art for Audit Trailing Activities and exposes the software engineering experience with the implementation of this feature surfaced by an API and a friendly customer-facing web page.

Contents

1 Introduction

This internship took place under Amazon Web Services EC2 division, in one of the teams responsible for internal resources automation.

Quite recently released, the service owned by this team still needed to gain more customer trust and lacked an easy way to access past activities details. Simple problems caused by unexpected operations could only be solved by reaching to our team and spending valuable engineering time.

This need eventually led to my internship project: creating an audit trail of activities taken on it, by designing its main components: the data persistence, collection, and access methods, after studying the many approaches known in the academic and industry environments.

In this report, I will attempt to organize my experiences respecting the following road-map: a background overview of the areas and technologies covered by the project, a state-of-the-art of the audit trail framework, and finally the design, architecture, implementation, and results of the internship project under the section The Implemented Audit Trail Framework.

2 Background and Feature Overview

2.1 Amazon architecture

Amazon architecture is structured by services, separately owned by different teams and evolved independently. This, along with the ability to deploy website contents independently is key to have an efficient and scalable website.

The AWS platform provides to Amazon developers (as well as to many customers outside Amazon) a flexible, scalable, cost-effective and secure platform to provision resources on-demand, in real-time and with minimum effort to address infrastructure needs.

With the same spirit of reducing operational effort to address infrastructure needs but in a smaller scale, Amazon has teams dedicated to automate internal infrastructure provisioning.

2.1.1 Resources automation and operations

Automating resources allocation for development teams is considered an important task at Amazon and is the work of many teams.

In a few words, our service allowed users not only to directly manage their resources through an API but also to specify configurations and states that should be ensured. This was done through a difference engine that periodically checked these states and performed any necessary operations.

Therefore, the audit trail should track operations performed by both users and the automation engine.

2.2 Audit trailing

2.2.1 Feature overview

Before investigating possible approaches, it was important to understand what was expected to be delivered in this feature.

The requested feature was a customer-facing audit trail of activity history on their services. This feature should bring access to activities taken both by users and by the automated engine.

We believed that this would increase consumer trust, since automated operations would then be visible and more easily understood.

2.2.2 Requirements

The audit trail feature was expected to be scalable, efficient and valuable to our users.

The amount of data that is written to the audit trail tables everyday is proportional to the number of services using our service. Since our service still is on its early age, the audit trail resources should be able to scale up to at least 10 times in the next years.

In addition to that, we wanted to allow efficient queries for activities history occurred in specific components of our service, which we believed to be of great value to our users.

2.3 Amazon DynamoDB architecture

One of the most important technologies used when implementing the audit trail was Amazon DynamoDB[1].

DynamoDB is a non-relational key-value store database service that provides fast and predictable performance with seamless scalability. It already was the database of our choice for most features in our service and was used it in this project after a thorough investigation.

Unlike most popular databases, DynamoDB is a fully managed service instead of only a program, so its users do not have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. Scaling up or down DynamoDB tables's throughput capacity can be easily and instantaneously done using the AWS console or calling the DynamoDB API with no extra work.

DynamoDB automatically spreads the data and traffic over a sufficient number of servers to handle throughput and storage requirements, while maintaining good consistency and fast performance. All the data is stored on solid state disks and automatically replicated across multiple Availability Zones in an AWS region, providing built-in high availability and data durability.

2.3.1 Unlimited horizontal scaling

One of the main features of DynamoDB is its great ability to scale horizontally[2]. That is possible because it models tables as collections of items, and each item as a collection of attributes. For each table, one or a combination of two attributes is used to define its primary key. In this framework, each item contains a partition hash key and possibly a range key. As long as tables partition hash keys are not "hot", i.e. there are not too many items with the same partition key values, accessing individual items and querying for items that exactly match one value on the partition key are both very efficient operations on DynamoDB.

The implementation justification comes from the fact that stored items are distributed in multiple nodes according to their hash keys. As a table grows, more nodes are required so a new hash function is used to split the data and avoid heavy nodes. Naturally, it is only possible to keep all items with the same hash keys values in the same nodes if no partition hash key value is present in too many items.

Since DynamoDB allows composite primary keys, a very common kind of query consists of searching for elements with a exact match on their hash keys and a range of values for the range key. These queries also are very efficient because they explore the database implementation design.

More complex queries can be performed as well by filtering by other

attributes values the results from queries following the above described patterns. However, they consume the same throughput capacity of the simple queries they are constructed after, which is not ideal.

2.3.2 Additional indexes

Another way to allow different queries consists of creating indexes. There exist both local and global secondary indexes on DynamoDB. A single table can contain at most five of each kind of index[3], implemented in different ways to allow different operations.

On the one hand, a local secondary index uses the same node distribution defined by the primary key, but allows a different range key by doubling each item into the same node and arranging the copies in a different order. So it allows efficient queries with the same hash key from the primary key and a new range key[4].

On the other hand, a global secondary index works almost as a completely new table. It copies all the items and redistributes them according to a new key pair. Differently from the primary key, this key does not need to be unique[5]. Efficient queries using a global secondary index look for exact matches on its hash key and optionally a range of values for its range key.

3 State-of-the-art

Tracking activities in a service is a recurring requirement in large-scale services and has been approached in a multitude of ways in the tech industry and at Amazon.

It also has been discussed in academia in many insightful papers.[6][7]

3.1 The standard audit trail framework

The auditing framework can been split in 3 stages which often are managed by different softwares: data collection and shipping, data indexing, and data searching and analysis.

Each of these components can be implemented differently and provide different guarantees on durability, efficiency, independence, security, and customer value.

3.1.1 Data collection and shipping

A log shipper is a software that is responsible for centralizing the logged data into some resource that allows efficient queries. [8]

The expected work of a log shipper includes fetching data from a source (a file, a UNIX socket, among others), processing it (appending a time stamp, parsing unstructured data, adding geographic location based on IP, among others), and shipping it.

3.1.1.1 Consistency vs Independence

It is generally accepted that a trustworthy audit trail system should be consistent and reliable. However, this is not easily achievable and has some downsides.

Ensuring consistency must be done by the data collector component. In fact, a synchronous data collector is the only solution that provides a

hundred percent consistency guarantee.

In this approach, operations are audited before they are performed, so that when they fail to be shipped, the operation is not executed.

Strongly binding operations execution to auditing has some unavoidable disadvantages, that of course can be mitigated. First, a failure in the audit trailing system would result in failures to any operation that it audits. Second, the synchronous approach results in an inevitable increase on the waiting time to have any operation executed.

Other approaches include simply creating an asynchronous job when intercepting operations in exchange for consistency. Another common approach consists of logging operations details locally and periodically shipping log files to a resources management system.

This second approach is particularly bad because outages on the hosts where the activities are performed lead to inevitable data losses. In addition, implementations using this approach usually do not allow real-time logging. Their main advantage is that they make integrating an external auditing system with a client website easier.

3.1.1.2 Handling Big Data

When logging every single activity from every user on a service, one must make sure that the writing throughput capacity of the resource management component can handle all of the requests.

When processing huge amounts of data, a dedicated streaming service designed to handle Big Data might be necessary to be able to collect and process everything in real-time.

3.1.2 Data persistence and indexing

A log indexer must store increasing amounts of data in a way that allows efficient queries and data analysis. It should provide a standard API to perform a wide range of queries that will be made by an searching and analysis tool.

When studying possible log indexers, we concluded that most of the commonly used ones use as their underlying technology either a simple storage service, a text searching engine, or a non-relational database.

3.1.2.1 The downsides of in-place storage

Before I did my project, activities records were only stored on the hosts where our service was running in log files. This approach was not sustainable for many reasons, among which:

1. It was not accessible by developers using our service: they needed to contact someone from our team whenever they needed to know activities history. In addition, it was not easy to make them reliably accessible. Simply running a script to ssh into our hosts and retrieve logs would be a extremely bad software engineering practice from a security and efficiency point of view
2. The currently used logging configuration consistently erases all logs more than one month old: it becomes impossible to track not so old activities
3. It was not immune to hosts outages, that would lead to unrecoverable data losses

Any approach that relies exclusively on local logging cannot be considered the state-of-the-art of auditing activities and is very rudimentary. In fact, this was simply the easiest way to audit activities in an incipient service.

3.1.2.2 Simple Storage Services

One of the least expensive implementations of an activities history feature can be achieved with Amazon Simple Storage Service, simply called S3.

First of all, using a separate service lightens a lot the burden of dealing with scalability, security and durability.

Besides that, by crafting appropriate S3 objects keys, some queries can be performed fairly efficiently.

This approach had already been implemented in another service owned by our team. It uses a combination of services ids and the date of logging as an object key.

Each object has as its key:

```
service id/year/month/day/randomly assigned id
```

This approach allows easily querying for data logged in the same service, year, month or day by calling the API method `listObjects(String bucketName, String prefix)`[9].

By emulating DynamoDB implementation, one could also create "global secondary indexes" to query for activities occurring in a specific object. This could be done by storing a copy of each activity data in a object with key:

```
object id/year/month/day/same randomly assigned id
```

This flexibility is the reason why some more complex approaches use S3 as their main underlying technology.

However, some limitations for this approach are inescapable. Since each activity record is stored in a separate S3 object, accessing it is done by a `getObject` request, which takes fractions of seconds. In fact, when a query returns many results, the sum of the total time and resources spent to perform these calls become unacceptable.

3.1.2.3 Text Search Engines

Another common technology used on log indexers is a text search engine, which is designed for easy information retrieval.

Text search engines like Lucene allow very efficient prefix and range queries as well as combining multiple conditions. For these reasons, even when used with structured data, they can perform complex queries that the other studied approaches cannot.[10]

However, current open-source tools still lack some capabilities. Elasticsearch, the most popular one, is used by some companies to index and search petabytes of data every day, but it cannot be said that it is completely scalable. In fact, Elasticsearch indexes are distributed in static shards, so increasing writing capacity by using new shards is not trivial. A naive strategy that created new shards when they were needed actually would be very slow, as creating the standard procedure to create new shard involves re-indexing the data, which is slow[11]. It is worth mentioning that some predictive approaches are implemented by private services built using Elasticsearch[12].

In addition, resiliency of Elasticsearch data is not as strong a point as in other database services. Since index data management uses a relatively complex infrastructure, data stored in Elasticsearch is prone to loss, as reportedly happened in a 2014 incident and despite the increasing effort of Elastic engineers to ensure resiliency. For this reason, many projects rely on more solid databases as their primary storage resources and use Elasticsearch for indexing and querying efficiently. This introduces a certain degree of complication that should be avoided when possible. In addition, if data is lost on Elasticsearch, it must be re-indexed, which can take very long depending on the amount of data and be painful to users[13].

3.1.2.4 Non-relational Databases

By only using a non-relational database, one can have both strong scalability and durability, and still be able to index and query by multiple attributes.

Logging data to non-relational databases has been studied by Mahmood K., Risch T., and Zhu M[14]. They used MongoDB, a state-of-the-art document-oriented NoSQL database to store large volumes of data logs. Like DynamoDB, MongoDB provides both primary and secondary results.

In their study, they observed that MongoDB did not substantially improved persisting large-scale data logs but still proved to be a viable alternative to relational databases for queries where the choice of an optimal execution plan is not critical. Therefore, for its scalability and easiness of use, MongoDB could replace relational databases in logging systems in most cases.

Compared to using simple storage services, non-relational databases allow actual indexing on attributes other than the time stamp. By extracting relevant attributes to the database, one can perform indexed queries and filter on them.

Following a reasoning similar to when using Elasticsearch coupled with a primary database, one can couple a non-relational database with a cheaper storage service, such as S3. In fact, if some objects attributes are never to be used on queries and only need to be accessed when an individual item is wanted, they can be stored on S3 without any downsides. However, one can expect less flexible queries than the ones provided by Elasticsearch.

3.1.2.5 Relational Databases

As observed in the above mentioned paper, relational databases could be used to store quite large amounts of data and allow efficient queries. However, the relational model would not really bring any benefit to the project, for logged activities are not dependent on any other structure in our service. The best relation we could find was coupling logged data with services, so that deleting a service could delete past activity on it.

In addition, from the CAP theorem[15], we knew the so wanted strong ACID guarantees (atomic transactions, transactional consistency and isolation, and data durability) on relational databases would prevent them to provide complete availability or consistency[16]. In addition, when loading massive logs produced at high rates, relational databases might not be fast enough due to their consistency guarantee and the high cost of indexing.

Therefore, since we could not see any actual value in using a relational database and some risks existed, we concluded that using a key-value or document-oriented database would be a better approach, for they would possibly allow greater scalability and easiness of use.

3.1.3 Data access - searching and analysis

The third component of the auditing framework provides a customer facing interface to perform searching and analysis of the audited data. It can range from a simple set of user API methods to a platform that enables

data visualization and dash-boarding.

This component has less rights and wrongs, as the features it provides are limited to the ones provided by the data persistence and indexing component. If analytics and dashboarding is required, some private-owned and open-source solutions provide complete solutions. If simple searching is wanted, then a user API or a web page can be more adequate.

3.2 Some well-known frameworks

Based on this ideas, multiple frameworks are used in the industry and some AWS products also exist. We studied some of them and attempted to compare them in an illustrative analysis.

3.2.1 Splunk

Splunk products aim to make sense of machine-generated data. They do that by capturing, indexing, and correlating real-time data in a searchable repository from which it can generate graphs, reports, alerts, dashboards, and visualizations[17].

Splunk offers a paid software and has originally grown in response to the demand for comprehensible and actionable data reporting for executives outside a company's IT department. Although the target consumer of the auditing tooling is usually different, the work to store and display logged data follows the same three-component pattern.

In its implementation, Splunk uses a standard API to connect directly to applications and devices[18] and their own search engine for indexing data[19].

3.2.1.1 Splunk specifications

- **Consistency:** Splunk reads from a log file and ships its content as soon as it is modified
- **Scalability:** Splunk architecture is designed for Big Data and scales

horizontally. One can add reading and writing capacity by increasing the number of forwarders and indexer nodes[20]

- **Flexibility:** powered with a search engine, Splunk supports multiple different queries
- **Support for logging:** it is one of the most largely used logging frameworks and a paid software offering customer support
- **Pricing:** when indexing up to 1GB of data per day, each gigabyte costs 2070 dollars

3.2.2 The Elastic Stack

Elasticsearch and the projects Logstash and Kibana, commonly known as the Elastic Stack, can be used together to centralize and index logs.[21]

Elasticsearch is a search engine based on Lucene, an open-source information retrieval software library. It provides a distributed search engine with a web interface and schema-free `JSON` documents[22]. It is responsible for the data indexing role when used in the auditing framework[23].

It is the most popular enterprise search engine, beating Splunk, that ranks third[24].

Logstash is a JRuby application that can read input from several sources, modify it and push it to a multitude of outputs. It needs a configuration file that determines where the data is and what should be done with it. Therefore, when used in the Elastic Stack, it is used to periodically access logged data, do some predefined processing, and store it on Elasticsearch, fulfilling the role of the data collector and shipper.

Finally, Kibana is a JavaScript application that can be used to build dashboards, using data stored in Elasticsearch, thus fulfilling the role of searching and analyzing the data.

One very positive aspect of this stack from an engineering point of view is that all of its components can be replaced by other open-source products more suitable to each project use case, without any significant loss in components integration and efficiency.

3.2.2.1 Elastic Stack specifications

- **Consistency:** Logstash reads from a log file and ships its content as soon as it is modified
- **Scalability:** Although Elasticsearch can handle up to petabytes of data, it does not provide auto-scaling and has reportedly suffered data losses. These issues can be explained by its rather complex architecture
- **Flexibility:** powered with Elasticsearch, that is a search engine, the Elastic Stack supports multiple different queries
- **Support for logging:** it is the most largely used logging framework
- **Pricing:** if deployed on an EC2 instance, Elasticsearch costs 0.018 dollars per hour of use when set up with minimal computational power (1 gigabyte of vCPU and 1 gigabyte of RAM memory)[25]

3.2.3 An internal Amazon Web Services logging service

Given the large spectrum of services provided by AWS and their easy integration with each other, Amazon projects tend to prefer them to external services that frequently are expensive or unreliable. In addition to that, Amazon software engineers usually are more familiar with Amazon products than with external products.

When studying which approach we would use in the project, the AWS internally used logging service was the first studied option.

Despite being a service dedicated to logging, it had some limitations. As a log shipper, it allows both using an API or a log pusher client that periodically reads from local log files. As a log indexer, its main storage resource was S3, which brings strong security and durability guarantees but has limitations on indexing the data. We remarked that it efficiently indexed the data by time stamp, but for other attributes it was only possible to index data logged in the latest 24 hours.

3.2.3.1 The internal AWS service specifications

- **Consistency:** it provides strong auditing consistency when using its API
- **Scalability:** backed by S3, it has strong durability and security guarantees. However, it fails to read data separated by long periods of time
- **Flexibility:** only allows complete indexes on the time stamp. Other indexes can only process data logged in the latest 24 hours
- **Support for logging:** it is one of the most (if not the most) largely used logging framework by AWS developers
- **Pricing:** its pricing is closely related to its S3 usage. S3 pricing is given by 0.023 dollars per gigabyte per month if using standard storage on a global AWS region[26]

3.2.4 CloudWatch Logs

Amazon CloudWatch Logs is a reliable, scalable, and flexible monitoring solution easily integrable with Amazon resources and with other resources through their AWS SDK, available to all AWS EC2 customers.[27]

It is built using Amazon Kinesis Streams, a service to collect and process large streams of data records in real time. For this reason, CloudWatch logs support up to 1500 transactions per AWS account per region per second[28].

A log posting frequency of up to five seconds, good security compliance, and low-cost pricing were the main advantages of using this service. However, no custom indexing option was offered.

3.2.4.1 CloudWatch Logs specifications

- **Consistency:** it consistently reads from a log file to provide near real-time log posting. Log shipping is backed by Amazon Kinesis Streams, which can handle large amounts of data

- **Scalability:** data is compressed, so larger amounts of data can be stored
- **Flexibility:** does not allow any indexes
- **Support for logging:** it is a public AWS service for monitoring and logging EC2 instances and other services, and has a good API support
- **Pricing:** 0.57 dollars per written gigabyte of data and 0.03 per stored gigabyte per month[29]

3.2.5 DynamoDB

DynamoDB fits well the framework of non-relational database coupled with a simple storage service.

Its unlimited storing capacity, the ability to create additional indexes by attributes, and its predictable performance also are interesting advantages.

Although DynamoDB use cases vary a lot, storing time series data is one of them and is described in its user guide[30].

One reason for avoiding DynamoDB could be its relatively high price - it costs ten times more than S3 to store the same amount of data[31][26]. However, storing part of the data on S3 can considerably mitigate this. In addition, the currently costs added to the team budget are quite low.

3.2.5.1 DynamoDB specifications

- **Consistency:** since DynamoDB is only the log indexer, one can choose between consistency and independence freely
- **Scalability:** DynamoDB provides infinite horizontal scalability
- **Flexibility:** Allows up to 5 global and 5 local secondary indexes. However, there is less flexibility than in search engines
- **Support for logging:** storing time-series data is one of the proposed use cases for DynamoDB, among a multitude of valid use cases

- **Pricing:** DynamoDB charges 0.47 dollars per writing capacity unit, 0.09 dollars per reading capacity unity, and 0.25 dollars per stored gigabyte of data per month[31]. The storage costs are ten times the ones for S3. In our use case, we use 5 reading and 5 writing capacity units, which allow a maximum of 5 reading and 5 writing operations per second and have a total cost of 2.8 dollars per audit trail table per month.

3.2.6 Choosing the most adequate framework

A combination of DynamoDB and S3, coupled with a synchronous Java invocation interceptor and a web page, was the chosen framework for the audit trail that we implemented, as it proved to better satisfy the main analyzed aspects of each framework.

Analyzing them one by one, we can first remark that only the implemented framework and the AWS internal service framework provided strong consistence. Most approaches dealt well with scalability and DynamoDB was one of them. It also allowed more flexible queries than the approaches using simple storage services and the ones present in AWS. It is true that it was the least suitable for logging. However, logging was one of the recommended use cases and could be approached well using composite indexes. Finally, its pricing was not bad, specially when compared with Splunk and the Elastic Stack, whose minimal offers were considerably more expensive than our selected one.

4 The implemented audit trail framework

In this project, we finally opted to use a combination of DynamoDB and S3 to store the logged data, as described in the previous section.

The other components of the audit trail framework also received extensive dedication during the project, in particular the implementation of a strongly consistent invocation interceptor, and the design of an API and a web page to see the audited activities.

We will detail the implementation particularities of each of the components defined in the state-of-the-art section.

4.1 Data collection and shipping

4.1.1 The invocation interceptor

An interceptor is a software mechanism that provides the hooks that are needed to introduce additional code dynamically into the execution path of an application. By exploiting interceptors, add behavior such as logging, auditing, validating and profiling actions on runtime execution[32]. Java has supported interceptors since earlier than 2001[33].

In our project, activities are logged to the database as soon as they are performed using an interceptor, so they can be tracked in real-time. This is an arguable advantage compared to the approach of writing to a log file and periodically reading it and shipping its data to the data persistence layer.

In the project, intercepting calls to a newly implemented method is as simple as adding an annotation `@WithAuditTrail` before the method's declaration.

4.1.2 The collected data

The audited activities are calls to our service standard API, and can be either performed directly or using the website. Since each `HTTP` request makes at least one call to the API, auditing read-only operations (called in

`HTTP GET` requests) would be expensive and useless most of the time. So only mutating operations are logged to the audit trail.

4.1.3 Handling failures when logging

Since posting to the audit trail is a relatively complex process, errors can happen, even if they are never expected to. Handling failures in a way to ensure consistency is an important issue. In fact, in a simple approach that posts to the audit trail once after the operation is executed, a failure on the posting method produces an error status for the whole API call. Our users that take automated decisions based on operations results would think that the operation failed while it did not. In addition to that, this operation would no be logged to the audit trail in any moment, so it could not be tracked by our users.

To avoid that, a two-time logging approach was adopted, as described in Algorithm 1. In this approach, when an activity method is called, the interceptor communicates with the storage resources twice using their common APIs. First, it logs the methods arguments, time of the call, requester name, and request id parameters to the audit trail table. If writing to the audit trail succeeds, it then invokes the intercepted method, that is executed. The output of this execution is then appended to the same audit trail entry in the storage resources using a second call to their APIs. Any exception thrown in this second posting attempt is escaped to avoid inconsistencies.

This approach guarantees that failures when posting to the audit trail will never lead to inconsistencies. It also ensures that all executed operations are present in the audit trail.

Finally, one aspect worth mentioning is the possibility of running the `CreateComplement` method asynchronously. Since we escape eventual failures when executing it, it cannot impact on the activity response. However, if run synchronously, the activity result will only be returned after the posting action is complete, which increases the total time of all activities executions unnecessarily.

The current implementation still is synchronous, but this will change.

Algorithm 1 Intercepting activities and handling failures

```
1: procedure INTERCEPT(invoker, activity, arguments)
2:     audit_trail_entry ← create audit trail entry activity, arguments
3:     post audit trail entry audit_trail_entry
4:     try
5:         output ← invoker.invoke activity, arguments
6:         returnoutput
7:     catch exception
8:         error ← exception.cause
9:         throw error
10:    finally
11:        complementary_entry ← create complement audit_trail_entry, output, error
12:        post result complementary_entry
13:    end try
14: procedure CREATEAUDITTRAILENTRY(activity, arguments)
15:    request_id ← activity.context.request_id
16:    operation ← activity.context.operation
17:    timestamp ← activity.context.timestamp
18:    requester ← fetch arguments, 'requester'
19:    service ← fetch arguments, 'service', null
20:    attribute_1 ← fetch arguments, 'attribute_1', null
21:    attribute_2 ← fetch arguments, 'attribute_2', null
22:    attribute_3 ← fetch arguments, 'attribute_3', null
23:    attribute_4 ← fetch arguments, 'attribute_4', null
24:    parameters_json ← arguments as JSON string
25:    result ← 'STARTED'
26:    return new audit trail entry with constructed parameters
27: procedure CREATECOMPLEMENT(audit_trail_entry, output, error)
28:    id ← audit_trail_entry.id
29:    attribute_1 ← fetch arguments, 'attribute_1', null
30:    attribute_2 ← fetch arguments, 'attribute_2', null
31:    attribute_3 ← fetch arguments, 'attribute_3', null
32:    attribute_4 ← fetch arguments, 'attribute_4', null
33:    if output == null then output_json ← error as JSON string
34:    else output_json ← output as JSON string
35:    if error == null then result ← 'SUCCEEDED'
36:    else result ← 'FAILED'
37:    return new audit trail entry with constructed parameters
```

4.2 Data persistence and indexing

4.2.1 The Audit Trail Entry schema

As a possibly useful reference for the following sections, I am listing the expected attributes in an audit trail entry. Some of them can be null.

- Service
- Time stamp as a string suffixed with a random id
- Operation name
- Result code: status of the operation - `STARTED`, `SUCCEEDED` or `FAILED`
- Requester: user that performs the operation
- Request id: unique identifier of a sequential series of activities
- Attribute 1
- Attribute 2
- Attribute 3
- Attribute 4
- Parameters JSON
- Output JSON

4.2.2 Data serializing

As described in Algorithm 1, some of the audit trail entries fields are retrieved from the activity context while others come from the parameters and output JSON blobs, which are string representations of the activity request and response objects, respectively.

4.2.3 The DynamoDB table

All the parameters other than the parameters and output JSON blobs from audit trail entries are logged to DynamoDB, either because they are used when performing queries on indexes or because we believed that they should be promptly visible to users of the audit trail in the results from queries, in particular when they would see them in the website. Attributes that fell in the latter case were the operation name, the result code, the requester and the request id.

4.2.3.1 The primary key

Following DynamoDB developer guide for storing time series data, a combination of the service name and the time of logging was our selected primary key. The choice for the partition key was quite natural, as all audited operations are performed on a service, which therefore is a required attribute of audit trail entries. In addition, a user will only see activities occurring on his service, so searching for all activities history in a service is quite common. Having the time of logging as a range key allowed queries by time range, which also was one of the requirements for the feature.

One simple detail of implementation consisted of appending a randomly generated suffix to the time stamp string to ensure the uniqueness of the primary key. Since the precision for the time of logging obtained through the `activity.context.timestamp` statement went only up to seconds, collisions were quite likely to happen a few times. By appending a six-character random suffix generated when first writing an item to the table, this chances became equal to 1 in $52^6 \sim 2 * 10^{10}$ for activities logged in the same second for the same service.

Out of curiosity, one can estimate that if two activities were logged by second for a period of six hundred years only one collision would be expected to happen. Currently, none of our customers services is producing more than `10` items per day. That means that each service has a really negligible probability of facing a collision.

4.2.3.2 The indexes

The main purpose of creating indexes was to implement interesting queries efficiently. As, from the beginning, the audit trail feature was expected to provide queries by four attributes, besides requester names and request ids.

Since all of these queries should allow searches by date range, using a global secondary index would be ideal, following the description in section 2.3.2 at page 13, in which one of the attributes would be the partition hash key and the time stamp would be the range key. Due to the limit of five global secondary indexes, we decided to use four global secondary indexes and two local secondary indexes.

The key observation was that some of these queries would always return a very low number of items: in particular, there would never be more than a few activities with the same value for the request id or attribute 4. So for queries on these attributes local secondary indexes were created. API queries that specify time ranges actually call the DynamoDB API without the time range and locally filter and sort the results from this API call.

Therefore, the final set of secondary indexes was:

- Local secondary index: Service name (hash key) + Request id (range key)
- Local secondary index: Service name (hash key) + Attribute 4 (range key)
- Global secondary index: Attribute 1 (hash key) + 'Timestamp + suffix' (range key)
- Global secondary index: Attribute 2 (hash key) + 'Timestamp + suffix id' (range key)
- Global secondary index: Attribute 3 (hash key) + 'Timestamp + suffix id' (range key)
- Global secondary index: Requester (hash key) + 'Timestamp + suffix id' (range key)

One last remark on the DynamoDB table must be made. Using local secondary indexes made us lose DynamoDB promised infinite scalability property[3]. For, when using local indexes, DynamoDB can only handle `10 GB` of data for each hash key value. So each of our users is bound to `10 GB` of data, which corresponds to approximately $4 * 10^7$ items. Since in two weeks of use none of our services has generated more than a thousand items, this limit will most probably never be reached. However, two mitigation strategies were adopted: a feature to delete items older than three years was added to the table[34] (though not currently activated), and metrics and an alarm on the size of the table will be implemented soon.

4.2.4 The S3 bucket

The S3 bucket is used to store the parameters and outputs JSON blobs, respectively coming from activities request and response objects. The decision to store them on S3 instead of DynamoDB came from the observation that they would be very little useful when querying for audit trail items and were too long to be displayed in a table column in the website, where they would produce more visual noise than any actual benefit.

In addition to that, we already knew that storing on S3 was more than ten times cheaper than on DynamoDB.

4.2.4.1 S3 objects keys

Each audit trail entry has its parameters and output stored in two different objects with unique object keys ensured by the DynamoDB primary key uniqueness.

The adopted approach looks quite similar to the one described on section 3.1.2.2 at page 17

- parameters JSON blobs are being stored on the unique path: `parameters/service_name/timestamp/suffix`
- output JSON blobs are being stored on the unique path: `output/service_name/timestamp/suffix`

4.3 Data access - searching and analysis

4.3.1 The API

Access to the data on DynamoDB and S3 was implemented using the AWS SDK, with the `DynamoDBMapper`[35] and `AmazonS3Client`[9] classes.

Two customer-facing methods were implemented:

1. `ListAuditTrailEntries`: performs search queries against the DynamoDB table.

   A service name must be provided in the arguments. Specifying an entity to query by (such as an attribute or a requester name) is optional. Date ranges and expected result codes are also optional.

   The output contains all attributes from the retrieved items that are present on the DynamoDB table.

2. `DescribeAuditTrailEntry`: retrieves all the parameters for a specific audit trail entry.

   The audit trail entry primary key must be provided: a service name and a suffixed time stamp.

   The output contains attributes with all the details from the requested item without redundancy: service name, suffixed time stamp, result code, requester, request id, parameters JSON blob, and output JSON blob. Attributes that can be seen on the parameters and output JSON blobs are not present.

4.3.2 The website

An interactive web page was added to the website providing the operations available through the API, as shown in the figures below. It was developed using the Ruby on Rails framework.

Figure 1: The audit trail web page

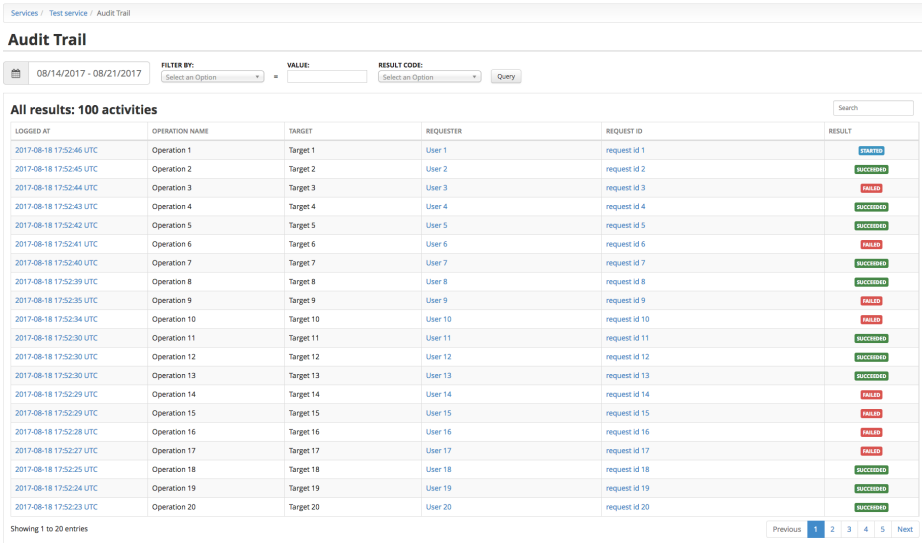


Figure 2: Clicking to see parameters and output from a failed operation

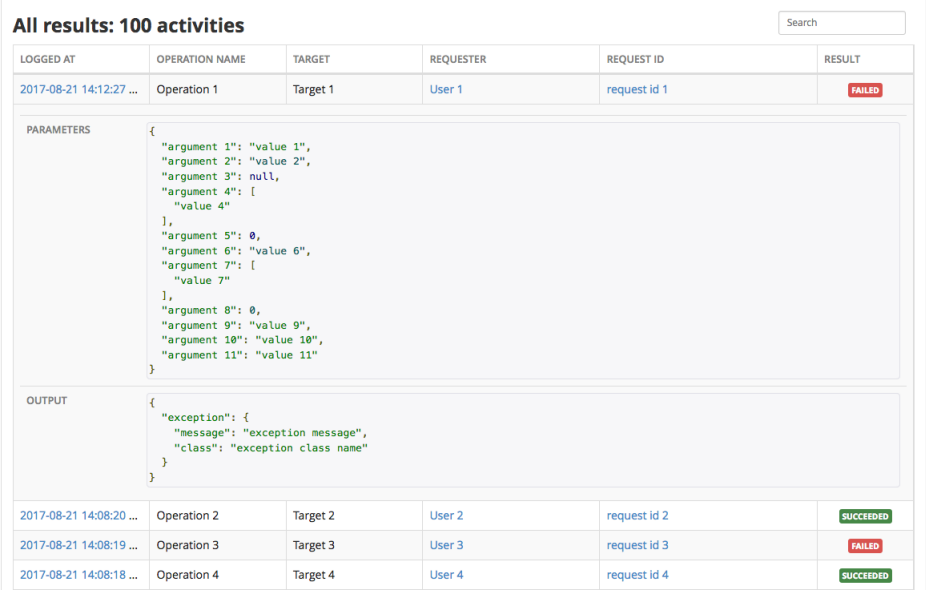


Figure 3: Selecting date range, attribute to index the query on, and result code expected

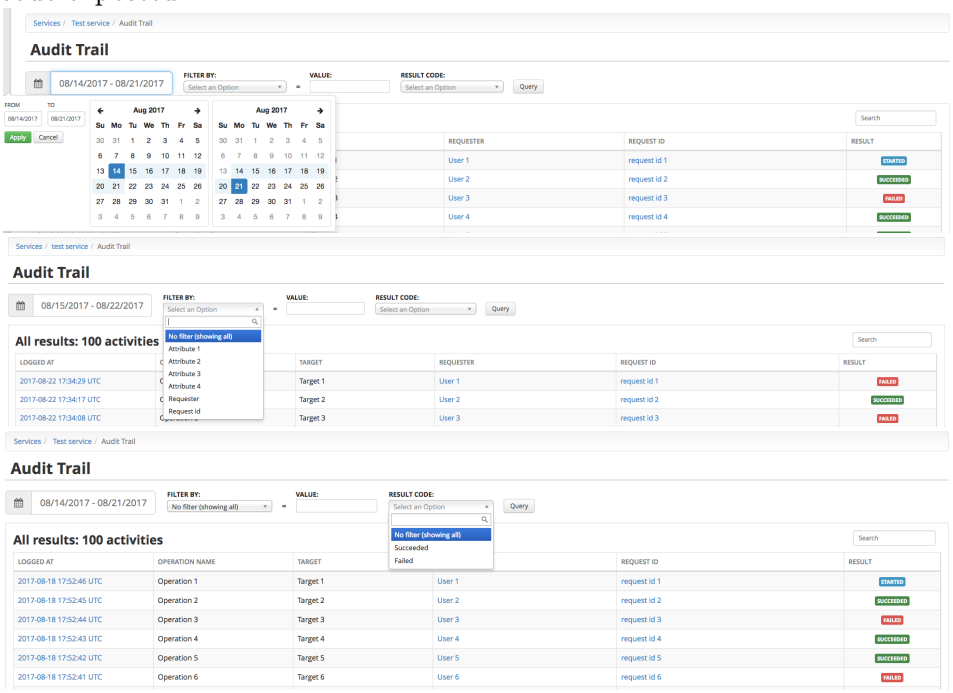
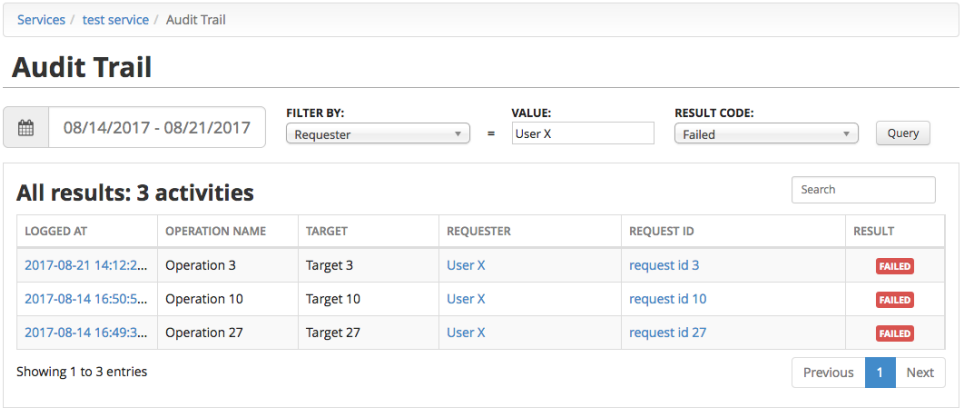


Figure 4: Results from querying by requester name and filtering by result code



4.4 Other considerations

Despite not being project core ideas, these considerations were indispensable for its success.

4.4.1 Security considerations

Ensuring that the used data resources are correctly secured against malicious attacks is important in every software company. Encrypting them using server side or client side is one of the multiple security requirements for services developed at Amazon. For this reason, the created S3 buckets have server side encryption (immune to host failures), and the DynamoDB tables data will be encrypted in quarter 4 (from October to December), according to the team eighteen-month planning.

Avoiding website attacks such as JavaScript injection and request forgery were also concerns we had to have during the web development phase of my project. One interesting issue we had to deal with was stored cross-site scripting. An attacker that performed requests with maliciously crafted parameters would expect to see them when displaying the audited activities. If they contained JavaScript code, an unaware implementation could easily allow such scripts to be executed.

4.4.2 Automated tests

All the implemented code was unit tested. Acceptance tests were developed and run against all website features.

Numerically, the Java code had 73% of its lines covered by unit tests while the Ruby on Rails code had a 88% line coverage. More than 50 acceptance tests were implemented and successfully run against the audit trail web page to emulate user behavior.

4.4.3 Metrics and alarms

Not only a strategy to detect and predict failures, metrics and alarms are used to measure the progress and confirm the correct behavior of services operations.

We briefly analyze the performance of the feature on the US East region, where an estimated 70% of AWS IPs were housed on 2012[36], still considered one of Amazon Web Services most important regions.

4.4.3.1 Audit Trailing API calls

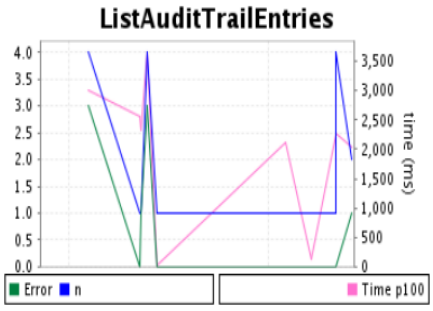


Figure 5: `ListAuditTrailEntries` calls on AWS US East region

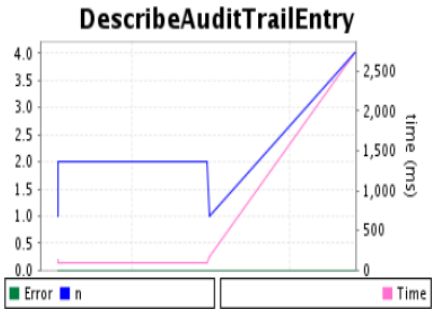


Figure 6: `DescribeAuditTrailEntry` calls on AWS US East region

The graphics show that both methods are being used relatively frequently. Both APIs activity maximum time of execution (or p100) were significantly inferior than 500ms in the last 10 days.

Although some errors were recorded in `ListAuditTrailEntries` calls, they should not cause any concern and can be ignored as the page is still being tested by developers and errors actually are the "expected behavior" in some cases. For example, an internal failure error should be produced when one tries to see the audits from a service one does not own by typing the exact URL path. This is the same behavior expected from trying to access non-existing pages and is not a bug, for there are no links to do this in the website.

Finally, systematic errors are being tracked by alarms that notify the

team when at least one failure occurs in each of five five-minute periods in a row.

4.4.3.2 Posting to the audit trail

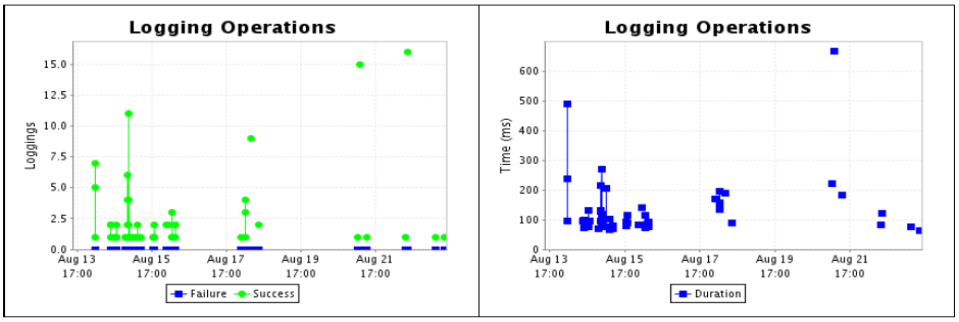


Figure 7: Successes/failures of posting activities data to the audit trail (left) and p100 (maximum) duration of these posting operations (right). Each data point represents a five-minute period with some existent activity

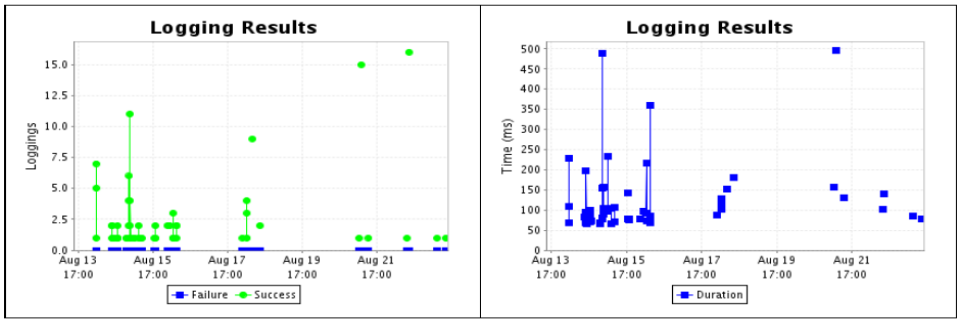


Figure 8: Successes/failures of posting activities results data to the audit trail after they are performed (left) and p100 (maximum) duration of these posting operations (right). Each data point represents a five-minute period with some existent activity

These graphics show a zero-error posting method with relatively good efficiency. Posting to the audit trail consumes some time for each API call, but in average it is considerably slower than 500ms when summing the two posting calls.

Nevertheless, periods with dense calls proved to have slower logging.

This suggests that the DynamoDB table capacity throughput is lower than it should be. As one can see in the graphics, the aggregate time to log the slowest operation in the past 10 days was close to 1.5 seconds. Thankfully, this can time can be reduced by increasing capacity throughput, which can be done with a simple API call. In addition, once posting results is made asynchronously, the actual maximum time added will be almost always of less than 0.7 seconds, which is more acceptable.

5 Conclusion

Audit trailing activities in a service is an interesting and approachable however open problem.

Balancing consistency and independence, handling Big Data, ensuring data durability and scalability, and providing meaningful queries are some of the challenges that one faces when developing an audit trail framework. Separating the framework in thee main components can be helpful to identify these challenges and engineer a solution to address them.

We could observe that the studied technologies used in the industry and at Amazon were good solutions for different use cases. On the data collection side, most of them favored independence in exchange for consistency, which was wanted in our project. On the data persistence side, solutions powered by search engines allowed more flexible queries but did not provide sufficient durability or were disproportionately expensive, while solutions backed by S3 provided queries too limited to our use case. DynamoDB provided enough flexibility and durability for a reasonable price. Finally, on the data search and analysis side, some approaches provided simple APIs while others allowed to build dashboards and statistics for the audited data. We opted for developing an API and a web page.

We believe that the selected and implemented framework can be extended to any use case in which one wants to be able to search for activities containing specific attributes or performed by a specific user.

The Audit Trail Framework developed in this internship went in production code and proved to meet all the initial requirements from theoretical and practical perspectives, in particular through extensive automated tests and careful metrics review.

This internship was a great opportunity to understand the standard auditing framework and also a good opportunity to understand how research and investigation can take place in an industry environment and how it is applied for the development of product features on Software Engineering.

References

[1] Rangel D. *DynamoDB: everything you need to know about Amazon Web Service's NoSQL Database*. CreateSpace Independent Publishing Platform, USA, 2015.

[2] Sivasubramanian S. Amazon dynamodb: a seamlessly scalable non-relational database service. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[3] Limits in dynamodb, 2012.

[4] Best practices for local secondary indexes, 2012.

[5] Best practices for global secondary indexes, 2012.

[6] Singh S. Cluster-level logging of containers with containers. *ACM Queue*, 2016.

[7] Murugesan P. and Ray I. Audit log management in mongodb. *2014 IEEE World Congress on Services*, 2014.

[8] R. Gheorghe. Five logstash alternatives, 2016.

[9] Amazon s3 client sdk, 2016.

[10] What is in a lucene index, 2013.

[11] Elasitcsearch reindex api, 2017.

[12] The complete guide to the elk stack, 2017.

[13] Resiliency and elasticsearch, 2014.

[14] Zhu M. Mahmood K., Risch T. Utilizing a nosql data store for scalable log analysis. *ACM - International Database Engineering and Applications Symposium*, 2015.

[15] Gilbert S. and Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 2002.

[16] M. Rys. Scalable sql. *Communications of the ACM*, 2011.

[17] P. Tam. Start-ups aim to help tame corporate data, 2009.

[18] T. Olavsrud. Splunk updates platform, adds monitoring and analytics services, 2015.

[19] How splunk indexes work - documentation, 2017.

[20] Splunk distributed deployment, 2017.

[21] F. Hopf. Use cases for elasticsearch: Index and search log files, 2014.

[22] Tong C. Gormley C. *Elasticsearch: The Definitive Guide*. O'Reilly Media, Inc., 2015.

[23] Holmes R. Kononenko O., Baysal O. Mining modern repositories with elasticsearch. *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014.

[24] Databases engines ranking of search engines, 2017.

[25] Elasticsearch pricing, 2017.

[26] Amazon s3 pricing, 2017.

[27] Amazon cloudwatch documentation, 2013.

[28] Cloudwatch logs limits, 2017.

[29] Cloudwatch pricing, 2017.

[30] Dynamodb: Design patterns and best practices, slide 35, 2016.

[31] Amazon dynamodb pricing, 2017.

[32] Mathur A. Wang Q. Interceptor based constraint violation detection. *Institute of Electrical and Electronics Engineers*, 2005.

[33] Melliar-Smith P. M. Narasimhan N., Moser L. E. Interceptors for java remote method invocation. *Concurrency and Computation, Practice and Experience*, 2001.

[34] Dynamodb time to live, 2017.

[35] Dynamodb mapper sdk, 2013.

[36] Amazon data center sizes, 2012.