Paavo Nykänen

Error monitoring in a distributed system

Master's Thesis in Information Technology

May 8, 2024

University of Jyväskylä

Faculty of Information Technology

Author: Paavo Nykänen

Contact information: paavonykanen@gmail.com

Supervisor: Tommi Mikkonen, tommi.j.mikkonen@jyu.fi

Title: Error monitoring in a distributed system

Työn nimi: Virheiden monitorointi hajautetussa järjestelmässä

Project: Master's Thesis

Study line: Software and Telecommunication Technology

Page count: 72+0

Abstract: While the concept of distributed systems encapsulating and isolating functionalities and errors inside their services is a great benefit for a system, it can also introduce various problems. One of these drawbacks is that if an error occurs in one of the services, the rest of the system is not aware of it and the error might go unnoticed. This would delay fixing it and allow the same error to keep repeating and causing issues. This problem is addressed in this thesis by creating and applying an error monitoring framework for a single service of a distributed system. The framework is supposed to monitor for different kinds of errors in the target service and its resources. When these metrics notice errors, they should be indicated to the developers or system admins in different ways like notifications or visually on a monitoring dashboard. The framework is also tested by applying it to Akamon Innovations' Dataplatform system's *Timeseries* service and measured against various need statements given by Akamon.

Keywords: Distributed Systems, Error Monitoring, AWS, Cloud Computing

Suomenkielinen tiivistelmä: Yksi hajautettujen järjestelmien suurimpia etuja on niiden tapa kapseloida ja eristää toiminnallisuuksia ja virheitä yksittäisten palvelujen sisään jolloin ne eivät häiritse muun järjestelmän toimintaa. Tämä voi kuitenkin luoda uuden ongelman jos virheistä ei kommunikoida palvelun ulkopuolelle. Virheet voivat tällöin jäädä kehittäjiltä huomaamatta jolloin ne jatkavat esiintymistään, vahinkojen aiheuttamista ja niiden ko-

rjaus viivästyy. Tähän ongelmaan kehitetään ratkaisu tässä tutkielmassa luomalla viitekehys virheiden monitoroinnille hajautetussa järjestelmässä. Viitekehyksen on tarkoitus huomata erilaisia virheitä kohde palvelussa ja tuoda ne kehittäjien ja järjestelmän valvojien tietoisuuteen esimerkiksi ilmoituksilla tai monitorointtikäyttöliittymillä. Viitekehys testataan Akamon Innovationin Dataplatform-järjestelmän *Timeseries*-palvelussa ja sitä arvioidaan Akamonin antamia tarvevaatimuksia vastaan.

Avainsanat: Hajautetut järjestelmät, Virheiden monitorointi, AWS, Pilvipalvelut

Preface

Thank you to Akamon Innovations for providing the subject and guidance for this thesis. Thank you also to my supervisor Tommi Mikkonen.

Jyväskylä, May 8, 2024

Paavo Nykänen

Glossary

ARN	Amazon Resource Name is a unique identifier specifying an AWS resource explicitly within a platform or environment.
AWS	Short for Amazon Web Services, an on-demand cloud computing provider which offers different platforms, APIs, services and resources for companies or individuals.
Cloud provider	An IT company that hosts, manages and provides scalable on-demand computing resources over the internet.
CloudFormation	AWS service used for modeling and setting up usage of AWS resources to automate provisioning and configuring them.
CloudWatch	AWS service used for monitoring resources, applications and infrastructure on the AWS cloud.
CodePipeline	AWS service used to enable continuous delivery with modeling, visualizing and automating the steps of releasing software.
CRUD	Abbreviation for *CREATE*, *READ*, *UPDATE* and *DELETE*, which are the basic operations to implement a persistent database.
DLQ	A feature of SQS that is used by source queues for their unprocessed messages to debug or handle the errors that caused the message processing to fail.
EDA	Event-driven architecture is a software design pattern in modern microservices where the system's services activate triggers and communicate with events carrying some state or identifier to depict an action happening.
FaaS	Function-as-a-Service is a cloud service used to provide a platform for running custom functions without the need for creating or managing infrastructure for them.
IaaS	Infrastructure-as-a-Service is a cloud service used to provide computation, storage, and networking resources on demand over the web on a pay-per-use basis
IAC	Infrastructure-as-Code is a technique for creating, managing

	and automating IT system infrastructure as if it were software and data.
IT	Information Technology refers to the use of computers, storage, networking and physical devices, infrastructure and processes to manage electronic data.
IPC	A mechanism that allows processes or services to communicate with each other and synchronize their actions.
KPI	Key Performance Indicator, which can mean any metric that can measure performance in some capacity such as errors, resource usage or computation power.
PaaS	Platform-as-a-Service is a cloud service where an entire software development and deployment environment with resources, development tools and infrastructure is provided on a pay-per-use basis.
SaaS	Software-as-a-Service is a cloud service that provides complete cloud-based applications for the use of their customers.
SES	Simple Email Service is a platform for managing, sending and receiving emails with configured addresses and domains.
SQS	Simple Queue Service is a scalable messaging service offered by AWS that allows distributed and decoupled distributed services to communicate through hosted message queues.
X-Ray	X-Ray is an AWS service that collects data about an applications requests and offers tools to view, filter and analyze them for fixing issues or optimization.

List of Figures

Contents

1 Introduction

Distributed systems have become a common architectural paradigm in software development, especially after the rising popularity of cloud computing and cloud services. Distributed systems are made out of multiple independent computation pieces that strive to achieve the functionality of a singular system by communicating with each other (Thoke 2014). Each computation piece is responsible for their own actions and functionality and should encapsulate themselves so that there would be minimal dependencies between different parts. Van Steen and Tanenbaum (2016) and Thoke (2014) explain that the strengths of distributed systems are high scalability and load balancing, improved performance by concurrency, flexibility and modularity, fault isolation and tolerance and increased reliability. They also list distributed systems weaknesses to include increased communication overhead, higher latency, network dependency, complexity of infrastructure, security risks and difficulties in debugging, monitoring and troubleshooting.

This thesis will focus on solving the difficulty of monitoring and troubleshooting errors in different services of a distributed system. This problem is caused by distributed systems services encapsulating their functionality so that their internal errors would not affect the rest of the system. This also causes the errors to be hidden from the outside, which means that they will easily go unnoticed, keep causing problems and will not be fixed. This problem also occurred on Akamon Innovations' Dataplatform system, where they noticed that if an error occurred on some part of the system, they might only notice it by chance or if an end user brought it to their attention.

To solve this problem a framework was developed for monitoring distributed systems which allowed different errors to be tracked in the systems' services and brought to the developers' attention by notifications and monitoring dashboards. The framework was tested by applying it to one of Dataplatform's services and analyzed for how well it works and fits the given requirements. Design science was chosen as the research strategy, since it revolves around developing and analyzing an artefact in development cycles. Firstly existing research, technologies and frameworks for similar use cases and systems were researched. Then the framework was designed and developed so that it would fit the subject system and the re-

quirements set for it. After that the framework was analyzed to see how well it matched the given requirements and solved the research questions. If it did not fulfill all of the requirements, new development cycles were created and gone through until it did.

Chapter 2 will go over and explain distributed systems, common terms and technologies in using or developing them and explain more about Akamon Innovations' Dataplatform-system, which will be the subject of the developed framework. Chapter 3 will go over what monitoring is and how it is done in distributed systems, as well as explore existing solutions that could fit or be used for the solution of this thesis. Chapter 4 will explain the research objectives of this thesis, the research problem and strategy, the design process we that will be used and the evaluation criteria for the framework. Chapter 5 will go over the development and implementation of the framework. Chapter 6 will go over the results of applying the framework to the target service and analyze and evaluate how well the framework fills its requirements as well as describe any further development subjects. Finally Chapter 7 will summarize the contents and results of this thesis.

2 Distributed systems

With the development and evolution of computers and networking, it has become possible to create a whole new type of computer system based on multiple independent computers working together over a designated network. These kinds of systems are called distributed systems, based on their ability to host the systems independent computers geographically very separately, as long as they are connected to a common network. This network is used to bind the computers together to create a cohesive singular system and to allow the systems computers to communicate and interact with each other. The concept of distributed systems has been around for quite a long time already but technology is still being developed for it and it has become a very popular concept amongst technology companies.

This chapter will first study and define what distributed systems are and how they work in Section 2.1 as well as go over different types of distributed systems in Section 2.2. After that some common modern technologies that are being used with distributed systems are discussed in Section 2.3. Lastly the Dataplatform information system by Akamon Innovations will be explored in Section 2.4 since it is the subject system for the framework developed in this thesis.

2.1 What are distributed systems

A way to define a distributed system would be that it is a collection of autonomous computing pieces that appear to the user as a single coherent system according to Van Steen and Tanenbaum (2016). From this definition two important characteristics of a distributed system can be found. The first is that it is composed of autonomous or independent computing elements. These elements could be any kind of computing units like computers, IoT devices or even software running on the cloud. The term independent for these units means that they do not rely on each other to work, but rather encapsulate their functionality and do their computation independently from the other units of the system. This would mean that if one computing unit stops working or responding, it would not break the entire system, but the other units could still work and perform their own actions and keep the system running.

The second important characteristic from the definition of Van Steen and Tanenbaum (2016) is that the system should appear to its users as a single system. This means that even though the system is made up of many independent computing units, it is supposed to feel and look like a single system to the outside. For this goal the system's units are expected to work together and communicate with each other to achieve the system's functionality (Thoke 2014). The independence and loose coupling of the computing units also ensures that if one part of the system stops working, it would not break the entire system, so it would still seem like a singular system where only a part of it would not work. Comparing this to a traditional monolithic system where a problem in one part would cause the entire system to malfunction, distributed systems can achieve better fault tolerance while still seeming like a singular system like a monolithic one.

These aspects are also brought up by Verissimo and Rodrigues (2001) as they gave three aspects that define a distributed system. The first is that the system needs to include multiple computers, just as explained earlier that the systems are composed of multiple computation units. The second aspect is that the different computers need to be connected through a common network, as mentioned above as the way to make the computation units work together. The third aspect is that the systems parts need to share their state. This means that while each piece of the system has its own state according to the encapsulation principle, the pieces need to communicate this state to others so that the system that they comprise can have a defined state. This would tie in with the way that distributed systems are supposed to appear as single cohesive systems with a defined state to the outside.

2.2 Different types of distributed systems

Van Steen and Tanenbaum (2016) explain three different types of distributed systems based on the types of units that the systems are made out of. These three types are distributed computing systems, distributed information systems and pervasive systems. Distributed computing systems can be further divided into cluster computing, grid computing and cloud computing.

Cluster computing systems are made out of many identical or very similar high-performance

computers that are used together to increase the computational power and performance of the entire system. By increasing the number of computation units and organizing workload distribution and shared memory, the system's tasks can be split into smaller pieces and solved in parallel on the computation units. Afterwards the results are brought together from the computation units to solve the original larger task. These types of systems are generally used in projects that require high computation power like scientific research and simulation (Yeo et al. 2006).

Grid computing on the other hand is a system that is composed of many different kinds of computation units that might differ by their software, hardware or network topology. They are often systems aimed at creating federation for smaller systems or individual units so that software can be run on different computers of the grid rather than always needing to use the same one. So while cluster computing is focused on running single large jobs split into smaller parts in parallel, grid computing is used for running many independent single-machine jobs or batches of jobs (Iosup and Epema 2011).

The third type of distributed computing and the type of the system that is used as the target in thesis is cloud computing. The idea of cloud computing is to buy computation power and virtualized resources like storage and networking from cloud providers such as institutions or private companies. These cloud providers can offer their services or resources in different layers, which are Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) (Srinivas, Reddy, and Qyser 2012). Cloud providers have made cloud computation very easy to adopt for any types of companies by creating lots of different tools and services and allowing the user to dynamically increase their usage according to their needs and paying by usage or subscriptions.

Distributed information systems are essentially information systems that are made available by different endpoints and allow various kinds of client systems to contact and perform operations on them. Also known as web services, the idea is to expose an information system through common web technologies and standards (Alonso et al. 2004). This allows better interoperability and system integrations and reduces heterogeneity between different components in the target system. By using common web standards to expose the system, new components could be continuously developed and integrated into the system without needing

to make sure that they are compatible. The systems parts could internally be built in whatever technology or architecture, but as long as their exposed interfaces followed the same standard they could be bound together. The development of web services has also led to further advances in service-oriented architecture, an architectural style of IT systems that use small independent web-based services, that are individually tied to certain business functionalities, to be consumed by clients from different applications (Oracle 2005).

Pervasive systems, also known as ubiquitous computing, are the third type of distributed systems and they differ quite a lot from distributed information systems and distributed computing systems. Pervasive systems are composed of multiple kinds of IoT, mobile and embedded devices and systems, often integrated into their environment. Pervasive systems should be distributed, connected through a common network and transparently accessible according to Poslad (2011). They should also aim to hide the human-computer interaction and to be context aware to achieve a seamless interaction with their environment and be able to observe it.

2.3 Technologies in distributed systems

Even though the concept of distributed systems has been around for some time, new technologies for distributed systems are continuously being developed. Earlier different types of distributed systems were studied and it became clear that the concept covers a lot of different types with even more technologies that can be used with each type. Since the target system of this thesis is classified as a distributed information system, this section will go over some of the most common technologies used in those systems while focusing on the technologies that are used in the target Akamon Dataplatform information system.

2.3.1 Cloud services and providers

Providing computing resources over the web has become a very popular business field for many IT companies. For example, Amazon Web Services (AWS) offers a wide variety of on-demand cloud computing platforms, services, infrastructure and APIs to consumers and companies, which are highly reliable, easily scalable and relatively low-cost (Mathew and

Varia 2014). By using cloud services companies can access a shared pool of affordable computation resources by hosting applications through virtualization, accommodating for diverse computational requirements with a single physical infrastructure without the need to invest in the infrastructure itself (Srinivas, Reddy, and Qyser 2012). Scaling resource usage with cloud providers is made easy with pay-by-usage and subscription-based allocation. This means that customers only need to pay for the resources that they use and if they would need more resources like computation power, they can just scale their usage up and pay accordingly, instead of needing to add more computers to their local systems.

Cloud providers such as Azure and AWS offer many different cloud services that customers can use and combine according to their own needs. These include computation services, networking, storage, databases, analytics, application services, deployment and management and mobile applications (Mathew and Varia 2014). Since the target system in this thesis is heavily based on AWS resources, services and infrastructure, some of the most common AWS services are explored next.

For computation AWS offers the Elastic Compute Cloud or EC2, which provides resizable compute capacity with containers meant to make web-scale computing easier and faster for developers. EC2 also includes load-balancing, auto-scaling, lambdas and other useful features for development. For networking, AWS offers the Virtual Private Cloud (VPC), which allows customers to create and provision a logically isolated section of the AWS cloud. They can manage AWS services within that network and configure the network themselves. For storage AWS offers the Simple Storage Service or S3, which provides a safe, secure and highly scalable object storage for developers. Different use cases for S3 could be back-up storage, content storage for images or files, data storage for analysis and many more. AWS also offers many types of databases, like the Relational Database Service (RDS) for relational databases or DynamoDB for NoSQL databases. AWS also offers the Simple Queue Service (SQS) for messaging and message queues, which can be used for example in messaging between services in distributed systems. AWS also provides services for CI/CD pipelines and DevOps work, as well as identity management for service security.

These were just a few examples of the different kinds of services that AWS supports and provides. The most important feature of AWS is that the developers are free to choose the

resources that they wish to use and pay for them according to their usage. With such a wide collection of different services and resources, AWS allows the developers to create all kinds of different applications, services, platforms and anything that they need to. AWS has also created the AWS Software Development Kit (SDK) to help the developers in their work with AWS resources.

2.3.2 REST

REST has become a common approach in modern web services and their interfaces. It is an architectural style or practice in the relationship between a client and a server (Fielding 2000). As the name Representational State Transfer or REST implies, its focus is to perform the operations requested by the client without holding information about the server's state. Applications or services that implement the REST architecture, also called RESTful services, are expected to expose and enable CRUD operations on the server through HTTP methods such as *GET*, *PUT*, *POST* and *DELETE* (Halili, Ramadani, et al. 2018).

Halili, Ramadani, et al. (2018) explain that REST has become the go-to option for system interaction by the usage of RESTful web services as the way that cloud providers deliver their services. REST has been adopted as a kind of standard in distributed systems, with many big technology companies also using it. Adopting a good REST architecture will make the services easier to access by using a common standard, which will also reduce the learning curve.

The learning curve and comprehensibility of the REST architecture is also supported by its flexibility for message formats, by being able to communicate with JSON, plaintext and other known common message formats. Using these simpler message formats allows REST to have better performance and lighter bandwidth, when compared to SOAP for example. These simpler messages also make it easier for any clients to consume the messages. The simplicity, variety of message formats and a common standard for operation makes REST a very good architecture to use in distributed systems, which want to communicate with each other without tightly coupling their interfaces.

REST of course is not the best solution for all kinds of problems but a good option to consider

when developing software. Halili, Ramadani, et al. (2018) also explain that REST can have disadvantages such as being too light weight for communication that requires moving large amounts of data. REST also depends on the HTTP standard for security and is not very reliable if it is not done correctly.

2.3.3 Cloud service classification

As already explained in Section 2.2, AWS like many other cloud providers have settled in on three main classes for cloud services. These classes are *Infrastructure-as-a-Service* (IaaS), *Platform-as-a-Service* (PaaS) and *Software-as-a-Service* (SaaS) (Boniface et al. 2010) (Goyal 2013). Many companies have followed in the footsteps of the cloud providers and adopted some of these classes for their own internal systems as well.

IaaS is the practice of cloud providers selling their own hardware and resources to run the customers software, usually through virtual machines (Goyal 2013). With IaaS the customers can buy easily scalable and cost-effective hosting without needing to worry about managing the hardware themselves. IaaS is provided by many cloud providers for private companies that need hardware to run their software but have such special needs or context that they do not wish to use premade platforms or software for their applications or systems.

PaaS is the practice where the provider is selling a platform and an environment consisting of different services for the customer, which the customer can use on their own developed system. The provided platform can include services for storage, networking, hosting, development tools and computation (Goyal 2013). The customers can explicitly choose which services they want to use and on what scale and they are designed to be easy to integrate by being available on the cloud by common interfaces.

SaaS is the practice of providing premade applications or software as a service over the internet or distributed environment for the client (Goyal 2013). SaaS is usually used by clients that do not have their own software development team and would rather want to buy an application or access to one for their business case. Some companies might buy their back-end system as SaaS but use it on their own client application, which reduces their required workload for a whole working system. Using SaaS can also decrease the initial resources

and time needed to get an initial version of the application online.

2.3.4 Serverless computing

With the ease of scaling their resources with cloud computation, some companies have opted to overscale their resources because of their fear of being underscaled at some point. This practice goes against the scale-on-the-go idea of cloud computing, since the developers are overscaling and thus holding and paying for unused resources. A new paradigm called serverless computing has been developed to solve this issue by leaving the optimization of resources for the cloud provider, who gives true dynamic scalability and pay-as-you-go services for the customer (Castro et al. November 2019). This makes it easier for customers to develop properly scalable services without needing a lot of knowledge about cloud computing or work for configuring their resources. Serverless computing can be seen as a lighter and simpler version of normal IaaS cloud computing, where the customers resource usage is automatically configured by the cloud provider.

Castro et al. (November 2019) define serverless computing as follows: "Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running". The two key aspects of serverless computing found from this statement are that billing is done only per code runs and usage is automatically scaled according to usage. Because of this automatic scaling from 0 upwards, the software needs to work when starting up from zero. This means that some level of statelessness is required from the software. This has evolved serverless computing to be used for running simple programs and functions, which has earned a new name of Function-as-a-Service (FaaS). Castro et al. (November 2019) define FaaS as being a serverless computing platform where the given code or function is executed on triggers activated by events or requests.

2.3.5 Event-driven architecture

Event-driven architecture (EDA) is a way of asynchronously messaging between services in microservices or service-oriented architecture systems with minimal dependencies and

loose coupling between the services (AWS 2023f). In EDA events are meant to trigger state changes or updates on the target system in a way where the systems services do not communicate directly to each other but through a shared independent middleware. Sliwa (2003) describes that events are supposed to be records of different actions that happened in the system. Events should also carry information about the actions, like a new state, any data to save or an identifier of the specified action occurring, so that other services that need to react to those actions can notice them and react correctly.

In event-drive architecture there are three different roles: an event publisher, an event consumer and an event router (AWS 2023f). Event publishers are any services or software that are allowed to send events to the system that others can react to. Event consumers are parties that subscribe or listen to specific or any events and usually react in a given way when noticing their tracked event happening. Event routers are the medium for delivering the system's events from the publishers to the correct subscribers, for example message queues or event bridges.

In a process workflow a publisher is triggered by an action, such as user input, to communicate to other services of the system that a specific action occurred. To communicate this, it publishes an event that matches the action to the event router, which indicates that the given action occurred in some part of the system. The event router receives the event message and checks which parts of the system are subscribed to that specific type of event. Once it finds the correct subscribers, it sends the event to them and they consume it and react according to their own logic to complete the workflow. This publish-subscribe approach allows simultaneous one-to-many messaging and the rule-based event matching allows the subscriber to listen to a large number of sources at the same time (Sliwa 2003)

2.4 Akamon Dataplatform

The target system for the framework developed in this thesis, Akamon Dataplatform, is a distributed information system based on a service-oriented architecture and used by many different applications from Akamon Innovations (Akamon Innovations Oy 2024). It uses an event-driven architecture for messaging between its loosely coupled services. A picture of

Dataplatform's architecture is presented on Figure 1. This section explains the architecture and current monitoring capabilities of Dataplatform.

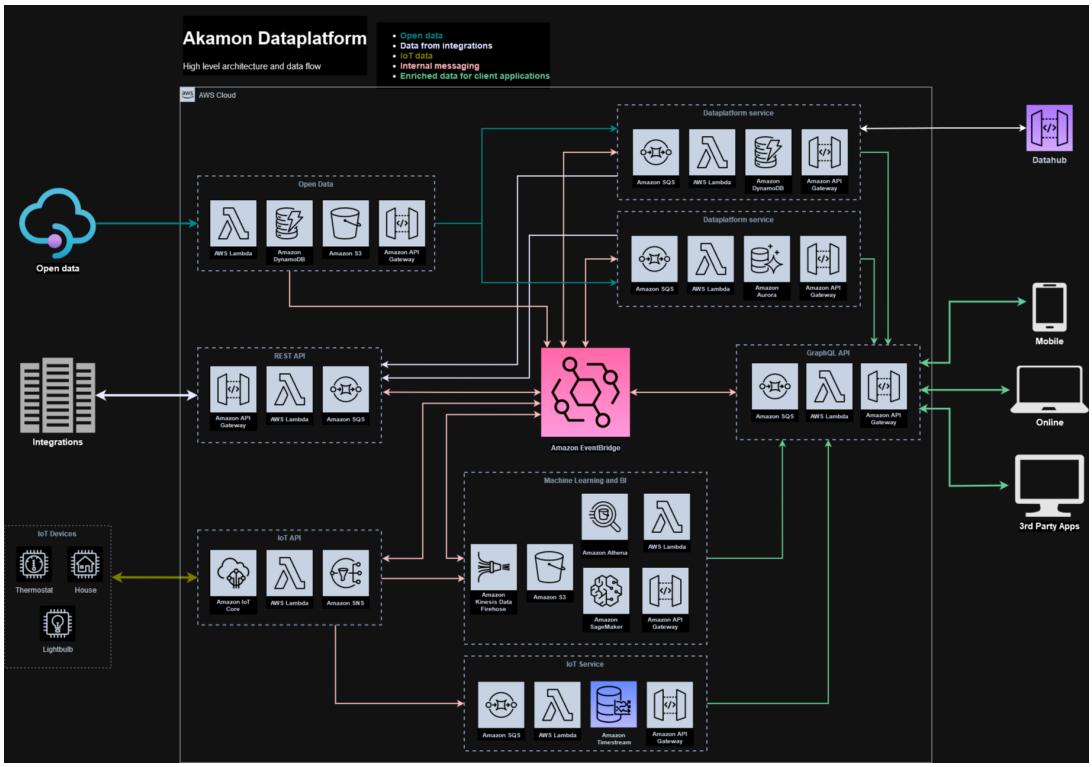

Figure 1. Akamon Dataplatform architecture.

2.4.1 Dataplatform architecture

Akamon Dataplatform is a distributed information system that is constructed of several independent services that are run on the AWS cloud in a cloud computing style. On top of using AWS for hosting the entire system in an IaaS style, many of Dataplatforms services utilize different AWS services for their functionality in a PaaS way. Dataplatform also uses the AWS Cloud Development Kit (CDK) framework, which makes managing and deploying AWS resources much easier by defining the applications cloud infrastructure as code (AWS 2023a).

Dataplatform deploys an event-driven architecture in its service messaging with the AWS EventBridge, which supports an event-driven communication model with subscribe-publish relations. The events are received with the help of AWS Simple Queue Service (SQS),

which allows the events to be stored in a queue before handling to ensure that no events are dropped. With event-driven messaging, Dataplatforms services become very loosely coupled and independent since they do not need to rely on the receiving service to handle their message and respond to it correctly. They can simply send their outgoing message to the AWS EventBridge and forget about it and it is the consuming services responsibility to handle the event correctly after receiving it from the EventBridge event bus. EventBridge also allows multiple services to consume singular events, which makes the architecture and workflows more dynamic.

Dataplatform includes lots of serverless computing inside its services in the form of event-handlers that are subscribed to specific events from the EventBridge event bus. When one of Dataplatforms services publishes a specific event, the subscribing service consumes the event and triggers the correct handler to perform according to its specific functionality. Consuming the event might include running lambda functions, performing data manipulations, using AWS services, for example sending emails with AWS SimpleEmailService (SES), or anything that the specific service has been configured to do depending on its purpose or business case.

Many of the services within Dataplatform also expose their functionality through REST APIs. These APIs allow different applications from Akamon to access and manage these services and the Dataplatform system. REST APIs also allow authenticated third parties to access Dataplatform if they wish to integrate their own systems to work with it. The REST APIs support many different endpoints to allow different operations for the services. The REST APIs can also publish events on certain actions to the Dataplatform if they need to activate certain actions in the system.

2.4.2 Dataplatform monitoring

Akamon Innovations' Dataplatform does not yet have a proper dedicated monitoring platform or framework to use for monitoring the system and its different services. Because of the nature of a distributed system being made out of independent services, it is difficult to have unified monitoring tools that encompass the entire system and all its services. AWS

offers some solutions like the AWS CloudWatch service which allows resource specific monitoring, or the AWS CodePipeline, which shows a graphic and textual representation of the CI/CD process of any deployed applications.

Dataplatform does contain some default monitoring provided by using AWS services and infrastructure. By deploying AWS infrastructure like serverless lambdas or API gateways, AWS automatically attaches an AWS CloudWatch log group for those resources, where that individual process' logs can be monitored. This is currently used for all types of monitoring in Dataplatform, like debugging, error monitoring and general operation monitoring. The logs in a resource's log group usually include information about the invocation of the resource like its start and end times, any runtime logs from the resource like default console logging and any other default information from different AWS services.

Dataplatform also includes monitoring of the service deployment process through the AWS CodePipeline. Dataplatform's services can have very different architectures, but they all include being deployed through the AWS CodePipeline. This pipeline exposes a default monitoring view of each service's deployment process, its different steps and their statuses. Individual steps can also be further monitored with the help of AWS CloudFormation service, which includes lots of information about the deployment steps and their events and operations, which allows the entire deployment process to be monitored very thoroughly.

3 Monitoring

This chapter focuses on learning about monitoring and its usage in distributed systems. Section 3.1 studies what software monitoring is and how it is done in distributed systems. After that different techniques for monitoring are explored as well as their strengths and weaknesses in Section 3.2. Then Section 3.3 lists how monitoring information should be displayed. Finally Section 3.4 explores some existing technologies and tools for monitoring cloud based distributed systems. This research should help to find or design a framework for monitoring cloud based distributed information systems like the target system Akamon Dataplatform.

3.1 Monitoring in distributed systems

Monitoring software systems is the act of following the target systems state, actions and possible errors and responding to them. Monitoring is usually displayed as either textual with software logs or graphical by monitoring dashboards and components or as a combination of them both. Software monitoring is used for many things, including profiling, performance analysis, software optimization, fault-detection, diagnosis, recovery and debugging (Delgado, Gates, and Roach 2004). Its goal is to determine if the system or software is working as intended, and if it is not, monitoring should help in finding the reason why not or why it is behaving the way it is.

There are some differences when comparing monitoring in a sequential program versus a distributed one. The independence of services, asynchronous execution of tasks and messaging between services cause many problems for monitoring. These include having many foci-of-control, communication delays in messaging, nondeterministic nature, monitoring altering system behaviour and more difficult developer and system interaction (Joyce et al. March 1987). Tsai, Fang, and Chen (1990) also emphasize the difficulties of multiple asynchronous processes, critical timing constraints and bigger communication delays in distributed systems. One common problem between distributed and sequential systems is the need to make large amounts of monitoring data readable and understandable for the user.

Joyce et al. (March 1987) found many important characteristics about monitoring distributed systems. First of all, collecting monitoring information should be done with the same inter-process communication (IPC) facility as the services use for communication, since it allows simpler design, debugging and maintenance for the monitoring system, as well as being simpler to port in, since it is already used in the target system. Also, the detection and collection of monitoring data should be separate from its analysis and display, which allows better separation of functionalities and less coercion, which in turn allows both parts to be developed independently without dependencies to each other. Monitoring should also allow better understanding of the system's nondeterministic functionality and help with recreating event flows for testing and troubleshooting.

Telang (2023) also list a few best practices for monitoring distributed systems. Firstly using a log management tool is encouraged, as it makes collecting and managing logging data easier. Secondly setting up alerts is mentioned, as they are a good method to quickly receive knowledge of an independent service malfunctioning or certain events triggering, rather than have them happen silently and only knowing about them after the users notice the problems in the application. The third practice is using log aggregation to help collect data from multiple different servers to one place if the target system is distributed, which gives expanded visibility and reachability by showing the status of every system and service within the organization in one place and spotting issues very quickly. Newman (2023) also mentions autodiscovery to being beneficial in software system monitoring, which makes system scaling easier, provides better reliability through noticing errors and problems early and staying up to date on the system's performance and resource management.

For adding monitoring to a distributed system Telang (2023) give three steps to follow. The first is to define the key performance indicators (KPI), or metrics for measuring performance, that you wish to monitor in your system. For the KPIs it is required to decide which metrics the KPIs should track like errors or service downtime and over what time. Next the monitoring infrastructure needs to be selected as well as any monitoring tools that the system might require and these should then be deployed to interact with the target system. The final step is to collect and analyze the monitoring data for the goals set for the monitoring. This can be done by following the monitored metrics, responding to the monitoring infrastructures

alarms or other performance indicators or viewing the monitoring data on a dashboard or other components.

3.2 Different types of monitoring

A distributed system's monitoring can be split based on the target or level of the monitoring, the data that it monitors or the goals of the monitoring. A system's monitoring can be done on different parts or levels of the system, including service, system, infrastructure and application level monitoring (Telang 2023) (Newman 2023). Monitoring can also be classified based on its goals and target datapoints, including the system's availability, performance, security, errors and resource usage (Kufel 2016) (Harness 2023).

Telang (2023) splits monitoring distributed systems in two types, application-level monitoring and infrastructure-level monitoring. Application-level monitoring can be used to detect slow response times or errors in the application. Infrastructure-level monitoring can be used on either hardware or software, depending on the type of distributed system in question. Monitoring the infrastructure's hardware can be done by monitoring that the physical components, such as servers and storage, are working properly. Monitoring the infrastructure's software is used to check that the application is available, running and working as intended and it can include monitoring databases, web servers, cloud services and other software parts.

Newman (2023) also splits monitoring software into two categories, systems and services monitoring. They consider system monitoring to include monitoring servers, virtual machines, containers, devices or cloud deployments. Service monitoring they consider to include anything that runs on the target system. This can include monitoring single applications or software, cloud services, databases, processes, APIs or network connections and messaging. Newman (2023) explains that while service level monitoring is focused on individual parts of the system, the system level monitoring is used to give a top-level view of the entire system or a unified view of all the parts of the system together.

Hariri and Mutlu (1991) split availability monitoring in distributed systems to include two different levels, user and component level. User level is used to determine the availability of the systems tasks based on the availability of the system's components. Component level

is used to measure the availability of individual components or services. Harness (2023) define availability monitoring to measure how well the system is accessible, operational and responding to user requests. A clear standard has not been defined for implementing availability monitoring, but many companies either do it by designing and providing abstract stochastic models while others simply rely on logging any failures that affect the systems availability (Haberkorn and Trivedi 2007). Kufel (2016) explain availability to be calculable as a percentage by comparing the time when the system was available and responsive to the whole tracked duration.

Performance monitoring is used to measure the applications throughput with metrics about network usage like its response time or bandwidth usage, CPU and memory utilization and storage usage. It can be used to identify slow database operations, network latencies or CPU usage spikes (Harness 2023) (Kufel 2016). Kufel (2016) explain that monitoring a distributed system's performance can help with identifying overutilized and underutilized systems and services that need to be fixed with better resource investment or sharing, redesigning architectures or application decommissions. Performance monitoring can be used to forecast future demands for computing power to allow it to be allocated early before system shortages occur.

Error monitoring is used to detect and raise alarms on errors, such as code run exceptions, errors and crashes (Harness 2023). It usually also includes relevant information to be gathered about the error to make solving it easier. Error monitoring can not only be used for noticing errors, but also testing individual components of the system and measuring the system's reliability (Zulkernine and Seviora 2002). Noticing errors and failures as early as possible also gives value to the system by allowing the errors to be communicated to any customers or relevant third parties and also to be fixed as fast as possible so that they would not keep repeating and causing issues.

Resource monitoring is used for tracking software resources like CPU usage, memory and disk space, to prevent resource-intensive processes, memory leaks and disk space shortages which could be costly or break the systems or one of its services (Harness 2023). Resource monitoring is especially important in cloud systems where running out of resources is uncommon because of automatic resource allocation, but if the used resources are subscribed

to with a pay-per-use model then using any excess resources will incur extra costs.

Security monitoring is done to detect and alert on security vulnerabilities like malware, phishing, unauthorized access or loose system role restrictions (Harness 2023). Security monitoring can also be used to store and analyze security events such as DDoS attacks, brute forced access with multiple login attempts from a single source or unauthorized access attempts from non-admin users (Newman 2023). Security monitoring can increase the integrity, confidentiality and availability of the system and its services and it might also be required by the industry's regulations like GDPR.

3.3 Displaying monitoring information

There are also a few different ways to display the monitoring data for the user or developer, the most common being textual or graphical. Joyce et al. (March 1987) found that both graphical and textual monitoring views should be used, since they offer more complete views of the system together. They found that graphical displays would be best for overall system status monitoring, as they can represent the entire system and its parts and their metrics in different types of diagrams and graphs. These graphics are usually done in a way which allows the user to see the system's status easily with a glance, for example by separating the system's services or different parts in their own boxes and coloring them red or green based on their status.

Joyce et al. (March 1987) explained that textual displays are better for lower-level error and functionality monitoring and debugging. With textual displays an error or code run can be monitored as a list of logs, which are ordered by code execution, which creates a timeline of the run. This same idea can be familiar from stack traces, which are a very common way to represent steps in function workflow runs. A textual trace is considered better for errors because it is easier to follow exactly what happened in the run, in what order and what led to the error, whereas a graphical view of the same subject could not easily make that information as easy to understand while providing detailed information about what happened.

3.4 Monitoring tools and existing technologies

Many monitoring tools and applications already exist that are meant for monitoring distributed systems. Several cloud providers like AWS and Microsoft Azure offer some logging tools and services which are meant to be used together with their cloud resources, services and infrastructure. Several private companies or open source projects also offer many different external tools and services for monitoring your applications or software. These tools are usually designed to be easy to add to the target system without affecting its functionality. In this section we will explore some of the tools and technologies that AWS offers for monitoring, since the target system in this thesis is based on AWS infrastructure, services and resources.

3.4.1 CloudWatch

CloudWatch is one of AWS's most used monitoring tools and it allows observing and monitoring the target resources, applications and infrastructure on AWS, on premise or on other clouds (AWS 2023c). CloudWatch collects real-time logs, metrics and event data from services and infrastructure and enables that data to be visualized and analyzed. This data can be used to monitor the target system's health, performance, operations or networks. CloudWatch also enables setting alarms with actions for different metrics that are monitored, allowing the system to react to anomalies or changes. CloudWatch includes many different features meant for different services or resources, aiming to enable complete monitoring and transparency for all kinds of systems.

CloudWatch also provides developers with resources to create dashboards for displaying their monitoring information. These dashboards are highly dynamic and can be customized with some predesigned components, or widgets, to display the monitoring data. Developers can also create their own widgets with lambda code and HTML to fit just right for their monitoring needs. CloudWatch also includes some automatically generated dashboards meant for monitoring the system's AWS resources split by their parent service. For example for the target systems DynamoDB instances a dashboard is generated which includes information like the number of errors generated by DynamoDB, the amounts of consumed read and

write capacity on DynamoDB tables and the amount of successful and failed requests to DynamoDB.

3.4.2 X-Ray

X-Ray is another service that AWS offers for monitoring cloud infrastructure and resources. X-Ray enables requests to be traced as they operate inside the target system, allowing a transparent view of the system's actions and workflows across services and applications (AWS 2023e). X-Ray gathers data from traced requests and responses, AWS resources, microservices, databases and APIs with information about what happened, when and what the outcome was. This data can then be used to analyze, optimize and find issues such as long delays, overloading or security risks. X-Ray's trace data can also be used to create trace maps, which display all of the resources or applications that the traced request went through, for example any front-end UIs, back-end REST APIs and databases. This way trace maps create a visual representation of the target system's workflows, which can be used to find bottlenecks, latencies, redundancies or other issues.

3.4.3 CloudTrail

CloudTrail is another service provided by AWS for tracking what is happening in the target cloud system, whether it is running on premise, on AWS or another cloud provider. CloudTrail is used for recording and monitoring user activity and service calls as events, providing information about who performed what actions, where they did them and when were they done (AWS 2023b). CloudTrail mainly records three types of events: management events, data events and insight events. Management events include information about resource management such as who created, deleted or accessed what resources and when. Data events capture information about actions performed on data such as reading a database. Insight events are used to bring attention to irregular activities which the admins should look in to, like unusually high error rates or unauthorized accesses. CloudTrail also includes a CloudTrail Lake which is a data lake meant for storing, accessing and analyzing CloudTrails event data, which can be used for security audits and operational issue troubleshooting.

3.4.4 MyApplications

AWS MyApplications is a management tool in the AWS Console meant for monitoring and managing AWS applications costs, health, security and performance (AWS 2023d). MyApplications provides a dashboard which includes separate widgets that display information about important application metrics such as service and resource costs and usage, computation statistics about running instances, load balancing, alarms and their statuses, performance metrics, security risks and much more. MyApplications dashboard is highly dynamic since it is generated with Infrastructure-as-Code (IaC), which allows it to also be configured and modified by the developers to include information they wish to monitor. MyApplications works as a system level dashboard providing information about the developers entire AWS infrastructure on a single dashboard.

4 Research objectives

This chapter explains the research objectives of this thesis. Firstly the research problem is formulated in Section 4.1 and the research strategy and methodology are explained in Section 4.2. Next in Section 4.3 the design process for the monitoring framework is planned and discussed. Finally the evaluation criteria for the developed solution is created in Section 4.4 that will be used to evaluate how well the developed solution works and fits the given needs.

4.1 Research problem

The subject of this thesis is to design and develop a framework for monitoring errors in a distributed system and its services. This framework will be applied to the Akamon Dataplatform-system and later evaluated on how well it accomplishes its goals. Akamon gave the following need statement for the monitoring framework, translated to English from Finnish:

> "We should have system monitoring at such a level that we would get an alert if any part of Dataplatform fails. In it's current state, for example, if writing measurement data to the database doesn't work, we will only know about this when a customer reports that the measurement data has not been updated or by chance we notice the error messages in the logs. It is important to note that malfunctions do not necessarily only apply to updating data, but errors can also happen in our data search interfaces."

Starting with this need statement the problem statement is constructed as follows. The wanted functionality from the framework is to receive an alert if any part of the Dataplatform system fails. An alert and the mention of failure means that the monitoring should be scoped to only include reacting to failures or errors. The statement also mentions receiving these alerts from any part of Dataplatform, which means that the framework should be able to be applied to any of Dataplatform's services. It is also explained that currently there is no functionality for receiving alerts of errors in the different services of Dataplatform. This means that this framework will serve as a proof of concept and an initial version of error monitoring

in Dataplatform, instead of improving existing monitoring techniques or resources. Finally some cases are mentioned where errors can happen, which means that the framework should account for many different types of errors in different resources instead of focusing on a specific error.

From these notes the following problem statements can be built:

PS1 "How would we monitor a distributed system on service level so that alarms are activated on errors or failures in any of the system's services and information about the errors is provided to help solve them"

PS2 "How could we enable notifications for the systems alarms so that developers would be notified as soon as possible when an alarm activates to enable them to address the problem quickly"

4.2 Research strategy

Design science was chosen as the research method for this thesis, since the objective is to develop and test a framework for error monitoring in a distributed system. Design science is used to design and develop an artefact in development cycles (Wieringa 2014). The development cycles include two steps which are iterated over until the developed artifact satisfies the given requirements. These steps include developing the artefact according to a development plan and evaluating the developed artifact with the given requirements.

Hevner A. R. (2004) list different research guidelines for design science in information systems which will be considered in this research. These include the following:

- **Design as an Artifact**: Design science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
- **Problem Relevance**: The objective of design science research is to develop technology-based solutions to important and relevant business problems
- **Design evaluation**: The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
- **Research contributions**: Effective design science research must provide clear and

verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.

- **Research rigor**: Design science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
- **Design as a search process**: The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
- **Communication of research**: Design science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

The goal of this thesis is to develop a framework for monitoring errors in a distributed system's service and evaluate it by applying it to one of the target system's services. This should result in developing a model and method for developing the previously mentioned monitoring per the *Design as an Artifact* guideline. Also applying the framework to the target systems service can be seen as an instantiation. The *Problem relevance* was mentioned earlier to be the difficulty of locating errors in a distributed system caused by the independence and fault isolation of the system's parts or services. The *Problem relevance* is also evident from Akamon's need statement given in Section 4.1. The *Design evaluation* of the framework will be done in Chapter 6 with the evaluation criteria presented in Section 4.4 after applying the created framework to the target *Timeseries* service.

The result of this thesis should be qualified as a *Research contribution*, since the produced design artifact can be used for monitoring distributed systems which organizations could apply for their own systems. *Research rigor* should be evident from the design, application and evaluation of the framework, where the design is created with industry standards and qualified tools from AWS and the evaluation criteria come from the target system's operational environment and its requirements. The guideline *Design as a search process* should be fulfilled by the earlier study of the problem environment and existing monitoring tools, as well as applying an iterative design process as explained in the next section. This thesis and its content should also be constructed in a way that any audiences can understand its concepts and results which should satisfy the *Communication of research* guideline.

4.3 Design process

The chosen design process consists of 6 steps which are *problem identification and motivation*, *objectives of a solution*, *design and development*, *demonstration*, *evaluation* and *communication*, as explained in the process model for design science in information systems by Peffers et al. (2006). Based on this model a new version of the design process was created for this thesis, which is displayed in Figure 2. This process will be followed during this research by first *identifying, explaining and motivating* the research problems in Section 4.1. The *objectives of the solution* are then explained through the created problem statements. In Chapter 5 the artefact will be *designed and developed*, as well as *demonstrated* by applying it to the target system. In Chapter 6 the framework and its application to the target system will be *evaluated* with the evaluation criteria 4.4. If the evaluation criteria is not fulfilled, a new development cycle will be started by moving to the *design and development* step to further develop the artefact in the correct direction, as explained in Section 4.2. Finally once the framework and its implementation fit the evaluation criteria, the last step will be fulfilled by *publishing* the research as a masters thesis.

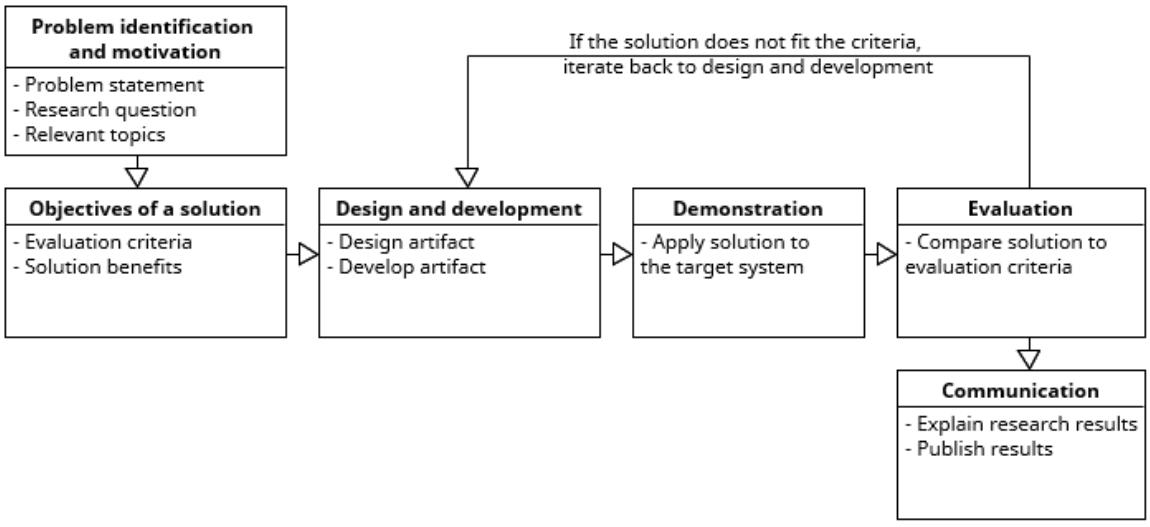

Figure 2. Design process for developing the framework.

4.4 Evaluation criteria

The success and level of functionality of the created framework will be evaluated with the requirements given by Akamon. Individual evaluation criteria can be separated to a list from the given requirements in the following way:

EC1 The framework is able to detect runtime errors or failures.

EC2 The framework is able to activate alarms on errors or failures.

EC3 The alarms should be able to be subscribed to so that they would send notifications to the subscribers when activating.

EC4 The frameworks alarms can be displayed in a dashboard.

EC5 The frameworks alarms contain enough information on the error to help find and solve it in the right part of the system.

EC6 The framework can be applied to any service within Dataplatform with minimal modifications.

By evaluating the developed framework with these criteria, it can be measured how well the framework works and what parts of it should be improved. The previous evaluation criteria were also approved by Akamon Innovations to fit the given need statement and the goals of the monitoring framework.

To measure the framework against the evaluation criteria, the framework will be applied to the *Timeseries* service of Dataplatform. In the evaluation phase of the design science development cycle, the framework will be applied to the chosen *Timeseries* service and analyzed to see how well it works and fits the given requirements. If after the analysis there are still some requirements that are not yet fulfilled, the development will be continued on those parts of the framework. This development cycle is repeated until all the evaluation criteria are fulfilled.

5 Design and implementation

This chapter goes over designing and developing the framework for monitoring errors in a distributed system and its services. Section 5.1 explains the design of the framework and its monitoring components. The framework is then developed and applied to the chosen *Time-series* service in the Akamon Dataplatform system in Section 5.2 which will also later be used to evaluate the framework in Chapter 6. Since the target system is based on AWS infrastructure and services, the application of this framework will also be heavily AWS based.

5.1 Framework overview

Firstly all the different aspects that the framework needs to implement per the research question and the framework's criteria are gone through. Since the goal of the framework is to monitor for errors, the monitoring targets need to be chosen to include any possible causes of errors in the target service. Then alarms need to be created for these errors, which should activate when the error metrics cross a given threshold of error counts. The alarms should also enable automatic actions to be made when the alarm goes into an active in-alarm state. These actions should include displaying the errors in dashboards and sending notifications of the errors to the developers or other appropriate parties.

5.1.1 Monitoring targets

The first step in creating monitoring to a given service is to design the KPI to monitor in that service (Telang 2023). Per the research question, the goal of this framework is to monitor for different errors in the individual services of the target distributed system. These services can include many different technologies, resources and external services that should be considered to be monitored to make sure the entire service is included in the monitoring and all the services possible errors could be caught. The targets can include serverless lambda functions, databases, REST APIs and any other software or resources that the target service and system use for its functionalities. The targets for the monitoring and their error metrics should be considered case-by-case depending on the service and the resource to be monitored

and the goals of the monitoring.

Since the target service for error monitoring can include so many different technologies and resources with all of them having many different types of errors, it is important that the monitoring design considers as many of the different types of errors as possible. While designing error monitoring for a system or service, it should be considered which of all the possible errors should be monitored for since all of them might not be needed. Including all the possible error types of the target application in the monitoring can increase the cost of the monitoring resources unnecessarily and increase the system's, the monitoring resource's and the infrastructure's complexity. This can lead to making the monitoring dashboards cluttered and hard to analyze, flooding the notification channels and making the monitoring infrastructure very hard to maintain.

On top of the service's resources, runtime logic errors or bad functionality in code could also be considered to be included in the error monitoring. These types of errors are usually handled in the code for example by logging an error message or throwing an exception. For catching error log messages, the computation log stream should be a target for the monitoring. It should be noted that for noticing these runtime logic or business errors, the correct handling for these error cases should be well designed and implemented in the service when adding the log error messages to be monitored. This means that for example only actual errors should be logged as error messages, rather than logging non error cases as error messages. Since it is harder to notice from the individual messages whether the message is caused by an actual error and not a normal logged message, error logging should be properly separated from the normal log messages in some way which the monitoring tools can use to separate their targets, for example a log level.

5.1.2 Raising alarms

After an error threshold is breached in the target service, the monitoring framework should be able to notice it, react to it and act accordingly instead of passing it and continuing normally. This means that alarms need to be created and configured for the errors that are being monitored. These alarms should be tied to follow the chosen monitoring targets or KPIs and

activate and change their alarm status to an activated "IN ALARM" state when the alarm's configured error threshold is exceeded. These alarms should be configurable to allow them to be only activated by the errors that they are meant for and to be activated by different types of errors, since there can exist so many in the target service.

Another important part of activating an alarm on noticing an error is the ability to also act on it. Depending on the type of error and resource or service that it appeared on, as well as the contextual business logic and functionality, the actions to perform can vary a lot. The alarms should be able to perform any types of activities on noticing an alarm to enable the error monitoring to be more dynamic and recover correctly from the errors or perform further actions to increase the system's fault tolerance. The actions to perform can also be used to develop the error monitoring further by increasing the information gathered from the errors, performing different actions like enriching the error data with a stack trace, sending notifications about the noticed errors to outside the system or any other actions that the developers wish to add to the alarms to handle their errors correctly.

An important aspect to also consider is how often the alarm should activate when noticing errors. In a software system some functions might cause single errors very rarely or a very large amount in a short time frame, depending on how often they are run and what they do. For example if the system performs batch data update operations on a large amount of data points and a high number of them result in an error, it would cause the alarm to also be activated a high number of times for basically the same problem. Especially when using actions on alarms, activating the alarm on every error might cause an unnecessarily high number of alarm activations and actions to be run, which could for example lead to flooding the alarm notification channel.

This problem could be solved for example by setting a higher threshold for the number of errors in a given time frame for those resources where the normal number of errors is high. On the other hand, some functions or resources might be wanted to activate an alarm on a single error, for example in critical operations which should never fail. From this analysis it is clear that the alarm activation threshold, error criticality level and evaluation period length for each alarm should be considered case-by-case depending on the monitored resource, its context and functionality.

5.1.3 Sending notifications

When an error occurs in the target service, it is very important to notice it and act on it as fast as possible to enable it to be fixed as early as possible. This way the error would cause minimal damage for the system and its users and organizations. With the ability to activate alarms and perform actions on them, the damage that the error causes for the system can be reduced and solving the error can be made easier for the developers by providing important data about what activated the alarm. The error would still need to be investigated and fixed by the developers so that it will not keep happening and causing issues. For this goal, it is necessary that the developers gain knowledge of the errors and when and where they happened. This was also one of the requirements for the error monitoring framework.

For this, notifications would need to be sent to the developers through a messaging channel like a private issue board, a messaging platform, an email or with an SMS message. Depending on the developers' choice of communication channel, the notifications should also be configurable to fit different needs or monitoring goals of the organization. These configurations could include selecting the recipients of the notifications, like a common issue board or an email list, which developers can subscribe and unsubscribe from. It is also important to note that notifications from different alarms might want to be received by different developers but not everyone, for example a team working on a front-end application would not probably need to receive notifications of an alarm activating for back-end database read operation errors. The notification feature should take this alarm specific subscription point to consideration if the chosen notification channel is done with individual messaging rather than a generic common notification board, where all notifications are gathered to common shared platforms.

It should also be considered how often a single alarm should send notifications. If a notification were to be sent each time an alarm goes into an active state, it could easily flood the error notification channel if there is an alarm which goes into alarm state many times in a short timespan. This could lead to other notifications being hidden in this flood of alarm notification messages and would make it harder to notice any of the notifications of other alarms. To solve this a cooldown between notifications could be considered, so that the same alarm does not send a big batch of notifications for the same error appearing a large amount

of times in a small time window. This problem should be considered separately from the alarm error threshold and evaluation period problem, since the threshold of errors in a given time frame should be used for activating the alarm and the notification cooldown problem should be used for how often notifications should be sent, since not every alarm activation necessarily needs to send a notification.

5.1.4 Monitoring dashboard

It is important for monitoring software to display both textual and graphical data to achieve a complete view of the entire target system. One way to achieve this is by implementing a dashboard which would allow the monitoring data to be displayed with different kinds of visual and textual components. It is important that the target systems developers carefully choose what kinds of data they wish to view on the dashboard about their target system and design the components to use in the dashboard according to those needs (Janes, Sillitti, and Succi 2013). With error monitoring the data could include service or resource error alarm statuses, metrics of errors noticed in the target service or resource, a log stream of error messages and any further information about the errors that the targets can produce.

For graphical displays of the alarms and errors of the target service the dashboard could include graphs and alarm status components. An alarm status component could be used to display the target services resources alarms current status, which would make it easier to see the status of each part of the service as well as the entire services overall status at a glance. This could be done for example by binding the alarms to a graphical on/off component which shows normally working components with a green light and components in error states as red. Graph components could be used to display the number of errors over a given time period which could be used to see how often a given error happens in the target resource. While the status component could show the current status of each alarm, the graph component could show the history of those alarms to get a better view of how the alarm behaves and errors are produced over time.

To include textual data the dashboard could also either include textual components, for example to display the target resources error logs, or components which would allow the mon-

itoring targets logs to be accessed. This would allow the dashboard to be used to dive deeper into the problems that are observed and help solve them by providing further information about the events that led to the observed errors. A log stream component could also be used to follow the logs of the monitoring targets with filtering for error messages, which would allow errors that are logged but do not activate an alarm to be discovered. These kinds of errors could be logical or other run time errors that are not software related or handled as proper errors that would cause alarms to activate. A log stream component could also be used as a tool to increase the monitoring in the service by discovering new types of errors that were not yet monitored and alarmed for.

Any other dashboard components could also be designed and used based on the developers' goals, users' or customers' requests or the KPIs that are monitored. Also, since the service can include many different monitored resources and technologies, they might be wanted to be displayed differently on the dashboard. Having a flexible platform or method for creating the dashboard and its components would be important to allow as much freedom for the monitoring as possible.

5.2 Implementing the framework

In this section monitoring tools will be designed and developed for monitoring the target services resources error metrics, creating alarms for those metrics and adding actions such as notifications for those alarms. In the design easily reproducible methods are wished to be used to make it easier to apply these same monitoring tools for the target system's other services as per the evaluation criteria **EC6**. AWS tools, resources and services will be used as much as possible since the target system is already heavily AWS based and thus the monitoring could be easier to integrate to the existing platform and have fewer outside dependencies. This would also allow the monitoring to be easily accessible in the AWS Console, where all the other resources and services of an AWS based platform can be accessed.

5.2.1 Metrics and alarms

By default, AWS CloudWatch provides some premade monitoring metrics for their resources and services, which include metrics for different errors that the target can run into. Initially error metrics for only AWS lambdas, REST APIs and DynamoDB tables were searched for since the target Akamon Timeseries service only included these resources. For lambdas there exists a metric for measuring the number of invocation errors, such as timeouts and exceptions. For REST APIs there exist error metrics for tracking different client-side errors such as bad requests and server-side errors such as internal service errors. For DynamoDB tables AWS has also split the error metrics into user and system errors, where user errors include bad requests and requests to update non-existing resources, and system errors include 500 coded errors such as internal service errors. These monitored metrics should be chosen based on what errors the monitoring is supposed to catch and from which resources or services.

CloudWatch also includes Alarm constructs that are meant for following a given metric and reacting to the followed metric's behaviour. These alarms can be configured with a specific threshold of the metric's invocations and an evaluation period's count and length. The threshold of invocations is used to define the number of invocations that need to be breached to cause the alarm to activate. The evaluation periods can be used to set a number of back to back periods that exceeded the metrics threshold to activate the alarm and the length of those periods. The alarm also takes in a comparison operator to be used for determining the alarms activation based on the number of invocations and the invocation threshold, for example by comparing if the number of invocations in the given evaluation period is larger than the set invocation threshold in which case the alarm would be activated. AWS CloudWatch Alarms also allow certain actions to be attached to them which will be invoked when the alarm changes to an activated state. These actions can be used for example to run lambda functions or send default alarm notifications with the AWS Simple Notification Service (SNS).

The premade error metrics were first tested by creating three test functions with AWS lambdas and adding the invocation error metrics and simple CloudWatch Alarms for each of them. The three different lambdas were supposed to cause errors in different ways to see how well the metrics and their alarms worked and what data they provided about the alarm state. The first lambda was configured to throw an error in code, which resulted in the error metric

counting one error and the corresponding alarm being triggered. The second lambda was configured to sleep for 10 seconds while the lambda's timeout value was set to 5 seconds to simulate a lambda timing out. This also caused the error invocation metric to count one error and the correct alarm was activated. The third lambda only had a simple case of logging an error message, which did not cause the invocation error metric to count an error and that metrics alarm did not activate. This meant that catching errors in code such as logic errors would need to be monitored in a different way by analyzing the log stream.

CloudWath also allows creating custom metrics for different datapoints or KPIs. One of these metrics is a MetricFilter construct, which is a metric created for reading and filtering the given resources or services logstream with a specified filter pattern. This MetricFilter construct was used for monitoring the target services resources log streams with a filter pattern that would search for matches of logs which included the text *"Error"*. This meant that any runtime error cases would need to be logged with a message that would include the word *"Error"* and this would cause the MetricFilter to count one invocation. An alarm was also created for this metric like all the other error metrics of the framework, which would allow the same activations and actions to be applied for this metric. The MetricFilter worked on the error logging test function correctly by counting one error invocation when the error message was written on the lambdas log stream and the corresponding alarm was activated.

CloudWatch also includes a construct called composite alarms, which are in essence parent alarms for following the given child alarms states. This means that they can bind multiple alarms into one and by comparing the child alarms states to the given alarm rule they can determine their own alarm state. These kinds of composite alarms were used for all of the resources mentioned before to bind all of the alarms of a single resource under one composite alarm with an alarm rule *"Any Of"*. This meant that if any of the composite alarm's child alarms were activated, the parent composite alarm was also activated. These composite alarms could enable each resource's overall error status to be easily monitored with a single alarm. This could be useful for example in a dashboard with alarm status components so that the components could be created for each resource rather than each resources alarms and notifications could be subscribed for by entire resources rather than needing to subscribe every alarm of a single resource.

5.2.2 Monitoring the target service

When applying the error monitoring framework to the target Timeseries service, all of the services monitoring targets were firstly recognized and the error metrics and their alarms were created for each target. The created monitoring resources also mention dashboard widgets, which are dashboard components meant for viewing the monitoring data. These will be further explained in the next subsection.

For lambda functions the default invocation errors metric was used with the metric calculating the sum of error invocations in one-minute periods. Then an alarm was created for the default invocation error metric with an invocation threshold of one and a comparison operator of *"GREATER THAN OR EQUAL TO THRESHOLD"* which meant that even one error would cause the alarm to activate. The alarm's evaluation period's length was also set to one minute from the default value of 5 minutes. For the lambda functions logs a MetricFilter with a filter pattern of *"Error"* was also created with a corresponding alarm to notice any errors in the logs. This alarm had the same configuration as the one created for the default error invocation metrics alarm. The error metrics, alarms and dashboard widgets that were created for the lambda resources can be seen on a model in Figure 3.

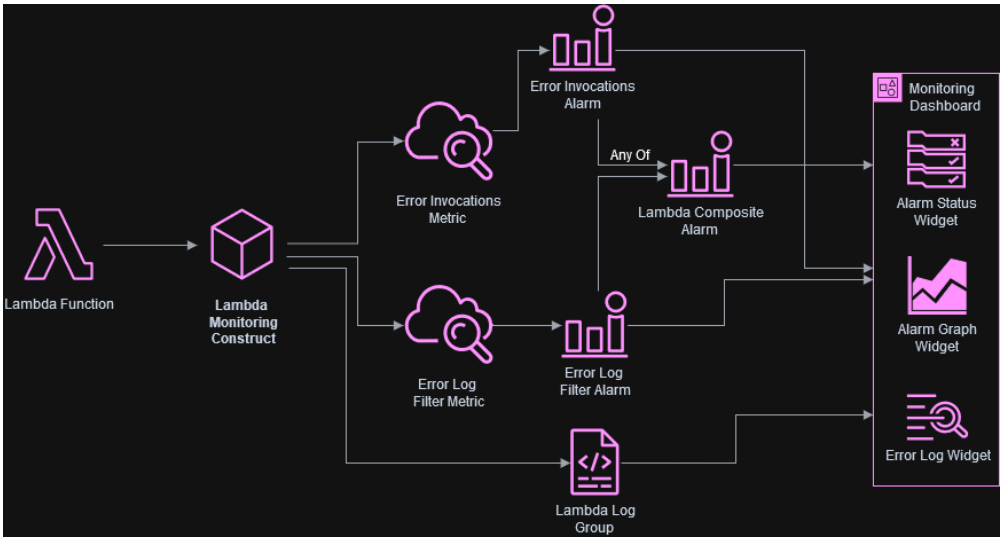

Figure 3. Model for creating error monitoring for an AWS lambda resource.

For monitoring the target services REST APIs both the user error metric and the server error metric were used to achieve maximum error coverage. Since the REST API construct itself

does not include any custom logging but rather only the resources default logs, it was decided that a log MetricFilter was not needed for this resource since any errors would already be caught by the error metrics and filtering in the logs would not help catch any extra errors. For the REST APIs alarms the same configuration was used of having the error threshold at one instance and the evaluation period at one minute to activate the alarms as early as possible on single errors. The REST APIs entire monitoring model can be seen on Figure 4. It includes the created error metrics, their alarms and the dashboard widgets created for the alarms.

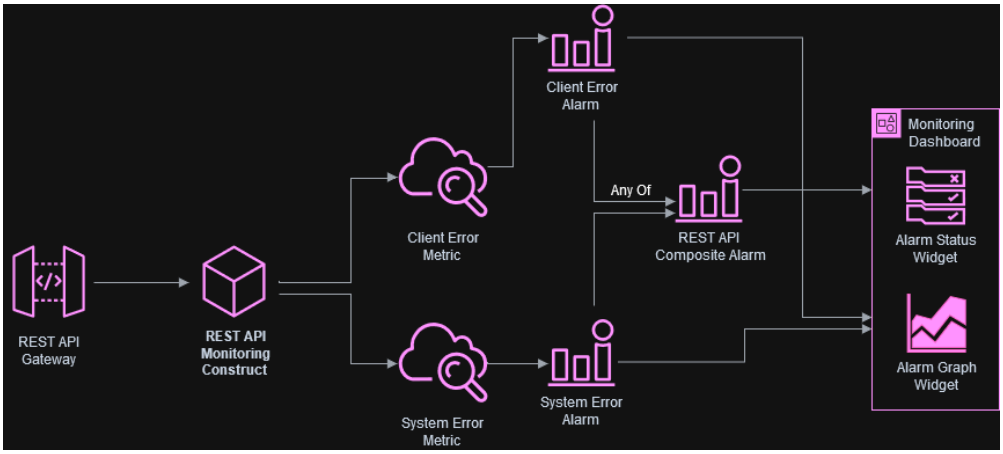

Figure 4. Model for creating error monitoring for an AWS REST API resource.

For monitoring the target services DynamoDB tables it was wanted again to include all the possible errors that the resource could produce, which meant that the default user error metrics and the system error metrics were used. The system errors metric also needed to be configured for specific database operations and one metric had a maximum capacity of 10 operations per metric. This meant that all of the operations could not fit into one metric, so the operations were split into two metrics. The first metric included all of the different read operations and the second metric included all of the create, update and delete operations. This was thought to be the best course since it allowed all of the operations to be monitored for errors while also keeping the amount of metrics and alarms as small as possible. If each operation type had its own metric, then all of them would also need an alarm to be created for them, which would easily clutter the monitoring infrastructure and could increase the monitoring costs unnecessarily. A model for monitoring errors in DynamoDB tables can be seen on Figure 5 with the resourcers error metrics, alarms and dashboard widgets.

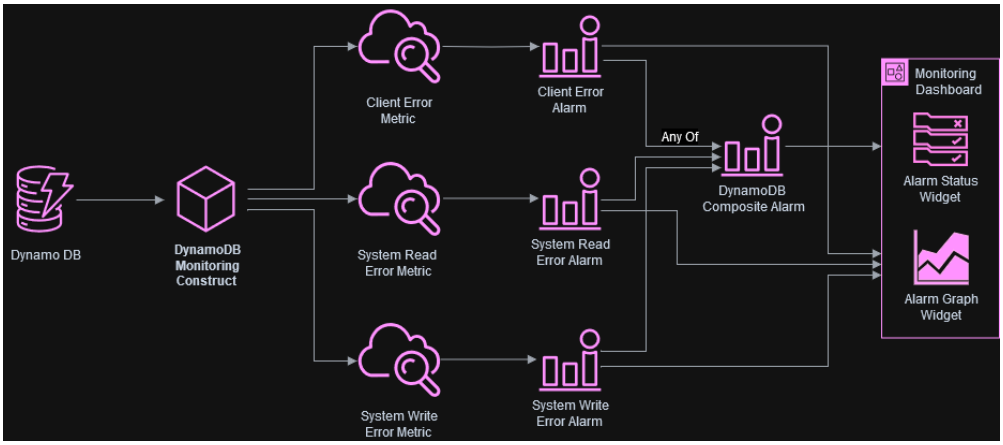

Figure 5. Model for creating error monitoring for an AWS DynamoDB table resource.

5.2.3 Dashboard

For viewing the target service's error metrics and their alarms a dashboard could be used, since it can make important data easily accessible and readable. For this feature AWS CloudWatch includes a Dashboard construct which can be used for creating monitoring dashboards with configurable widgets for displaying the monitoring data. While AWS provides some premade service specific dashboards for monitoring your cloud resources like S3 databases or EC containers, it also allows the developers to completely build their own custom dashboards for their data and monitoring needs.

For the dashboards CloudWatch offers different widgets for displaying the monitoring data such as graphs, gauges, status indicators, charts, different tables and number displays. CloudWatch also allows the developers to add pictures and text to the dashboard with Markdown and create custom widgets for the dashboard with lambda functions that take in the monitoring data and output the custom widget code. For the Timeseries service's error monitoring dashboard the data was focused on error metrics and alarms and their statuses. For displaying this data, the *AlarmStatusWidget*, *LogQueryWidget* and *AlarmWidget* were chosen to be used.

The *AlarmStatusWidget* is used to display a given alarm's status with colors and icons as well as the alarm's name. The widget turns red with an exclamation mark when the target alarm goes to an activated "IN ALARM" state, to green with a check mark icon for an "OK" state

and grey for a "NO DATA" state. This allows the target alarm's state to be determined on a glance value. The widget can also be opened to see information about the alarm such as its description info, the alarms state change reason to see what caused the alarm to activate and a link to the alarm's resource page which includes even more information about the alarm like its state history.

In the Timeseries service's error monitoring dashboard an *AlarmStatusWidget* was created for all of the service's resources composite alarms. By adding the widget only for the composite alarms, it would show the overall alarm state of each of the resources of the service while keeping the number of widgets to a minimum to not clutter the dashboard. The alarm that caused the composite alarm to activate can be seen in the composite alarm's resource page, which can be accessed through the widget. An example of the *AlarmStatusWidgets* on the dashboard can be seen in Figure 6, where a DynamoDB resource "EntityTable" and the lambda function "IloDataUpdateFn" have encountered errors and changed to an "IN ALARM" state. The rest of the resources' alarm statuses are in "OK" state meaning that they have not encountered errors during the last evaluation period.

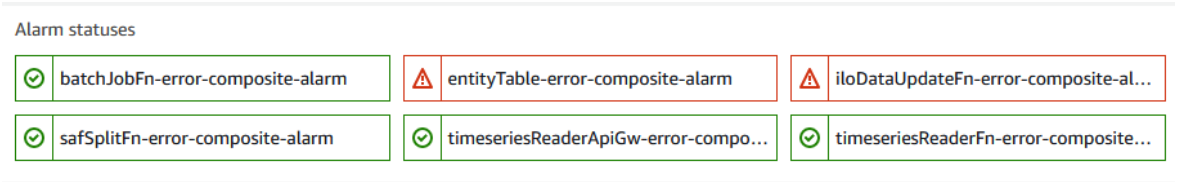

Figure 6. The *AlarmStatusWidget* displaying the status of each composite alarm of all the resources in the target service.

The *LogQueryWidget* is used for scanning and displaying the target logs as a table with a log message on each row. The data displayed of each log as well as selecting which messages are shown is configured with an SQL-like query string, which filters the given logs and then chooses which fields of the logs to show on the table. The log groups given to the *LogQueryWidget* in the Timeseries service's error monitoring were all of the log groups of the Timeseries service's lambdas. The query string used for filtering and selecting shown log message data on the widget was:

```
fields @timestamp, @log, @message, @logStream, level
sort @timestamp desc
filter @message like /Error/ or level like /ERROR/
limit 20
```

The query filter string specifies that the *LogQueryWidget* should display the fields *timestamp*, *log*, *message*, *logStream* and *level* of a single log message on the columns of the widget's table view. The query filter also sorts the logs by their timestamp to be descending so that the latest log is at the top. The query filter string also filters the given logs to only show logs that include the word "Error" in their message or logs which have their log level set to "ERROR". The query filter also limits the logs to 20 to not take up too much space of the dashboard.

The *timestamp* field displays the time when the log was created, the *message* field is the logs message, the *logStream* field includes a reference to the log stream where the log belongs and the *level* field is used by AWS tools to split logs to different levels for different handling. The *logStream* field can be used to navigate to the message in its log stream where other logs of the same run can be seen to figure out the context of what happened before the error message was logged. An example of the widget can be seen in Figure 7, which is split into two parts to fit the page.

The *AlarmWidget* is a widget for displaying the target alarms activation history with the alarm's metric counts and the activation threshold. While the *AlarmStatusWidget* only shows the alarm's current status, the *AlarmWidget* can be used to see the history of the alarm's status activations. The graph can also be used to monitor the alarm's metrics values, for example if the error metrics are consistently near the alarm activation threshold but never cross it, those errors should be studied and fixed even though they are not activating alarms or displayed in an error state in the *AlarmStatusWidget*.

An example of the *AlarmWidget* can be seen in Figure 8. In the example figure it can be seen that there was one caught error at 15:50 UTC time in the "IloDataUpdateFn" lambda's logs and before that there were no errors for about 3 hours. After the one error count there were no errors for 10 minutes, so the alarm changed back to an "OK" state. The error threshold can also be seen at the top as the red line at 1 count, meaning that even one error causes the associated alarm to activate. If the error count would be higher than the threshold, the graph

Figure 7. *LogQueryWidget* displaying error logs of the target services resources.

would show the error count which is also useful to see how many errors are happening in the target during each evaluation period.

5.2.4 Alarm notifications

One of the framework's requirements was the need for sending notifications of alarm activations to given subscribers with information about the activated alarms. The first solution designed for this was to use AWS CloudWatch alarm actions with the AWS Simple Notification Service (SNS), which is a service used for sending messages through different communication channels to the configured receivers. With SNS an alarm activation topic could be created and an action of sending a notification email to that topic could then be added to the created alarms. This SNS topic could be subscribed by all the parties that wish to receive notifications of that alarm activating.

With this solution the alarm notification topics should also be designed to allow individuals to only subscribe for some of the alarms but not all of them, since everyone might not need

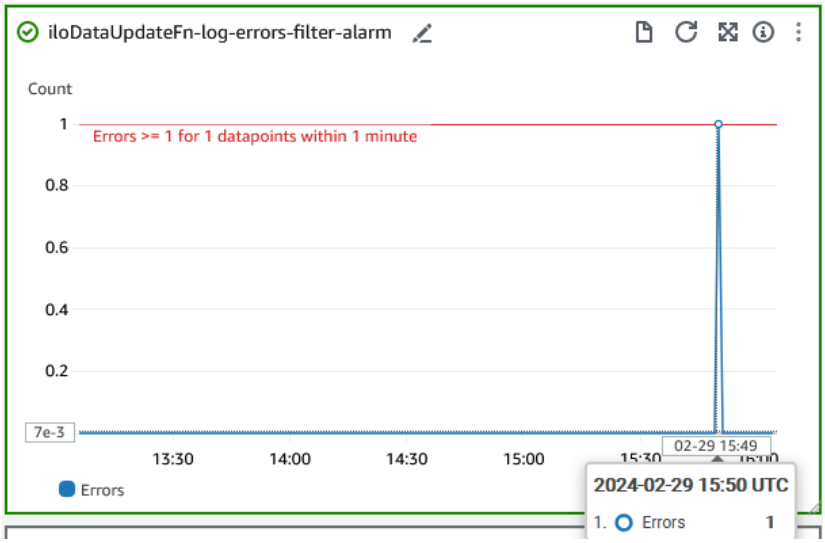

Figure 8. *AlarmWidget* displaying the error metrics, threshold and activations of the target alarm.

to be notified of every alarm's activation. If the chosen notification channel is a single Teams channel which would display notifications of all of the system's or services' alarms, a single SNS topic would be enough and all the alarms would send notifications to that topic. However if the alarm notifications are wanted to be able to be subscribed by resource, individually or some other grouping, a new SNS topic would need to be created for each of them to be subscribed by the correct receivers so others would not be needlessly notified. One thing to note with this approach is that it could lead to a very high number of SNS topics and a lot of complexity with managing subscriptions for each topic.

The default alarm activation notification email from SNS includes information about the activated alarm such as the alarm's name, description, state change data including the previous and current states, the reason of the state change, timestamp of when the alarm was activated, which account the alarm exists on and the alarm's ARN (Amazon Resource Name) which can be used to access the alarm from the AWS Management console or through the CloudWatch API. The notifications also include information about the metric that the alarm is tied to, including the metric's name and namespace, data point or KPI dimensions, evaluation period length, the set evaluation statistic and data unit name. The notification email also includes a link to access the activated alarm in the AWS Management console and another

link for unsubscribing from the notifications SNS topic.

Another way to send alarm activation notification emails would be to use the AWS Simple Email Service (SES). It is a cloud-based email service provider for high value email automation that is easy to integrate to different applications or AWS services and resources. It can be used to send customized emails from an authenticated user to chosen recipients. Using SES for sending alarm activation notifications instead of using the alarm actions with SNS would allow the notification emails content to be more customizable to fit the organization's needs. SES would also allow using email addresses as recipients which would avoid the added complexity of managing many different SNS topics and their subscribers.

It is important to note that SES requires the users to create email identities for sending emails and these identities need to verified by AWS if they are wanted to be used in non-sandbox environments. SES also includes some restrictions for sending emails like a daily send limit and measurements for bounces and complaints. Exceeding the bounce or complaint limits can cause your email identities to be restricted from sending emails for a while. These measurements and limits should be continually considered and monitored when sending alarm notification emails with SES.

Akamons Dataplatform also includes a Message service with the functionality to send emails with external email software which was considered to be used for the alarm notifications, but since the Message service is part of the system to monitor it could also include errors that should be monitored. If an error would occur in the Message service it could lead to the service not sending any alarm activation notifications, which could lead to errors happening with no notifications being sent about them. For this reason, it was thought that directly using an AWS service like SES would be safer since the errors there would be handled by AWS and they would notify their customers and users of services being down, which the developers could react to then by for example monitoring the system for errors another way.

To send notification emails with SES some handler such as a lambda function would need to be created which would handle using SES to send the notifications. When using SES for sending notification emails for activated alarms, the function that would send the notifications would need to trigger reactively when alarms change to active states. This could

be done by either running the notification function as an alarm action similarly to the SNS notifications or the function could be triggered by alarm activation events sent to the AWS EventBridge. By default, CloudWatch publishes alarm activation events for every activated alarm to AWS EventBridge's default event bus, which the notification function would listen to. The default alarm activation event contains information about the activated alarm such as the alarm's name, its tracked metrics, a timestamp of the activation, reason for activation and information about the resource which activated the alarm.

For sending alarm notification emails, the SES email notifications were chosen. They allowed the most customizability for the email's content and future development, and easier subscription management by not needing to create and subscribe individual SNS topics. The trigger for the notification handler was chosen to be the default alarm activation event from the AWS Eventbridge's default event bus rather than using alarm actions, since the alarm activation events would already include lots of useful information about the alarm and its targets without the need to fetch that data from CloudWatch with extra queries. The event triggered notification function would also fit in well with the rest of the Dataplatform system because of its event-based design as well as allow future development so that other sources could also publish events for the notification handler if needed.

The created alarm notification handler includes a simple serverless lambda function which would receive a CloudWatch alarm state change event as input from the system's default event bus and send a notification email with SES with information about the activated alarm to the correct subscribers. In the notification handler's construct code the default event bus was added as an event source so that its events could be caught and a rule was created to filter the events that should be handled by the notification handler. The created rule included checking that the event's source is "aws/cloudwatch", which is the default source for the alarm state change events, the detail type of the event is "CloudWatch Alarm State Change" and the event's details included that the event's current state was "ALARM", so that only alarms changing to an active state would activate the notification handler and not for example alarms changing from an "ALARM" to an "OK" state.

The alarms were chosen to be subscribed individually to allow the most preciseness in receiving notifications. For this the subscriptions for the alarm activation notifications are handled

with a JSON file including a list of subscriptions with each subscription including the target alarm's name and the email addresses which should receive notifications when that alarm activates. After an alarm activates and the alarm's state change event causes the notification handler to run, the subscriptions are scanned to find the email addresses that are subscribed to that alarm's notifications. These email addresses are then added as receivers to the SES email request along with the other data of the email's content. As future development the handling of alarm activation subscriptions could be changed to saving them in a database instead of the JSON file, which would make them easier to manage. The subscriptions' data model could also be changed to include either the target alarm's resource name, service or some other way of grouping if the alarms would be wanted to be subscribed by those instead.

The alarm activation notification email's body is created with the alarm state change event's data, which includes information about the activated alarm and its metrics. The email body is constructed with HTML and it includes the activated alarm's name, its metric's names, namespaces and dimensions, the alarm activation reason, timestamp of the alarm's activation and the region and account that the alarm exists on. Depending on the developing organization's needs this data could also be enriched with any other data they wish to view on their notification emails. These could include for example a link to the activated alarm, its target resource or service in the AWS Management console or to some monitoring interface such as a dashboard. These could be used to access the alarm and its target resource to see more information about what caused the error and could help with fixing the problem faster.

The notification handler's architecture can be seen in Figure 9 and its functionality can be seen on a sequence diagram in Figure 10.

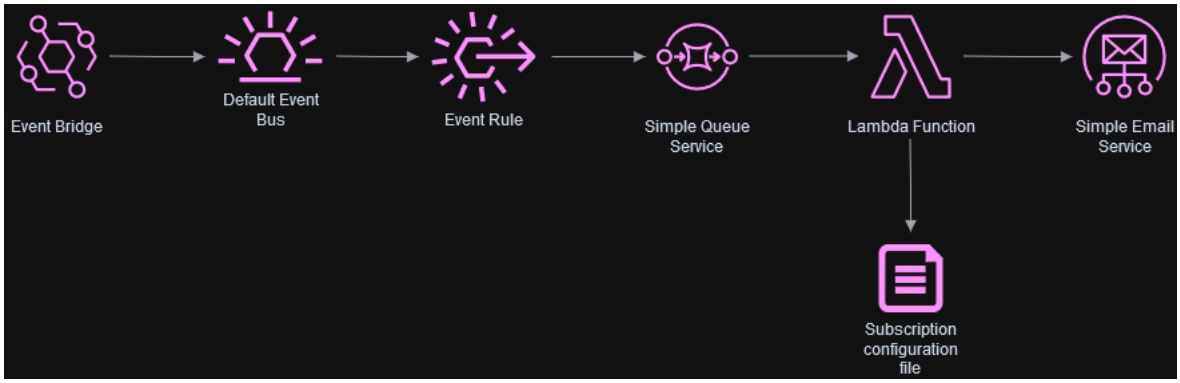

Figure 9. Architecture diagram of the notification handler's infrastructure.

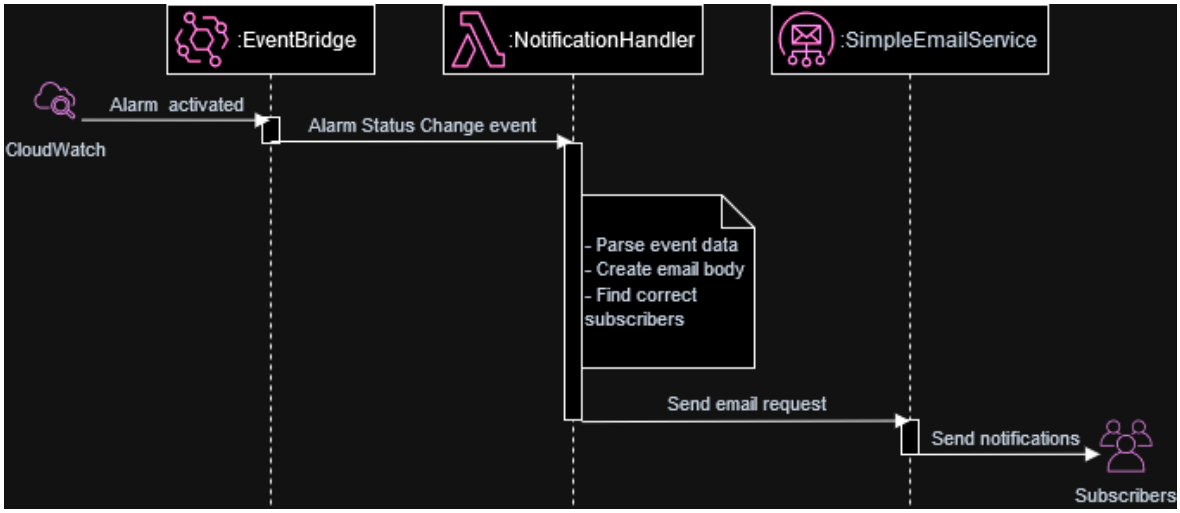

Figure 10. Sequence diagram of the notification handler's functionality.

6 Evaluation of the framework

In this chapter the created framework of error monitoring in a cloud based distributed system is evaluated based on the evaluation criteria mentioned in Section 4.4. Firstly the framework's application to the target Dataplatform system is evaluated in Section 6.1 to determine how well it works and what its results are. Then the framework and its application to the Dataplatform *Timeseries* service are measured against the given requirements and evaluation criteria in Section 6.2 to see how well the framework fits its given purpose and goals. Finally some future development cases are examined in Section 6.3 for improving or extending the frameworks functionality.

6.1 Framework operation

After the monitoring framework was applied to the target *Timeseries* service of Akamon Dataplatform its functionality was tested by causing different errors that should be caught by the error metrics and alarms. The tests were done by sending test events or requests to the target service. The errors were expected to be caught by the measured error metrics in each resource, activate the correct alarms for those metrics, send notifications about the alarm activations and display the error data on the service's dashboard.

6.1.1 Lambda errors and alarms

For testing the invocation error metrics in lambda resources, an empty event was sent to those handlers. This empty event would be qualified as an improper input that could not be handled by the target lambda, which correctly counted one instance of the invocation error metric. This test correctly activated the lambda invocation error metric alarm as well as the lambda's composite alarm. The correct lambda resource's composite alarm's *AlarmStatusWidget* widget also showed the alarm as activated on the service's dashboard. The *AlarmWidget* widget for the lambda invocation errors alarm also showed one instance of errors at the alarm's threshold correctly.

For testing the lambda resources error log filter metric an invocation of the target lambda

was activated which produced a log message including the word "Error" to the lambda's log stream. This caused the correct error log filter alarm to activate and that alarm's parent composite alarm was also activated. This caused that resource's *AlarmStatusWidget* widget to show as activated and the error log filter metric's *AlarmWidget* graph also showed one invocation of the error log filter metric at the correct time at the alarm's threshold level. Additionally the *LogQueryWidget* widget displayed the correct error log on the top row of the widget's table with information about the log's source, time, a link to the log stream which the log belongs to and the log's entire message.

The lambda error alarms were also noticed to be activated by a handler which included faulty code that caused its invocations to try running and fail to compile the handler's code. This activated the lambda's error log filter metric and alarm, but not the invocation alarm. The compile failure was also shown on the *LogQueryWidget* and the resource's composite alarm's activation was shown on the composite alarm's *AlarmStatusWidget*. The resource's log errors *AlarmWidget* graph also counted the failed compilations as error metrics because the failed compilation produced an error message. This proved that the error log filter metric and alarm could also be used to catch code compilation errors on code runs, which the invocation metric did not catch.

6.1.2 DynamoDB errors and alarms

The DynamoDB user errors metric and alarm were tested by sending a faulty query to the default DynamoDB API. The query was a *GetItem* operation with a request body that included a resource target key, or the key of the item to fetch, that did not exist in the target table's model. This caused an error response with the message "The provided key element does not match the schema", which is one of the errors that users can make with DynamoDB requests and an error that the user errors metric should catch. This error was correctly caught by the created metrics and its alarm and the alarm's parent composite alarm were correctly activated. The composite alarm also showed as active on the dashboard *AlarmStatusWidget* and the alarm's *AlarmWidget* graph showed one instance of the error at the alarm's threshold.

The other metrics for DynamoDB could not be tested, since they were errors meant to mea-

sure internal service errors split into read and write operations. Since internal service operations are handled by AWS, errors in them like a service outage could not be reproduced for testing this framework. However, these metrics and alarms were deemed to also work, since the other metrics and alarms worked according to the AWS documentation and the same documentation stated these ones to work similarly.

6.1.3 REST API errors and alarms

The REST API error metrics and alarms were tested by sending a test request to the API to trigger a user side error also known as a 4XX error. The test query that was used included the target service's and API's default query path with no specific resource, which should result in a "NotFound" error. According to the AWS documentation this should be one of the types of errors that the REST API user errors metric should track and after testing it the created metric correctly counted one instance of bad request. The test request also correctly activated the REST API user errors metric's alarm and the resource's composite alarm. This could be seen on the service's dashboard since the REST API's *AlarmStatusWidget* turned to an activated state indicating an alarm activating, as well as the error instances breaching the alarm threshold on the alarm's *AlarmWidget* graph.

Similarly to the DynamoDB server-side metrics the server-side metrics of the REST API could not be tested, since they measure internal service errors. Since those kinds of errors are handled by AWS, again it was deemed okay to trust the AWS documentation that those internal service errors would be caught by the server-side error metrics and activate the correct alarms and send notifications about them.

6.1.4 Subscribing and sending notifications

The notification handler was tested with the same test cases that were used to activate the different resource's alarms. After each of them activated an alarm, the notification handler was activated and sent a notification to the subscribers listed in the subscription file. The notification handler was able to listen to the AWS EventBridge's default event bus and with a rule for catching alarm state change events with a detail of the alarm being in an active

state, the notification handler was activated correctly. This handler received the alarm state change event as input and that information was used to construct the notification email. The notifications were sent for all of the test cases and their individual metric's alarms but not the composite alarms like they were configured to.

The notification email's subscriptions were handled with a local JSON file, where alarms could be subscribed and unsubscribed by adding an entry with the receiver's email address and the target alarm's name. The handler correctly sent notifications about different alarms activating to the correct subscribing email addresses as set in the subscriptions file. The sent notification included the activated alarm's name, the resource's name which activated the alarm, the alarm's tracked metrics and their namespace, the reason for the alarm activation, time of activation and the region and AWS account where the alarm exists.

Some things to note with the notification handler are that the delay of sending the notification emails is the same as the evaluation period that was set for the alarm. Since the notification handler only activated on alarm activations and not just on the error metric's invocations, the alarm would only activate once its evaluation period was passed. In this case the notifications might be sent later than expected if the error happened at the start of a long evaluation period. This should be noted when determining the alarm's evaluation period.

Also, if a composite alarm and the composite alarm's child alarms notifications were all subscribed, activating one of the child alarms would also activate its parent composite alarm and both of these activations would send an alarm notification. This could be seen as receiving two notifications about a single alarm or error. This might be considered as too many notifications for a single error so either the composite alarms or their child alarms should be subscribed, but not both.

A problem was also noticed when trying to activate the alarms with faulty test events when the service used a Dead Letter Queue (DLQ) for retrying processing failed events. The failed handler's alarms correctly entered an active state and sent alarm notifications, but also the automatic retry handling of the same event would activate the alarm and send a notification. This could be seen as sending multiple alarms for the same error, which caused some unnecessary flooding of the notification's communication channel.

6.2 Requirements analysis

In this section the created framework and its application to the target Timeseries service are measured against the requirement criteria from Section 4.4.

6.2.1 Detecting errors

The framework allowed many different kinds of errors to be monitored and detected in the target service. By using AWS's own tools for monitoring errors in the resources and services that they offer, the monitoring of those resources was easy to take up and add to the existing system. Each of the resources that error monitoring was added to included many different types of metrics to monitor, including many different error metrics. For this framework all of these error metrics were added to be monitored since the requirements specified that the framework should be able to catch as many different types of errors as possible per evaluation criteria **EC1**.

The resources to monitor were chosen to be all of the different resources and services that the target Timeseries service included, which were lambda functions, REST APIs and DynamoDB tables. The lambda functions could be monitored for different invocation errors like code exceptions and timeouts. The lambdas log groups were also filtered to notice any log messages indicating an error. The REST APIs were monitored for user side 4XX errors and server-side 5XX errors, including errors like bad requests and internal service errors. The DynamoDB tables could also be monitored for user and server-side errors, where server-side errors were split to two different metrics as read and write operations. These metrics allowed the framework to monitor and detect many different kinds of errors to cover as much as possible of the resource's possible errors.

6.2.2 Activating alarms on errors

Using AWS CloudWatch alarms for the error metrics of the different resources and services of the Timeseries service worked well and the alarms were always correctly activated after noticing an error. The alarms were also noted to being highly configurable by being able to set the alarm's error count threshold, evaluation period's length and the comparison operator

for the followed error metric. This allowed the alarms to fit many different monitoring scenarios where a single error should not activate an alarm but rather a certain amount of errors in a given time window. This fulfils evaluation criteria **EC2**.

AWS also offered a useful alarm construct for bundling a given resource's alarms together with CompositeAlarms, which could be configured to activate when some or any of their child alarms activated. This allowed the target service's resource's alarms to be bundled together to allow them to be monitored as a group by their resource instance. Any errors occurring in this resource group would activate the composite alarm for that resource instance and indicate clearly where the error happened. This feature could also be used to tie alarms together by some other common aspect such as same business context or handler.

6.2.3 Subscribing and sending alarm notifications

This framework explored a few different ways for managing alarm notifications and their subscriptions. Firstly the AWS SNS service was considered, which had the built-in functionality for creating the alarm activation notification emails when added to the target alarms as an alarm action. The SNS service however required individual SNS topics to be created for configuring the receivers for each notification. This would be a simple solution for cases where there would only be a few parties to notify for the alarms, since it would keep the amount of SNS topics low which would make it easier to manage the subscribers for those topics and these same topics could be set for all alarms. However, if the alarms or composite alarms would want to be subscribed individually, each of them would need their own SNS topic, which could lead to a very high number of topics and managing them and their subscriptions could become very laborious.

Another solution that was explored was using the Akamon Dataplatform system's Message service for sending notifications for the alarms. The Message service could be used to send customized emails to the given subscribers after alarms were activated. This solution was however decided against since the used Message service was a part of the system to be monitored. This introduced a possible risk of the Message service including an error that caused it to not work for sending the notifications. This would result in errors occurring

in the system which would not send notifications because of the problems in the Message service. The problems in the Message service could also not send notifications, so noticing them would be difficult.

The last explored solution was to send the notifications with AWS SES by creating a lambda function that would activate either on the alarm actions or with the AWS EventBridge's default alarm state change events. This lambda could then be used to construct a custom notification about the activated alarm and to send that notification to the configured recipients. This solution offered a lot of freedom when it came to the notifications content, activation methods where other sources could also publish an event to run the lambda or run it some other way, and notification channels that could be used to deliver the notifications.

While in this framework's application AWS SES was used, an external email provider or other software like a message to the organizations chosen communication software like Microsoft Teams could also be added to the handler. Subscriptions could be designed to be managed in many different ways including a table in a database or a local configuration file, which should fulfil evaluation criteria **EC3**. This way of sending the notifications with SES was considered to be safer since errors in SES would be handled by AWS. However it is worth considering to also monitor the notification handler for errors. If any errors would occur in the notification handler function, they could prevent any notifications from being sent which makes it a critical part of the alarm notifications functionality.

6.2.4 Alarm dashboard

The created framework allowed a service level dashboard to be created for the target *Time-series* service which could be accessed and viewed in the AWS Management console. The dashboard included components to display the statuses of each of the services resources alarms, a component to display any error logs that the services resources would produce and graph components to see the activation history of each of the services alarms. The alarm status components and the alarm history graph components could be opened to view more information about them and a link to access that alarm in the alarm details page.

The dashboard and its components were tested by activating different alarms with test events

and requests in the target service and they seemed to work correctly for all the tracked errors. The alarm status components could be used to see in real time which alarms were activated and when because they turned to an activate state with a red color when the alarm noticed an error. The log filter component also worked well with displaying any error logs that the target resources produced as well as providing a link to access those logs log streams for further investigation. The graph components also correctly showed the recent history of the alarm's activations with the metrics and error threshold of that alarm to display how it behaves over time. This dashboard, its widgets and the extra information that they provided about the error metrics and alarms satisfy the criteria **EC4** and contribute to the **EC5** criteria.

6.2.5 Notification information

The created alarm notifications were required to be sent on alarm activations to the given subscribers and to include information about the activated alarms to help solve them in the right part of the system. The notification handler was tested by subscribing to some of the alarms from the target Timeseries service and activating them. The handler correctly sent out the notification emails for the right subscribers on each of the test cases and all of them included the data that was configured for the notification email bodies.

The notification emails included data about the activated alarm's name, the alarmed resource's name, which also included the service's name, the metric's name and namespace that produced the error, the reason for the alarm's activation, time of activation and the region and AWS account where the monitored resource exists. These should provide enough information for the subscriber to gain a clear understanding of what caused the alarm to activate, when it happened and where to look for more information to fix the cause of the errors. This satisfies evaluation criteria **EC5**.

6.2.6 Ease of import to other services

The frameworks error metrics and their corresponding alarms were created as IAC with the AWS CDK, since the target system is also built with it. This meant that the code for the monitoring could easily be added to the target services CDK code and integrated into the

existing system and its resources, since the CDK code constructs are designed to be very compatible. The error monitoring dashboard and its alarm widgets were also created with CDK code and added to the target service. The CDK code could also be version controlled with the rest of the services code.

After the initial version of the framework was deemed to work correctly in the target service's CDK code, it was imported to Akamons CDK code library, where it was exported as a construct class that included individual methods for monitoring specific AWS resource. These methods would create all the correct error metrics and their alarms for the target resource and add them to the target service's infrastructure. The construct class also created a dashboard and alarm widgets for the service based on the resources that the class handled and the errors and metrics it created for those resources. With this solution adding the error monitoring to any of the target system's services would be very easy, since it would only require the monitoring construct class to be added to the service's CDK code and the resource monitoring methods to be called with each of the resources that the service uses. The functionality of creating the correct metrics, alarms, dashboards and widgets would be hidden inside the imported class. This way the framework also fulfils evaluation criteria **EC6**.

6.3 Future development

The framework developed in this thesis seemed to work well as a first version of a distributed systems services monitoring tool. It handled all the requirements set for it, but it also includes room for improvement. This section is focused on discussing these improvement topics.

One of the most important future development cases for the framework is to add more resources as its monitoring targets. Since AWS offers so many different resources which can be used in the target system or service, the monitoring should also be able to monitor all of these resources. In its current form this framework can only monitor for errors in lambda functions, DynamoDB tables and REST APIs. This is quite a small selection of AWS resources, though all of these are very common and highly used. The framework should be developed further by selecting a new service from the target system to be monitored, check which resources it includes that should be monitored and if any of the resources are not yet

supported by the framework support for those resources should be added to it. Then once the monitoring framework includes functionality for monitoring all of the services resources, it could be added to the target service. This would ensure that monitoring is only added for the required resources, while also making sure that all of the systems services and their resources would be monitored and that the framework is easy to apply to a new service.

Another part of the framework that includes a lot of room for future development is the notification handler. The notification email's body was left very simple in this thesis, only providing the most necessary information about the activated alarm with some simple styling with HTML. This notification email could be further developed in various ways like enriching the emails data. Some further data about the activated alarm could be fetched from CloudWatch and added to the email to improve its ability for tracing the alarms errors and their causes, for example the target resources log stream or X-ray trace. The notification email could also include links, for example to the correct services error monitoring dashboard, the activated alarm's details page or an external monitoring site if needed.

Since the notification handler was implemented with running a simple lambda, it should be very free to be developed in any way the organization sees best fit for their system and monitoring goals. The notification method could also be easily changed from SES if the organization wished to receive notifications another way. Some new features to the notification handler could also include notifying third parties like customers if a monitoring plan was made for them to also receive notifications of services producing errors. Alarm activation information could also be saved for creating system availability reports or fault tolerance reports.

The notification subscriptions are also very simple in this initial version of the monitoring framework. Instead of handling them in a local JSON file where any modifications require a new deployment of the code, their usability and management should be improved by allowing easier ways to subscribe and unsubscribe the different alarm notifications. This could for example be achieved by saving and managing the subscriptions in a database like DynamoDB. This would also allow different subscription models to be enabled by allowing alarm, service or resource specific subscriptions. A UI could also be added to make it easier for the users to manage their alarm subscriptions. This would be particularly beneficial for

non-technical users such as product owners or client representatives.

7 Summary

One of the core concepts of distributed systems is the ability to encapsulate functionality within small services that through mutual communication can achieve a singular system. This includes many benefits like reduced dependencies between modules or services, higher system fault tolerance and resistance and architectural flexibility. Since the systems services are independent units meant to only handle their own internal functions and communicate to the rest of the system by lightweight requests, all of the services possible errors would also be encapsulated inside that service so that they would not affect the rest of the system. While this would increase the systems resilience and allow the systems overall status to remain healthy even if some of its services were failing on errors, hiding the services errors from the system can also result in them being too hidden so that they are not noticed and thus also not fixed.

In this thesis this problem of distributed systems' services hiding their internal errors from the rest of the system was studied and a solution to noticing and bringing these errors to the developers' knowledge was designed. Several requirements were created for this solution including noticing as many of the services errors as possible, creating alarms to activate when those errors happen, including information about the noticed errors to allow the errors to be fixed, sending notifications of activated alarms and making the solution easy to apply to other services of the system as well. The designed solution was applied to Akamon Innovations' Dataplatform system where it was also tested and evaluated.

The designed solution was an error monitoring framework, which would create metrics for tracking errors produced by the target service's different resources. The framework would also create different alarms for following these metrics and if the target metric exceeded the alarm's error count threshold, the alarm would activate. The framework also created an error monitoring dashboard for the target service, where different widgets were added to display the error metrics' and alarms' data to enable them to be monitored to solve the service's errors. A notification handler was also created for the target system, which would run on alarm state change events from activated alarms. This notification handler would create an alarm notification email with the activated alarm's data and send it to the correct subscribers

who wish to know when those alarms activate.

The created framework worked well when it was tested in the *Timeseries* service of Dataplatform. Its first version could monitor several AWS resources for errors, including lambdas, REST APIs and DynamoDB tables. After causing test errors in these resources the resources' error metrics counted the invocations correctly and the alarms activated as they were supposed to, which could also be viewed on the services error monitoring dashboard. Different alarm activations could also be subscribed to in the notification handler which would then correctly send out notifications when the subscribed alarms would activate. The monitoring resources were also bundled together as an AWS CDK construct, which allowed them to be imported and used in other services of the system very easily, since all of Dataplatform's services are constructed with CDK code.

Overall, this framework seemed to solve the problem of services encapsulating and hiding their errors from the rest of the system by bringing them to the developers' knowledge through different alarms, notifications and visual monitoring dashboards. It also fulfilled all of the requirements set for it pretty well. Future development for this framework could include adding support for monitoring other types of resources, creating a more dynamic subscription method for the alarm notifications, enriching the notification's data about the activated alarm and providing support for custom widgets on the dashboards.

Bibliography

Akamon Innovations Oy. 2024. “Akamon Innovations homepage”. Visited on April 21, 2024. https://akamon.fi/.

Alonso, Gustavo, Fabio Casati, Harumi Kuno, and Vijay Machiraju. 2004. “Web Services”. In *Web Services: Concepts, Architectures and Applications,* 123–149. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN: 978-3-662-10876-5. https://doi.org/10.1007/978-3-662-10876-5_5. https://doi.org/10.1007/978-3-662-10876-5_5.

AWS. 2023a. “AWS Cloud Development Kit”. Visited on October 11, 2023. https://aws.amazon.com/cdk.

———. 2023b. “AWS CloudTrail”. Visited on October 11, 2023. https://aws.amazon.com/cloudtrail/.

———. 2023c. “AWS CloudWatch”. Visited on October 11, 2023. https://aws.amazon.com/cloudwatch/.

———. 2023d. “AWS My Applications”. Visited on February 1, 2024. https://docs.aws.amazon.com/awsconsolehelpdocs/latest/gsg/aws-myApplications.html.

———. 2023e. “AWS X-Ray”. Visited on February 1, 2024. https://aws.amazon.com/xray/.

———. 2023f. “What is an Event-Driven Architecture?” Visited on December 22, 2023. https://aws.amazon.com/event-driven-architecture/.

Boniface, Michael, Bassem Nasser, Juri Papay, Stephen C. Phillips, Arturo Servin, Xiaoyu Yang, Zlatko Zlatev, et al. 2010. “Platform-as-a-Service Architecture for Real-Time Quality of Service Management in Clouds”. In *2010 Fifth International Conference on Internet and Web Applications and Services,* 155–160. https://doi.org/10.1109/ICIW.2010.91.

Castro, Paul, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. November 2019. “The Rise of Serverless Computing”. *Commun. ACM* (New York, NY, USA) 62, number 12 (): 44–54. ISSN: 0001-0782. https://doi.org/10.1145/3368454. https://doi.org/10.1145/3368454.

Delgado, N., A.Q. Gates, and S. Roach. 2004. “A taxonomy and catalog of runtime software-fault monitoring tools”. *IEEE Transactions on Software Engineering* 30 (12): 859–872. https://doi.org/10.1109/TSE.2004.91.

Fielding, Roy Thomas. 2000. “Architectural Styles and the Design of Network-based Software Architectures”. *Doctoral dissertation, University of California, Irvine,* 76–106. https://ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

Goyal, Sumit. 2013. “Software as a service, platform as a service, infrastructure as a service-a review”. *International journal of Computer Science & Network Solutions* 1 (3): 53–67.

Haberkorn, Marc, and Kishor Trivedi. 2007. “Availability Monitor for a Software Based System”. In *10th IEEE High Assurance Systems Engineering Symposium (HASE’07),* 321–328. https://doi.org/10.1109/HASE.2007.49.

Halili, Festim, Erenis Ramadani, et al. 2018. “Web services: a comparison of soap and rest services”. *Modern Applied Science* 12 (3): 175.

Hariri, S., and H. Mutlu. 1991. “A hierarchical modeling of availability in distributed systems”. In *Proceedings 11th International Conference on Distributed Computing Systems,* 190, 191, 192, 193, 194, 195, 196, 197. Los Alamitos, CA, USA: IEEE Computer Society. https://doi.org/10.1109/ICDCS.1991.148664. https://doi.ieeecomputersociety.org/10.1109/ICDCS.1991.148664.

Harness. 2023. “What are Software Application Monitoring Best Practices?” Visited on October 29, 2023. https://www.harness.io/blog/software-application-monitoring-best-practices.

Hevner A. R., Park J. Ram S., March S. T. 2004. *Design Science in Information Systems Research.* 75–106. Visited on December 6, 2023. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.1725&rep=rep1&type=pdf.

Iosup, Alexandru, and Dick Epema. 2011. “Grid Computing Workloads”. *IEEE Internet Computing* 15 (2): 19–26. https://doi.org/10.1109/MIC.2010.130.

Janes, Andrea, Alberto Sillitti, and Giancarlo Succi. 2013. “Effective dashboard design”. *Cutter IT Journal* 26 (1): 17–24.

Joyce, Jeffrey, Greg Lomow, Konrad Slind, and Brian Unger. March 1987. “Monitoring Distributed Systems”. *ACM Trans. Comput. Syst.* (New York, NY, USA) 5, number 2 (): 121–150. ISSN: 0734-2071. https://doi.org/10.1145/13677.22723. https://doi.org/10.1145/13677.22723.

Kufel, Łukasz. 2016. “Tools for distributed systems monitoring”. *Foundations of computing and decision sciences* 41 (4): 237–260.

Mathew, Sajee, and J Varia. 2014. “Overview of amazon web services”. *Amazon Whitepapers* 105:1–22.

Newman, David Robert. 2023. “Distributed Systems Monitoring: The Essential Guide”. Visited on October 28, 2023. https://www.linode.com/docs/guides/monitoring-software/.

Oracle, Qusay H. Mahmoud. 2005. “Service-Oriented Architecture (SOA) and Web Services: The Road to Enterprise Application Integration (EAI)”. Visited on April 1, 2024. https://www.oracle.com/technical-resources/articles/javase/soa.html.

Peffers, Ken, Tuure Tuunanen, Charles Gengler, Matti Rossi, Wendy Hui, Ville Virtanen, and Johanna Bragge. 2006. “The design science research process: A model for producing and presenting information systems research”. *Proceedings of First International Conference on Design Science Research in Information Systems and Technology DESRIST*.

Poslad, Stefan. 2011. *Ubiquitous computing: smart devices, environments and interactions.* John Wiley & Sons.

Sliwa, Carol. 2003. “What is an Event-Driven Architecture?” Visited on December 25, 2023. https://www.computerworld.com/article/2570503/event-driven-architecture-poised-for-wide-adoption.html.

Srinivas, J, K Venkata Subba Reddy, and A Moiz Qyser. 2012. “Cloud computing basics”. *International journal of advanced research in computer and communication engineering* 1 (5): 343–347.

Telang, Tarun. 2023. “Distributed Systems Monitoring: The Essential Guide”. Visited on October 26, 2023. https://www.loggly.com/use-cases/distributed-systems-monitoring-the-essential-guide/.

Thoke, Virendra Dilip. 2014. “Theory Of Distributed Computing And Parallel Processing With Its Applications, Advantages And Disadvantages.” *International Journal of Innovations in Engineering Research and Technology,* 1–11.

Tsai, J.J.P., K.-Y. Fang, and H.-Y. Chen. 1990. “A noninvasive architecture to monitor real-time distributed systems”. *Computer* 23 (3): 11–23. https://doi.org/10.1109/2.50269.

Van Steen, Maarten, and Andrew S Tanenbaum. 2016. “A brief introduction to distributed systems”. *Computing* 98:967–1009.

Verissimo, Paulo, and Luis Rodrigues. 2001. *Distributed systems for system architects.* Volume 1. Springer Science & Business Media.

Wieringa, Roel J. 2014. *Design science methodology for information systems and software engineering.* Springer.

Yeo, Chee Shin, Rajkumar Buyya, Hossein Pourreza, Rasit Eskicioglu, Peter Graham, and Frank Sommers. 2006. “Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers”. In *Handbook of Nature-Inspired and Innovative Computing: Integrating Classical Models with Emerging Technologies,* edited by Albert Y. Zomaya, 521–551. Boston, MA: Springer US. ISBN: 978-0-387-27705-9. https://doi.org/10.1007/0-387-27705-6_16. https://doi.org/10.1007/0-387-27705-6_16.

Zulkernine, M., and R.E. Seviora. 2002. “A compositional approach to monitoring distributed systems”. In *Proceedings International Conference on Dependable Systems and Networks,* 763–772. https://doi.org/10.1109/DSN.2002.1029022.