MSc thesis

Master's Programme in Computer Science

Digital content popularity counting with Amazon Web Services

Joonas Sarapalo

October 8, 2020

Faculty of Science

University of Helsinki

Supervisor(s)

Prof. Jussi Kangasharju

Examiner(s)

Prof. Jussi Kangasharju, Dr. Walter Wong

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Joonas Sarapalo

Työn nimi — Arbetets titel — Title

Digital content popularity counting with Amazon Web Services

Ohjaajat — Handledare — Supervisors

Prof. Jussi Kangasharju

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	October 8, 2020	69 pages

Tiivistelmä — Referat — Abstract

The page hit counter system processes, counts and stores page hit counts gathered from page hit events from a news media company's websites and mobile applications. The system serves a public application interface which can be queried over the internet for page hit count information. In this thesis I will describe the process of replacing a legacy page hit counter system with a modern implementation in the Amazon Web Services ecosystem utilizing serverless technologies. The process includes the background information, the project requirements, the design and comparison of different options, the implementation details and the results. Finally, I will show how the new system implemented with Amazon Kinesis, AWS Lambda and Amazon DynamoDB has running costs that are less than half of that of the old one's.

ACM Computing Classification System (CCS)
Computer systems organization → Architectures → Distributed architectures → Cloud computing
Software and its engineering → Software creation and management → Designing software → Requirement analysis
Software and its engineering → Software creation and management → Designing software → Software design engineering

Avainsanat — Nyckelord — Keywords

cloud computing, serverless, AWS

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Networking study track

Contents

Technical Abbreviations and Acronyms

API	Application Programming Interface
CDN	Content Distribution Network
FaaS	Function as a Service
HTTP(S)	Hyper Text Transfer Protocol (Secure)
IaaS	Infrastructure as a Service
IOPS	Input/Output operations per second
JSON	JavaScript Object Notation
LRU	Least Recently Used
NAT	Network Address Translation
PaaS	Platform as a Service
RAM	Random Access Memory
SaaS	Software as a Service
SQL	Structured Query Language
SSD	Solid-state Drive
TTL	Time to live
URL	Uniform Resource Locator

AWS Specific Abbreviations and Acronyms

ALB	Application Load Balancer
AWS	Amazon Web Services
AZ	Availability Zone
DAX	DynamoDB Accelerator
ECS	Elastic Container Service
ELB	Elastic Load Balancer
EC2	Elastic Compute Cloud
IAM	Identity and Access Management
KDS	Kinesis Data Stream
KPU	Kinesis Processing Unit
RCU	Read Capacity Unit
RDS	Relational Database Service
S3	Simple Storage Service
SES	Simple Email Service
VPC	Virtual Private Cloud
WCU	Write Capacity Unit

Project Specific Abbreviations and Acronyms

CS	Core Service
EP	Event Processor
PHC	Page Hit Counter
UCS	User Content Service

Whenever a new abbreviation on acronym is introduced in the thesis it is marked with parentheses after the full name. For example: Application Programming Interface (API).

1 Introduction

Technological change over the past few decades has changed many domains of business by bringing forth new challenges and opportunities. A field that has faced quite significant change thanks to these influences is the news media. Year by year a greater proportion of news consumption happens in the digital world of social media, news websites and mobile applications (*Here's How The Way We Read Newspapers Has Changed* 2020; *5 key takeaways about the state of the news media in 2018* 2018). This has changed the way that news are consumed and allows news media organizations to become much more technical and data driven by releasing news and gathering data at even increasing volumes (Norberg-Schultz Hagen et al., 2020).

Arguably, for any digital content creator or distributor, one of the most important pieces of data to gather and analyze is the amount of traffic the content on their digital services receive. This data is useful as it can inform these organizations about which of their content are getting a lot of user engagement and which are not as popular. With this information in mind better decisions about what sort of content to produce can be made in the future. It is the job of a page hit counter service to process and store such data. The page hit data can also be utilized by customer facing services such as most popular content listings.

In this thesis I describe the planning, the implementation and the results of a project to redesign and replace a news media company's legacy page hit counter (PHC) service with a new one. This new implementation is used especially to generate customer facing most popular content listings.

New PHC is meant to be integrated with existing services in the news media company's architecture, namely the Event Processor (EP) and the Core Service (CS). EP is a public Application Programming Interface (API) (*Application programming interface* 2020) that consumes some of the events sent by the news media company's websites and mobile applications. These events contain all sorts of information, such as page visit information which are of interest in the context of the new PHC service. CS is a service which integrates many services together to allow them to communicate with each other. In the context of new PHC service it is used by some other services to query the new PHC service for the page hit data for the generation of the most popular content listings.

1.1 Description of the thesis

In this section I will discuss the scope of the implementation of the new PHC service along with the goals, the requirements and the research questions. I will describe how the performance measurements of the old and new systems are made and how they are compared.

The scope thesis is focused on describing the design and implementation of the new PHC service. I will describe the relevant technical background which includes storage technologies, cloud computing and Amazon Web Services (AWS) among other things (Hayes, 2008; *What is AWS* 2020). I will compare different service designs, methods and technologies. I will evaluate and compare the performance of the new and old services in different metrics like running costs. Finally I will reflect back on how well the end results satisfy the stated requirements and goals.

1.1.1 Requirements and goals

The requirements and goals for this new service are the following:

- To provide the same functionality as the old service:
    1. Consume and count page hits from event data
    2. Store page hit counts in such a way that they can be queried for with different query parameters
    3. Serve a page hit API which can be queried by other service
- To use EP as the public API for consuming page hit events.
- To use AWS to implement the infrastructure and different parts of the service
- To make the service brand agnostic by making it able to serve each one of the news media company's brands. This is in contrast to old PHC which had a separate instance running for each one of the brands.
- To filter out events triggered by spammers and bots if feasible.
- To optimize its performance over metrics of interest like running costs:

  - Completely remove some older services used to run the legacy PHC service if they are no longer used by anything else.
  - Make design choices based on the estimated costs of different approaches.

1.1.2 Research questions and measurements

The specific research questions this thesis answers are:

1. What is the best way to implement the new PHC service with the stated requirements and goals in mind?
2. How well does the new implementation perform when compared to the old one in running costs and other metrics?
3. Did the new service manage to satisfy the stated requirements and goals?

Research question 1. will be answered in chapter 5 where I will compare different designs that fit the project requirements. Chapter 6 describes the service implementation in detail to highlight some of the important choices made to achieve the stated goals of the project.

To answer the research question 2. I will compare performance metrics gathered by AWS on the old and new services. These comparisons are discussed in chapter 7. I will not discuss the exact running costs of the old and new services as they are company secrets, thus I will be making a proportional comparison between them. The running costs of the old PHC shall add up to one, and if the running costs of the new implementation are less than that I will have achieved the goal of reducing the running costs of the PHC service.

Finally, research question 3. will be answered in chapter 8 where I conclude the thesis.

1.2 Structure of the thesis

This thesis is organized as follows:

- In chapter 2 I describe topics in computer science and software engineering that are important background knowledge for the rest of the thesis.
- In chapter 3 I describe Amazon Web Services (AWS) with a focus on specific AWS services relevant to the project.

- In chapter 4 I describe the business and technical requirements for the project.
- In chapter 5 I describe and compare different designs for the new PHC that match the project requirements.
- In chapter 6 I describe the chosen design in detail and discuss changes made to it during implementation phase of the project.
- In chapter 7 I discuss and compare the measurements taken from the old and the new PHC services.
- In chapter 8 I conclude the thesis by reflecting back on it and discussing future work left in the project.

2 Theoretical background

In this chapter I introduce and describe an assortment of computer science, data science and software engineering concepts that are important background knowledge for the rest of the thesis. First I will introduce the concept of cloud computing, followed by concepts around databases, content delivery networks, virtual machines, containers, random forests and time windows.

2.1 Cloud computing

The cloud is a combination of remote data centers and all the hardware and software running there (Armbrust et al., 2010). Cloud computing refers to the on-demand access to these remote computing and storage resources with either a provisioned or a pay-as-you-go model (Hayes, 2008; *What is AWS* 2020).

There are many different cloud service providers with varying offerings. The ones that are holding most of the market share at the moment of writing this thesis are: Amazon Web Services (AWS), Microsoft Azure, Google Cloud Services and Alibaba Cloud (*AWS vs Azure vs Google Cloud Market Share* 2020). For the rest of the thesis AWS will be the only cloud service provider I will be discussing as that is where the infrastructure of the new PHC will be hosted.

2.1.1 Reasons to use cloud computing

There are many reasons why an entity might choose to utilize cloud computing rather than doing self-hosting or deploying another traditional setup like renting racks in data centers. Here are common reasons for using cloud computing:

- Economies of scale: Cloud computing providers can offer greater range of services at lesser prices when compared to traditional vendors or self-hosting, because they are operating their infrastructure at a greater scale (*Economies of scale* 2020; Armbrust et al., 2010).

- Agility: Cloud computing allows clients to experiment with new hardware and software at a faster pace and generally with no additional upfront costs (*What is AWS* 2020; Armbrust et al., 2010). Example of agility is getting a new prototype application running on a server without having to spend time on things like provisioning hardware in advance and manually installing all the required software on the server side.

- Scalability: In a cloud environment resources can grow and shrink in response to changing needs in real-time, thus optimizing the running costs and availability of these resources (*What is AWS* 2020; Armbrust et al., 2010). This can happen either with horizontal scaling by adding more instances of the resource in question or with vertical scaling by adding more capability to existing instances. Scalability is also known as elasticity in some contexts.

- Global infrastructure: Many cloud providers offer resources all around the world making it possible for clients to host their services as close to the users as possible to decrease latency (*What is AWS* 2020; *Azure global infrastructure* 2020). Distributed infrastructure also increases the durability of the services running on them as even in case some event (power outage, natural disaster etc.) causes some physical hosting location to become unreachable other locations are able to pick up their work.

- Service model: Clients have access to many services without having to understand the underlying technology at a deep level because of abstractions provided by the cloud computing vendors.

These reasons can be summarized with the following idea: the clients of cloud computing providers are practically outsourcing most of the upfront and continuous knowledge and capital required to run these computing and storage infrastructures, thus the clients can focus more on their core businesses without having to worry about provisioning the right amount of computing capacity in advance and doing infrastructure maintenance.

2.1.2 Potential downsides of cloud computing

While there are many upsides to using cloud computing as described in the last section, cloud computing is not completely devoid of potential downsides. These potential downsides include the following: data management, performance and reputation fate sharing (Armbrust et al., 2010). At the same time many cloud providers provide services and

features which address some of these issues. I will discuss some of the ones provided in AWS.

Downsides in data management are comprised of few parts: data lock-ins, data transfer bottlenecks and data confidentiality issues (Armbrust et al., 2010). Data lock-in happens when it is difficult to extract data out of a cloud computing platform. AWS offers many ways of avoiding data lock-in such as co-location hosting of data through Direct Connect (*Why Cloud Lock-In is a Myth: The Openness of AWS* 2018). Data transfer bottlenecks refer to issues around limited bandwidth in cloud computing settings. AWS has introduced services to address these issues, like AWS Snowball which is a truck carrying storage units capably of storing data in exabyte scale (*AWS Snowmobile* 2020). Data confidentiality refers to the fact that there is always a question about the confidentiality of the data when some other entity is handling it. AWS manages data confidentiality through many mechanisms such as encryption, access control and giving the customer the choice of where to store the data (*Data Privacy FAQ* 2020).

Performance problems include unpredictability, scalability and bugs (Armbrust et al., 2010). These are many issues that can affect any programs and services hosted by the cloud service providers. At the same time the service providers are constantly improving their software which leads to overall improvement in performance quality over time. AWS offers different availability and integrity guarantees to different services such as the 99,99% availability of the standard class of the storage service Amazon S3 (Simple Storage Service) (*Amazon S3* 2020). AWS performance is discussed in more detail in chapter 3.

Reputation fate sharing refers to events happening on the cloud platform which are out of the customers control but can still affect their experience and reputation in the digital and even the real world (Armbrust et al., 2010). These events include things like getting your IP address added to a email black list because some other entity's program is spamming emails over the same public IP address. AWS tries to address these kind of issues through many mechanisms such as the Amazon SES (Simple Email Service) automatically scanning outgoing emails for spam (*AWS SES FAQs* 2020).

2.1.3 Different types of cloud services

Cloud offerings can be split into the categories of Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). They represent different levels of abstraction on top of the underlying hardware running the cloud (Vaquero et al., 2009).

I will also describe the concept of serverless which is also known as Function as a Service (FaaS) (Castro et al., 2017). FaaS can be categorized as a special case of PaaS.

IaaS refers to virtualized infrastructure running in the cloud. This infrastructure is elastic by nature and provides a generic platform to run any applications on it. The clients are responsible for some portion of the configuration and maintenance of the IaaS infrastructure. IaaS is closest to the bare server hardware of the three categories (Vaquero et al., 2009; Rimal et al., 2009).

PaaS is one level of abstraction above IaaS. It refers to services that provide platforms to run applications on them. PaaS takes care of most or all of the underlying virtual infrastructure set up, thus allowing the clients to focus on the applications (Vaquero et al., 2009; Rimal et al., 2009). Heroku is a good example of a PaaS (*Heroku* 2020).

SaaS refers to remotely hosted web applications which are available to the users. SaaS software is often locked behind a subscription wall. SaaS services do not have to be hosted in the cloud. The benefit of SaaS is that it allows users to use software without having to install and maintain it (*What is SaaS?* 2020; Rimal et al., 2009).

Serverless (or FaaS) is similar to PaaS with the important distinction that serverless services do not have dedicated virtual servers continuously running them. Serverless processes are executed when they are needed and terminated some time after that. Utilizing serverless computing can yield significantly savings in operating costs, as serverless applications usually operates with a pay-as-you-go model. Creating new instances of a serverless applications takes time and thus can increase the latency of those applications (Adzic and Chatley, 2017). Serverless applications are almost stateless and can therefore be easily horizontally scaled (Kablan et al., 2015).

2.2 Database concepts

In this section I will describe how key-value stores like NoSQL databases and caches work and contrast that with the functionality of relational databases. I will also describe other database concepts like replication. This information will be utilized in describing AWS services and the design and implementation of the new PHC service. I will also describe the ACID and CAP theorems which are theoretical background information about database transactions and distributed databases.

2.2.1 ACID and CAP

ACID and CAP are theorems which describe properties of databases and transactions.

ACID (*atomicity*, *consistency*, *isolation*, *durability*) are properties of database transactions. Atomicity means that there are no partial transactions, they are either fully committed or they fail. Consistency means that transactions do not lead to states with inconsistencies in the data. Isolation means that transactions are isolated from other transactions until they are committed. Durability means that committed transactions cannot be changed. ACID properties guarantee the validity of database transactions even in rare cases like a power failure happening in the middle of a transaction (Gilbert and Lynch, 2002).

CAP is a theorem which says that a distributed database cannot posses more than two of the following properties at the same time: *consistency*, *availability* and *partition tolerance*. The proof for this is quite long and complicated and can be read here: (Gilbert and Lynch, 2002).

2.2.2 Database replication

Non-distributed databases experience performance degradation when the capacity of the database is not enough to serve all of the incoming traffic. Read replicas can be utilized to improve performance by reducing the load on any one database instance (*Amazon RDS Read Replicas* 2020; *Replication (computing)* 2020).

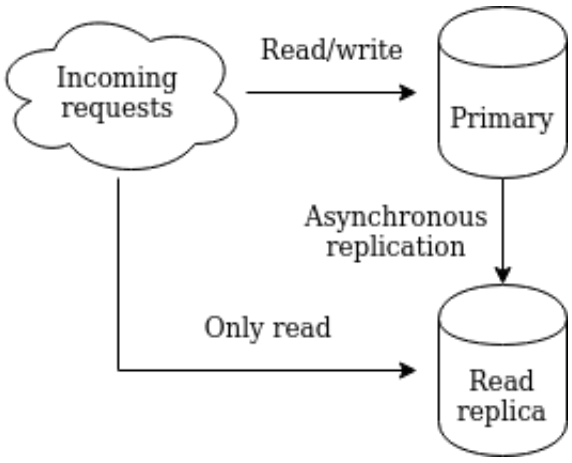


Figure 2.1: Primary database with one read replica

Read replica architecture has one primary/master database which can be written to and read from. The contents of this database are asynchronously replicated to all read replicas

of that database. Incoming write requests are directed to the primary. Read requests are split between the replicas and the primary/master (*Amazon RDS Read Replicas* 2020).

Other possible setups to improve database performance include multiple masters architecture where data is asynchronously synchronized between the databases (*Replication (computing)* 2020).

2.2.3 NoSQL database

NoSQL database refers to a non-relational database. Some of the motivations behind the emergence of non-relational databases was the desire to simplify database schemas and to make horizontal scaling of databases easier. There are four major types of NoSQL databases: key-value pair, document, wide-column and graph (Leavitt, 2010).

Key-value pair database stores a blob of any kind of data behind a key which is used to access that data (Leavitt, 2010). Document databases like MongoDB store data as documents (also known as semi-structured data) that can be queried to operate on specific fields of the documents (*MongoDB* 2020). Wide-column databases like Google's BigTable are similar to key-value databases except they can structure the data found behind each key into indexable columns for more structured data storage (*Overview of Cloud Bigtable* 2020). Graph databases like Neo4j model relations between data as graphs between nodes. Graph databases are very flexible in the ways that data can be queried from them and how relations can be structured (*Why Graph Databases?* 2020).

Compared to relational databases which are usually strictly consistent, NoSQL databases are often only eventually consistent. This is a trade off to make the database highly available and partition tolerant as described by CAP theorem (Gilbert and Lynch, 2002). Eventual consistency means that distributed NoSQL databases may not share the same state all the time but they are eventually going to arrive to it though communication (Vogels, 2009).

2.2.4 Cache

Cache can be a software or a hardware component. In the context of this thesis we are interested in the software in-memory caches because the new implementation of PHC utilizes many levels of in-memory caching to reduce running costs.

The purpose of a cache is to act as a fast access data store for frequently used data. Data

can end up in a cache through couple means. Some common ones include data being added there in advance because it is known (or predicted) to be needed in the future, or the data is put there as a result of some expensive computation that would be wasteful to run all the time (*Caching Overview* 2020). This second method can be thought as a memoization technique (Norvig, 1991).

A software cache is commonly a key-value storage of some sort. Reading data from the cache can lead to a cache hit or a cache miss depending on if the key used to read the data exists in the cache (*Caching Overview* 2020).

In the context of this thesis we are interested in the least recently used (LRU) cache as it is used for the memoization of the page hit database queries. An LRU cache refers to a cache using LRU as its cache replacement policy. A cache replacement policy defines what items are removed from the cache and when to make room for new items. LRU cache policy tries to figure out which pieces of data are the least recently used ones and drop them out. The idea behind LRU is to keep frequently accessed ”hot” data in the cache and thus increase the chance of a cache hit (*Using Redis as an LRU cache* 2020).

2.3 Content delivery network

Content delivery network (CDN), also know as content distribution network, is a distributed network built specifically for fast delivery of content around the Internet. PHC will be utilizing the CDN of AWS called CloudFront for query response caching (*Amazon CloudFront Key Features* 2020).

The content delivered by CDNs range anywhere from text and videos to applications. To speed up content delivery CDNs are composed of servers and data centers in geographically distinct locations. These hardware host edge caches which serve any requests going to the origin servers if they have the requested resource. Getting a cache hit at a edge cache can greatly reduce latency of a request (*Content Distribution Network* 2020).

CDNs utilize techniques such as distributed denial-of-service (DDoS) defences and load balancing across their servers to improve the availability of their services (Mirkovic and Reiher, 2004; *What Is Load Balancing?* 2020; *Content Distribution Network* 2020).

2.4 Virtual machines and containers

In this section I describe the similarities and differences between virtual machines and containers. These technologies are used by couple AWS services described in the next chapter.

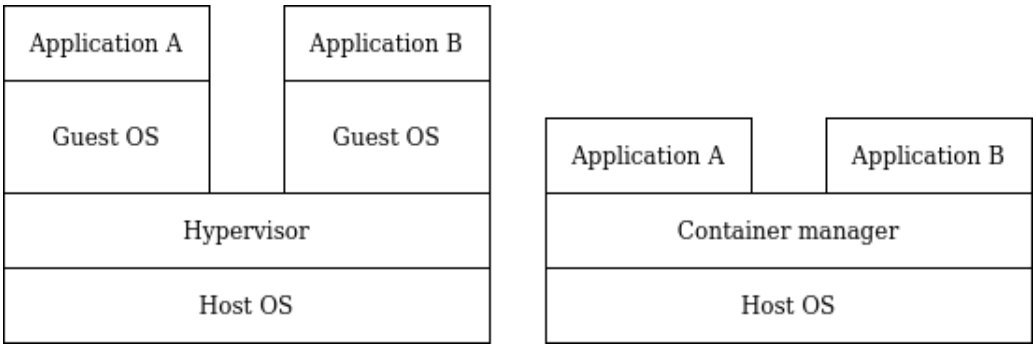


Figure 2.2: Simplified virtual machine stack on the left and a simplified container stack on the right.

Virtual machine is a piece of software that runs an operating system (guest machine) inside of another operating system (host machine). Virtual machines make it possible to emulate running many computers on the same hardware at the same time. The guest machine does not have direct access to the host machines resources and all access needs to go through a program called the hypervisor which does virtualization of the underlying hardware. Virtual machines are an example of virtualization which refers to making software act like hardware when interacted with by other programs (Bernstein, 2014).

Containers are slimmed down versions of virtual machines in which the containerized software can utilize features of the host machine's operating system's kernel. Software running in containers can be thought of as isolated user-space instances sharing the underlying operating system kernel (Bernstein, 2014). This means that containers running on same host machine can take up less memory than comparable virtual machines but they need to share the underlying operating system unlike virtual machines as can be seen in figure 2.2. Docker is one of the most popular container softwares (*Docker* 2020).

The benefit of running virtual machines and containers is the standardization of software on different hardware and operating systems as they let users deploy different applications requiring different environments on the same hardware. In the context of hosting it allows server vendors and cloud providers to host multiple applications with differing requirements on the same servers (Bernstein, 2014).

2.5 Random forests

Random forests are a group of supervised machine learning algorithms which construct decision trees that can be used for classification and regression purposes (Tin Kam Ho, 1995; Vapnik, 1999). In this section I will shortly explain the concept of supervised machine learning, classification and regression, followed by random forests in more detail. Random forests are useful background knowledge for understanding subsection 3.6.5.

Supervised machine learning is the process of training models with training data consisting of input-output pairs, and then using the trained models to predict the outputs of new input data. Supervised machine learning can be divided into many sub-classes like classification and regression (Vapnik, 1999; Tin Kam Ho, 1995).

Classification models try to predict the class of some piece of data. Classical example of this is pattern recognition in images, for example trying to figure out if an image contains a dog or a cat is a classification task. Regression models do not predict classes but rather real values like the predicted price of a house based on the input data (Vapnik, 1999).

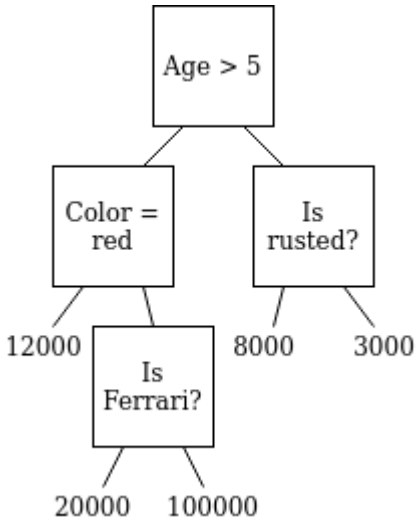


Figure 2.3: An example simplified decision tree used for predicting the price of used cars.

Random forests are a collection of differing decision trees. A decision tree is a tree data structure which contains prediction values in its leafs and decisions (comparisons) in it's non-leaf nodes. Figure 2.3 contains an example decision tree which models the price of used cars. At each non-leaf node there is a comparison made on the input data, when the answer is true the next comparison is made in the right children and vice versa. A random forest model's prediction for any input is the most popular prediction made by

all the different decision trees in that forest for that input value (Tin Kam Ho, 1995).

2.6 Time windows

A time window defines a continuous range in a time series, for example all points in time between one and two o'clock. Figure 2.4 shows the same time series where events E1, E2, E3, E4 and E5 happen at different points in time (t_1 - t_8) inside a tumbling time window on the top and a sliding time window on the bottom. The concept of time windows are a integral part of the old and new PHC implementations.

With tumbling time windows all the windows (W1, W2, W3) are distinctly separate from each other meaning that an event can only belong to a single time window. A tumbling time window is fully defined by the length of the windows. In the figure the sliding time windows have the same duration as the tumbling windows to highlight how the windows overlapping each other changes which windows the events belong to. Sliding time windows are defined by the duration of the windows and the time between creations of new windows (called granularity from now on). Sliding windows provide greater granularity than tumbling windows when the same windows duration is used for both of them.

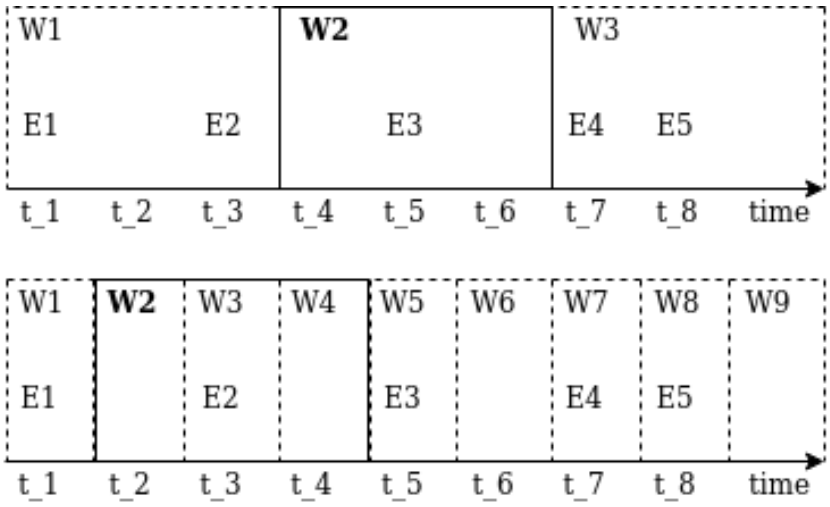


Figure 2.4: Tumbling and sliding time windows

3 Amazon Web Services overview

Amazon Web Services (AWS) is the worlds biggest and most popular cloud platform. It was first launched in 2004 by offering the Simple Queue Service (SQS), which was fully launched in 2006. Also, in 2006 AWS became a platform of services when the storage service S3 and the computing service EC2 were launched (*A Decade Of Innovation* 2016). Nowadays AWS offers over 175 different cloud services which range from storage solutions to computing, machine learning and networking (*What is AWS* 2020; *AWS vs Azure vs Google Cloud Market Share* 2020).

In this chapter I describe the AWS services and concepts that are necessary to know about to understand the rest of the thesis. These services include the storage service Amazon S3, the database services Amazon DynamoDB and Amazon RDS, the computing services Amazon ECS and AWS Lambda, the data streaming service Amazon Kinesis, the virtual network service Amazon VPC, the CDN service AWS CloudFront, the application monitoring service AWS CloudWatch and some others. Not every feature of each service discussed are described.

The features and pricing models described in this thesis are all based on the way things were in the spring and summer of year 2020. In the future new features might get added to AWS and the pricing models might change. The pricing models can be quite complicated and thus are explained at a level of detail required to understand and give context for the design decisions made in chapter 5. Not each service's pricing model is explained.

3.1 AWS concepts

In this section I describe general concepts about AWS which are important to understand to get a clear picture of the platform.

3.1.1 AWS global infrastructure

Amazon cloud is physically located all around the world in 24 Regions, 76 Availability Zones and over 200 Edge locations (*Regions and Availability Zones* 2020).

Amazon Regions are separate geographic areas like Northern California and Ireland. These Regions contain two or more distinct Availability Zones (AZ) which are data centers that are physically isolated from each other (max 100km) and connected via high speed links (*Regions and Availability Zones* 2020). AZs inside a Region are able to support each other in case of service failures. For example if the power of one AZ is cut off the traffic going there is going to get redirected and served from the other AZs of the Region. Edge locations work as the content caches for the AWS CDN named Amazon CloudFront (*Amazon CloudFront Key Features* 2020).

3.1.2 Security and permissions

Security in AWS works on a zero trust basis. To access any resource or execute any operation on them the principal entity trying to do so (be it an user or an application) has to have an explicit permission for that. Permissions are managed through AWS Identity and Access Management (IAM) policies (*Overview of Access Management: Permissions and Policies* 2020). Policies can be attached to user identities, identity groups, session identities, roles that services assume or even access control list among other things. These policies are usually lists of statements which can be written in JSON (*JSON* 2020) to define which operations on which resources the principal entities may perform.

3.1.3 Pricing model

Most AWS services have a pay-as-you-go model meaning that you only pay for what you end up using. Many services offer the possibility to reserve capacity by paying up-front. Reserved capacity is usually significantly cheaper than on-demand capacity (*AWS Pricing* 2020). Many services also have a volume based discount where the more you use them the less you end up paying for each individual unit of computing, storage etc. For example in a storage service the first n bytes per month might cost x for the customer, but the bytes after the first n might cost y per byte per month where $x > y$.

3.2 Amazon ECS and AWS Fargate

Amazon ECS (Elastic Container Service) is a Docker (*Docker* 2020) container management and orchestration service. Amazon ECS containers can be run as tasks which are a

collection of containers that should be ran together. For example a server container and a database container (*Amazon Elastic Container Service FAQs* 2020).

ECS container clusters can be run in EC2 or in AWS Fargate (*Amazon Elastic Container Service FAQs* 2020). Amazon EC2 (Elastic Cloud Compute) is a generic virtualized server. EC2 allows its users to provision re-sizeable computing capacity with different hardware specifications.

AWS Fargate is a service for running ECS containers in a serverless fashion. It removes the need for AWS clients to configure and maintain their EC2 instances. AWS Fargate can be run continuously like a container process on EC2 or in short batch jobs. Batch jobs run in response to events (see 3.7) which causes AWS Fargate to create a new container for the job and terminate it after it is done (*Amazon Fargate* 2020; *AWS Fargate FAQs* 2020).

AWS Fargate prices are calculated from the amount of computing and storage capacity the containerized applications consume during their lifetime rounded up to the nearest second. AWS Fargate bills the client for a minimum of one minute even if the task completes earlier, thus it should not be used with short running tasks (*Amazon Elastic Container Service pricing* 2020).

3.3 Amazon DynamoDB

DynamoDB (*Amazon DynamoDB* 2020) is a distributed, highly available NoSQL database service. It handles cluster scaling, replication and hardware provision automatically for the customer. DynamoDB tables can serve any amount of traffic with eventual consistency, as long as the tables are provisioned with enough capacity or if auto-scaling is enabled. DynamoDB can protect customer's data with backups and rollbacks.

3.3.1 DynamoDB keys, items and attributes

DynamoDB tables are schemaless, meaning that the individual item stored in DynamoDB can have any attributes. Only exceptions to this are the mandatory hash key (also called partition key), the optional range key (also called sort key) and the optional TTL (Time to live) attribute. The optional attributes can be configured on and off per table basis. TTL attributes (unix timestamp in seconds) are used to eventually expire items from the tables (*Amazon DynamoDB core components* 2020).

Each item in a table must have an unique primary key which is either the hash key or a composite primary key formed from the hash key and the range key. Items are located by running a hash key through a hash function, therefore the only way to access a specific item is by knowing that item's hash key or scanning the whole table (*Amazon DynamoDB core components* 2020).

Table 3.1: An example DynamoDB table containing product purchase information.

product	timestamp	price	c_phone	c_address	expire_time
12345	2015-12-21T17:42:34Z	150,0	1237163	221B Baker Street	1466271754
12345	2015-12-21T17:42:36Z	150,0	+2342342		1466271756
12345	2015-12-21T17:43:02Z	150,0	7653335		1466271782
21387	2016-01-02T12:43:12Z	5		Yliopistonkatu 3	1467290592

The example table 3.1 has been configured to use the attribute `product` as the hash key and the attribute `timestamp` as the range key. The TTL attribute is named as `expire_time`. Items are set to expire items six months after they have been added. Rest of the attributes can have any name and data type. Even when items share an attribute with the same name, those attributes do not have to have the same data type. For example `price` contains both floats and integers, and `c_phone` contains both integers and strings.

Choosing the right hash key is a very important design decision to make when working with DynamoDB. The hash key defines how items can be accessed directly, also internally DynamoDB partitions items to different distributed storage nodes based on the hash key. There should be enough variance between the hash keys to avoid bias in how the items are distributed between the different internal storages to avoid congestion when accessing the items (*Choosing the Right DynamoDB Partition Key* 2020).

DynamoDB support many kinds of attributes. There are the usual types like integers, floating point numbers, bytes, booleans and strings. More complex types include maps, sets and lists (*Amazon DynamoDB data types* 2020).

3.3.2 RCU and WCU

DynamoDB has two capacity modes: on-demand and provisioned of which provisioned is cheaper. In both modes DynamoDB automatically scales the read and write capacity up and down in response to incoming traffic. On-demand works best when the amount of traffic is unknown or changes unpredictably. Provisioned DynamoDB tables are configured to

have some base amount of capacity units all the time, thus working better with predictable work loads. These capacity units define how much I/O an DynamoDB table can support at any time. Write capacity units (WCU) define the maximum write throughput and read capacity units (RCU) define the maximum read throughput (*Amazon DynamoDB read/write capacity mode* 2020).

In provisioned mode customer's need to set minimum and maximum bounds for the RCU and WCU scaling and a target utilization percentage within that range. The RCU and WCU will be auto scaled inside these bounds. DynamoDB tries to scale the read and write capacities in such a way that the incoming traffic can be served and the average capacity usage is near the target utilization percentage. When a table does not have enough capacity to handle an incoming request the request will fail, this is called throttling. Throttling happens when there is a sudden and significant enough increase in the amount of incoming traffic or when the table cannot scale more as it has hit the upper bound of configured maximum capacity. This is why provisioned mode works best when there is a predictable amount of traffic (*Amazon DynamoDB read/write capacity mode* 2020).

3.3.3 DynamoDB operations

DynamoDB supports basic CRUD (create, read, update, delete) operations for individual items: `PutItem`, `GetItem`, `UpdateItem` and `DeleteItem`. Items can also be read in batches of 100 by `BatchGetItem` and created or deleted in batches of 25 by `BatchWriteItem` to improve latency between your application and DynamoDB (*Working with Items and Attributes* 2020).

New items can be inserted to a table as long as they have the required attributes (hash key and the optional ones) defined. Inserting an item will replace any existing item with the same primary key. In practice this means that multiple items can share a hash key or a range key but not the combination of them (if range key is used), and if only hash keys are used each one of them will be unique in the table.

Items are fetched with their hash keys. Multiple items with the same hash key may be returned when range keys are used, in this case the returned items are sorted based on their range key values (*Amazon DynamoDB core components* 2020). How the items are sorted depends on the data type of the range key (*Amazon DynamoDB data types* 2020). For example strings are sorted based on UTF-8 (*UTF-8* 2020) ordering.

Fetching data from table 3.1 with hash key `12345` using the `GetItem` operation would

return the first three rows of the table. These items would be ordered based on their range key values `timestamp`. In this case the range key ordering matches the order in the table from top to bottom.

The `Query` operation can only fetch items from a tables with composite primary keys (*Working with Queries in DynamoDB* 2020). `Query` operation supports more detailed queries than other operations through additional query parameters. For example the `KeyConditionExpression` parameter which can be used to define the hash key of the wanted item can also define the range key of the wanted item. When defining a range key the `KeyConditionExpression` supports boolean expressions like `count < 100` or `begins_with (timestamp, '2015-12-21T17:42')` and returns all matching items. The `ScanIndexForward` parameter is a boolean which defines whether the range key ordering happens in ascending or descending order. `Limit` defines how many matching items `Query` reads before stopping.

DynamoDB is distributed and thus follows the restrictions laid out in CAP theorem (Gilbert and Lynch, 2002). This means that DynamoDB is highly available and partition tolerant with the standard operations, but not consistent. To increase consistency it is also possible to use a transaction mode for most DynamoDB operations (*Amazon DynamoDB Transactions: How It Works* 2020). Using transaction mode for operations will make DynamoDB less available.

3.3.4 DynamoDB pricing

DynamoDB charges customers for the RCU and WCU used by the tables (*Amazon DynamoDB Pricing* 2020). Besides that additional costs are formed from backups, rollbacks, table replications and data transfers to other AWS regions.

For the purposes of this thesis I will only describe provisioned DynamoDB pricing in greater detail. In provisioned mode reading with `GetItem` and `GetBatchItem` can be eventually consistent, strongly consistent or transactional. Eventually consistent read operations consume one RCU for two items that are up to 4KB in size. Standard write operations consume one WCU per item up to 1KB in size, larger items require additional WCUs. Transactional writes and reads consume additional RCUs and WCUs (*Amazon DynamoDB Pricing* 2020; *Amazon DynamoDB read/write capacity mode* 2020).

`Query` operations does not consume RCU per item read but rather per the total amount of data read. It spends one RCU per 4KB of data read rounded up (*Amazon DynamoDB*

read/write capacity mode 2020).

Reserved capacity can be used with provisioned capacity for significant cost cuts. AWS clients can provision any amount of RCUs and WRUs in 100 unit chunks for 1 or 3 years in advance to get the capacity at a reduced cost. Any capacity scaling above the reserved capacity is paid with the standard provisioned price (*Amazon DynamoDB Pricing* 2020).

3.4 Amazon RDS

Amazon Relational Database Servise (RDS) is the relational database service in AWS. RDS allows clients to pick and choose between many popular database engines, for example PostgreSQL and MySQL (*PostgreSQL* 2020; *MySQL* 2020). RDS automates administrative work like updating software. Amazon RDS supports resizing the database storage and is highly scalable through techniques like read replicas (*Amazon RDS Features* 2020).

Amazon RDS supports automated backups, database snapshots and multi-AZ deployments with automated host replacements in case of a hardware failure, all of which makes RDS highly available and durable (*Amazon RDS Features* 2020). Multi-AZ deployments can be implemented with multiple masters or through read replica setups as explained in subsection 2.2.2. If any of the active replicas fail RDS will automatically failover to use the standby replicas to ensure availability of the database (*Amazon RDS Multi-AZ Deployments* 2020).

The performance characteristics of a RDS database are measured in input/output operations per second (IOPS). RDS can be configured to use a general purpose (SSD) storage or a provisioned IOPS (SSD) storage. The performance characteristics of a general purpose storage can auto-scale in response to usage to up to a maximum of 3000 IOPS. Provisioned IOPS gives the database a stable capacity of up to maximum of a 40000 IOPS regardless of the database usage (*Amazon RDS Features* 2020).

Amazon RDS costs consist of the following: RDS instance type, on-demand vs provisioned, storage size, backup recoveries, taking database snapshots and data transfer into and out of RDS. Like other AWS services RDS costs are significantly discounted if clients use reserved instances, these reservations are made for one or three years. RDS costs wary based on what database engine is used (*Amazon RDS Pricing* 2020).

3.4.1 Amazon Aurora

Amazon Aurora is a fully managed relational database built by AWS which is compatible with PostgreSQL and MySQL. AWS claims that it has better performance than a RDS instance running PostgreSQL or MySQL with only 1/10th of the cost. Aurora capacity can be provisioned in advance or run on a auto-scaling serverless architecture. It supports up to 15 read replicas across three AZs for high performance reads. By minimum it operates with six copies in three different AZs for over 99.99% availability. Aurora also support multi-master infrastructure with masters in multiple AZs (*Amazon Aurora FAQs* 2020).

Amazon Aurora's durability is guaranteed by storing backups in Amazon S3 and by automatically scanning databases for errors and auto-repairing them. Amazon Aurora can be multi-regional for reduced read latency (*Amazon Aurora FAQs* 2020).

Amazon Aurora supports parallel queries which are split to multiple Aurora databases to speed up query times in most cases. Using parallel query may incur higher I/O usage and thus increase Aurora costs (*Amazon Aurora FAQs* 2020).

3.5 AWS Lambda

AWS Lambda is a serverless event-driven computing service. Code run in AWS Lambda is the concern of the customer and almost everything else is taken care by AWS automatically. AWS Lambda can be used to create simple microservices or to stitch together complex cloud service architectures with custom logic.

AWS Lambda is always executed in response to some event like an item being added to a S3 bucket or another service directly invoking it (*AWS Lambda FAQs* 2020). Each AWS Lambda execution is isolated from the others and therefore "stateless" (Kablan et al., 2015). In this context "stateless" means that AWS Lambda can preserve state with tricks like storing the state in an external storage or by having subsequent AWS Lambda executions reuse some global variables (*Simple Caching in AWS Lambda Functions* 2019). AWS Lambda can auto scale by creating multiple running instances of itself. AWS Lambda is automatically run on multiple AZs to provide high level of availability and fault tolerance (*AWS Lambda Features* 2020).

AWS Lambda supports multiple programming languages like Java, Node.js and Python (*AWS Lambda FAQs* 2020). Any language can be supported by providing a custom runtime

(*AWS Lambda Features* 2020). Code run on AWS Lambda can do most things that a program run on a general purpose computer can do with the following restrictions: process stack tracing and networking have some restrictions, each execution of AWS Lambda has maximum of 500 MB of temporary disk storage, and AWS Lambda has a maximum of 15 minute TTL (*AWS Lambda FAQs* 2020). Third party dependencies are supported through popular dependency management tools the different languages have.

Like other serverless services AWS Lambda suffers from ”cold starts”. A cold start is the time it takes for the Lambda to get everything running from rest to process a request. Cold start time can greatly vary depending on the programming language and dependencies used. AWS Lambda can stay warm for a while after it was executed, which means that subsequent executions do not need to start from a cold state. Provisioned Concurrency can reduce cold start times to sub second by keeping functions warm for longer times than normal (*AWS Lambda FAQs* 2020).

AWS Lambda charges customers for number of invocations of the Lambdas, the total running times and the capacity assigned to the Lambdas. AWS Lambda capacity can be configured by changing the amount of RAM they can use. Any increase in RAM is proportionally applied to other capacities like CPU capability. AWS Lambda running times are always rounded up to the nearest 100ms. Using Provisioned Concurrency costs extra (*AWS Lambda Pricing* 2020).

3.6 Amazon Kinesis

AWS offers the Kinesis Data Stream (KDS) family of services for (near) real-time processing, forwarding and analysis of streamed data (*What is streaming data?* 2020). KDS services are managed by AWS and are resilient because of multi AZ synchronization. KDS services have one or more sources of data which can be an AWS services or an external applications (*Amazon Kinesis Data Streams FAQs* 2020). Varying from service to service there can be one or more destinations for the streamed data.

3.6.1 Kinesis record

Kinesis handles the data streamed through it as individual records, which can be buffered at different points in the streams. Records has a base64 (*The Base16, Base32, and Base64 Data Encodings* 2006) encoded `data` field which contains the raw data sent to Kinesis.

Records also have an unique `recordId` meta data field used to identify them (*Amazon Kinesis Data Firehose FAQs* 2020; *Amazon Kinesis Data Analytics FAQs* 2020).

Data streamed through Kinesis can be transformed and processed with AWS Lambdas before being forwarded (*Amazon Kinesis Data Firehose Data Transformation* 2020).

Kinesis records sent for transformation by Kinesis Firehose (subsection 3.6.2) or to a destination by Kinesis Analytics (subsection 3.6.3) need to be acknowledged with a status code and a `recordId`. If no acknowledgement is received or a failure status code is received the Kinesis service in question will try to resend the record. Different services handle the re-sending with different logic (*Amazon Kinesis Data Firehose FAQs* 2020; *Amazon Kinesis Data Analytics FAQs* 2020).

3.6.2 Kinesis Firehose

Amazon Kinesis Firehose is the simplest service in the Kinesis family. It is used to transport data in near real-time from one source to one or more destinations. Valid destinations are Amazon S3 and few different data warehouse services like Amazon Redshift (*Amazon Redshift* 2020). Kinesis Firehose is the service to use if real-time streaming is not required or if multiple destinations are required (*Amazon Kinesis Data Firehose FAQs* 2020)

Kinesis Firehose has a pay-as-you-go model where the costs accrue nearly linearly from the amount of data streamed through it, transforming streamed data adds additional costs. Kinesis record sizes are rounded up to the nearest 5 kB when calculating costs (*Amazon Kinesis Data Firehose pricing* 2020).

3.6.3 Kinesis Analytics

Kinesis Analytics is a service used to run applications on the data streamed through Kinesis. Kinesis Analytics application can be implemented as a Java application or with a special dialect of SQL which is designed to work on streamed data (*Amazon Kinesis Data Analytics FAQs* 2020). Unlike other Kinesis services the source for Kinesis Analytics has to be another Kinesis service. Valid destinations are Amazon S3, Kinesis Data Stream, AWS Lambda and few data warehouse services.

Kinesis Analytics processes incoming data in in-application streams which perform operations on each Kinesis record passing through them (*Amazon Kinesis Data Analytics FAQs* 2020). In-application streams can output the results to another in-application stream for

subsequent processing or to an output stream to pass the data to the stream destination. An example Kinesis Analytics application with one source stream and two destination streams can be seen in figure 3.1.

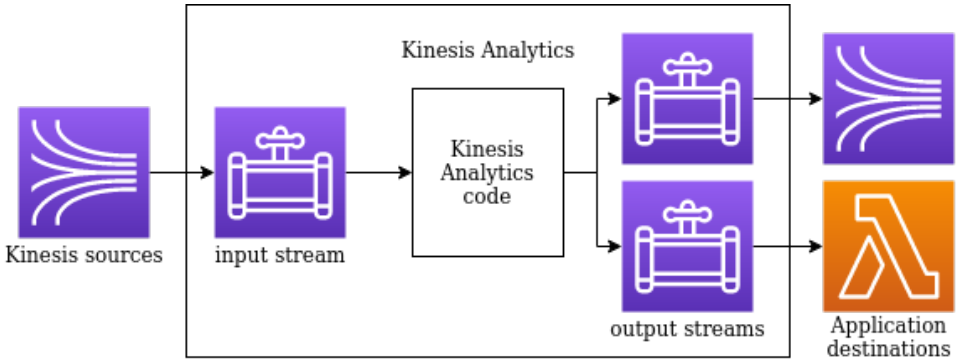


Figure 3.1: Example Kinesis Analytics application with one source and two destinations.

Kinesis Analytics applications have no upfront costs and auto-scale in response to incoming data (*Amazon Kinesis Data Analytics FAQs* 2020). Customers pay for average Kinesis processing units (KPU) usage per month. Each KPU provides Kinesis application with one virtual CPU and 4 GB of memory. Java applications always consume at least one KPU even if no data is streamed through them. Kinesis Analytics supports parallel execution of multiple instances of the applications. Parallelism will consume extra resources (*Amazon Kinesis Data Analytics pricing* 2020).

3.6.4 Streaming SQL

Kinesis Analytics SQL supports some concepts and operations that are not found in standard SQL specifications (*Streaming SQL Concepts* 2020). These concepts include in-application streams and pumps which define data flow sources and destinations inside a Kinesis Analytics application. Streams are defined like tables in standard SQL. `INSERT` and `SELECT` clauses can be used to add data into pumps. Additionally `WHERE` clause can be used to continuously filter data added to a pump. `JOIN` clause can be used to join multiple streams into one.

All in-application streams have `APPROXIMATE_ARRIVAL_TIME` and `ROWTIME` columns. `APPROXIMATE_ARRIVAL_TIME` is a timestamp containing the approximate time when the record was consumed by Kinesis Analytics. `ROWTIME` contains a timestamp about the time when the record was processed by the first in-application stream in Kinesis Analytics (*Streaming SQL Concepts* 2020).

ROWTIME can be utilized by windowed queries to continuously run SQL queries over tumbling or sliding time windows. Pumps can use the GROUP BY and ORDER BY clauses in windowed queries to aggregate and sort data (*Streaming SQL Concepts* 2020).

Figure 3.2: Example of streaming SQL for stock counting in one minute tumbling windows.

```
CREATE OR REPLACE STREAM "DESTINATION_SQL_STREAM" (
    "STOCK_NAME" VARCHAR(50),
    "STOCK_COUNT" INTEGER
);

CREATE OR REPLACE PUMP "DESTINATION_STREAM_PUMP" AS
    INSERT INTO "DESTINATION_SQL_STREAM"
        SELECT STREAM
            "STOCK_NAME",
            "PRICE",
            COUNT(*) AS "STOCK_COUNT"
    FROM "SOURCE_SQL_STREAM_001"
    WHERE "PRICE" > 15.0
    GROUP BY "STOCK_NAME",
        STEP("SOURCE_SQL_STREAM_001".ROWTIME BY INTERVAL '1' MINUTE)
    ORDER BY "STOCK_COUNT" DESC;
```

Figure 3.2 shows an example SQL code for counting the number of stocks in one minute tumbling time windows. DESTINATION_SQL_STREAM is the output stream of the application. DESTINATION_STREAM_PUMP gathers data from the default input stream SOURCE_SQL_STREAM_001. Columns STOCK_NAME and PRICE are selected from the input stream along with a matching row COUNT. Rows from input stream are filtered to rows where PRICE $>$ 15.0. The GROUP BY clause groups the data into rows by their STOCK_NAME and ROWTIME rounded up to the nearest minute with a STEP function. GROUP BY with a timestamp is how tumbling windows are expressed in streaming SQL. Finally the grouped rows are sorted in descending order by the number of stock in each row.

3.6.5 Anomaly detection with Kinesis Analytics

Kinesis Analytics can automatically detect anomalities in the data streaming through it with RANDOM_CUT_FOREST and RANDOM_CUT_FOREST_WITH_EXPLANATION SQL functions

(Tin Kam Ho, 1995; *RANDOM_CUT_FOREST_WITH_EXPLANATION* 2020; *RANDOM_CUT_FOREST* 2020).

`RANDOM_CUT_FOREST` calculates an anomality score for each row from all numeric column of that row. Kinesis Analytics constructs a n dimensional space of the data where n is the number of numeric columns. Any data point that is distant from other points in this space is considered anomalous. The underlying algorithm trains a random forest machine learning model to define which point are considered distant by partitioning the anomality score space with random cuts (*RANDOM_CUT_FOREST* 2020).

`RANDOM_CUT_FOREST` can be configured to work differently through different function parameters. `numberOfTrees` and `subSampleSize` can be used to control the accuracy of the estimations which in turn has an negative impact on the running times of the algorithms. As Kinesis Analytics work with streaming data the anomality detection takes into account the history of the data running through the stream. `timeDecay` defines how long data points will have an impact on the random forest model until they are considered stale and removed. `shingleSize` detects rapid changes in the streaming data by comparing up to last 30 data points to each other (*RANDOM_CUT_FOREST* 2020).

`RANDOM_CUT_FOREST_WITH_EXPLANATION` is similar to `RANDOM_CUT_FOREST` except besides labeling whole rows as anomalous it also returns a per column explanation on how the data on that column is anomalous. The explanation contains values for attribute score, directionality and strength. Attribute score tells how much a specific column contributes to the over all anomality score of the row. Directionality tells whether the attribute value is higher or lower than what the current trend is. Strength is value which signifies how sure Kinesis Analytics is about the value of the directionality attribute (*RANDOM_CUT_FOREST_WITH_EXPLANATION* 2020).

3.7 Amazon CloudWatch

Amazon CloudWatch is a service used to collect and monitor logging and metric data from other AWS services and applications. Amazon CloudWatch has automated tools to graphically and programmatically analyze the collected data for valuable insights. For example this information can be used for optimizing and debugging applications (*Amazon CloudWatch FAQs* 2020).

CloudWatch allows customers to create alerts which are triggered when a specified event

happens. Events can be configured to notify the customer or call other services when they are triggered. For example CloudWatch can be configured to shutdown a service it is monitoring and notify the customer when the monthly running costs of the monitored service exceed some threshold (*Amazon CloudWatch FAQs* 2020).

Amazon CloudWatch Logs Insights is a tool for querying application logs sent to Amazon CloudWatch. Querying is done through a purpose built language which supports picking matching parts of the logs with regular expressions (McNaughton and Yamada, 1960) and applying aggregation functions on them. Querying can be used to reduce the raw log data into a few useful data points (*Amazon CloudWatch FAQs* 2020). Logs Insights automatically parses valid JSON (*JSON* 2020) data into accessible fields which makes working with JSON data easier.

Figure 3.3: Example of CloudWatch Logs Insights query matching lines starting with the word *New*.

```
FIELDS @message
| PARSE @message "* *" as first_word, rest_of_the_line
| FILTER first_word = "New"
| PARSE @message "New payload received: *" as formatted
| DISPLAY formatted
```

Figure 3.3 shows an Amazon CloudWatch Logs Insights query which tries to match the log lines with a regular expression into variables `first_word` and `rest_of_the_line`. Then it filters out all the lines that do not start with the word *New*. Finally it displays the whole line with the message `"New payload received: "` prepended to the raw line.

3.8 Other services

In this section I will shortly describe some other AWS services not yet described. I will not go over their pricing as that is not important information in the context of the thesis.

3.8.1 Amazon S3

Amazon S3 (Simple Storage Service) is a highly available, highly durable object data storage capable of storing any amount of any kind of data. To achieve high availability and durability most data stored to S3 is automatically stored in multiple AZ's. Amazon S3 supports basic file system operations like creating, reading and deleting objects one

at a time or in batches. Amazon S3 organizes data into buckets which act like separate folders inside S3 (*Amazon S3* 2020).

3.8.2 Amazon VPC

Amazon Virtual Private Network (VPC) is a service for creating and provisioning virtual isolated networks inside the shared physical AWS infrastructure. Amazon VPC can be used to configure IP addresses, subnets, network protocols, NAT gateways (*Network Address Translation* 2020) and access control lists among many other things inside a VPC (*Amazon Virtual Private Cloud* 2020).

The main benefit of Amazon VPC are the security features and layers of protection it adds around cloud architectures. Amazon VPC is a safe way to connect multiple services running in AWS with each other. Amazon VPC can be used to monitor all the traffic happening inside the network (*Amazon Virtual Private Cloud* 2020).

3.8.3 Amazon CloudFront

Amazon CloudFront is the CDN service inside AWS infrastructure. It is closely integrated with many AWS services and most AWS of hardware (*Amazon CloudFront FAQs* 2020).

3.8.4 AWS ELB

AWS Elastic Load Balancing (ELB) splits incoming traffic to many targets inside AWS to make sure that no single target has to handle most of the traffic. AWS ELB can load balanced across or inside AZs (*Elastic Load Balancing* 2020).

There are three types of ELBs. Applications Load Balancers (ALB) which operate the HTTP/HTTPS level (URL) and are meant to load balance in the context of a VPC. Network Load Balancer operates on the transmission protocol level (TCP/UDP). Classic Load Balancer is a legacy load Balancer for EC2-Classic network (*Elastic Load Balancing* 2020).

3.8.5 Amazon SNS

Amazon Simple Notification Service (SNS) is a simple robust messaging service used communicate between publisher and subscriber applications inside many to many channels. These channels are called topics and they deliver the messages published to them by a publisher to all of their subscribers (*Amazon SNS FAQs* 2020).

Amazon SNS messages can be used to trigger AWS Lambdas or webhooks (*Webhook* 2020) etc. Messages can also be sent to end users with email, text messages etc. Amazon SNS supports many kinds of messaging protocols like HTTP/HTTPS, SMS etc (*Amazon SNS FAQs* 2020).

3.8.6 Amazon Athena

Amazon Athena is a Amazon S3 query service. The querying happens with standard SQL. Amazon Athena supports common data formats like CSV and JSON. For other types of data the customer can provide a schema or Amazon Athena may try to infer the schema (*Amazon Athena FAQs* 2020).

4 Project background

In this chapter I discuss the functionality, design, infrastructure of the old PHC and the adjacent services in order to describe the technical and business requirements for the new implementation. First I go over the infrastructure background and requirements for the whole project (subsections 4.1 - 4.3). After that I will describe different parts of PHC and adjacent services, and the requirements for them.

4.1 General business requirements

New PHC should be designed and developed with running costs in mind. If the new implementation costs less to run than the old one that can be considered a success. The end result should satisfy the technical requirements and it should not require much if any maintenance in the future. The code base and infrastructure should be built in such a way that new features are easy to develop and deploy in the future if they are needed.

The new solution will have to integrate with other services in the company infrastructure. Only newer services should be used as there is a plan to deprecate some of the older services in the future to decrease operating costs and technical complexity of the infrastructure.

In this chapter these business concerns are implicitly a part of all the technical requirements even if they are not directly mentioned in each section.

4.2 General technical requirements

The architecture must be hosted in AWS. The page hit data has to be calculated from the event data received by EP. New PHC should work externally mostly in the same fashion as the old one. Unlike old PHC the one needs to be able to support all different brands of the company.

New PHC should use a technology stack similar to other modern services in the company. This means that I should favor serverless solutions like AWS Lambda for computing and Amazon DynamoDB or Amazon Aurora for storage.

4.3 Page hit counter and adjacent services

PHC calculates page hit sums for individual assets and serves requests asking for those sums. Page hits are requested for, calculated and stored in different tumbling time windows (see 4.4). Old PHC stores hits in a relation database. Old PHC is one part of a bigger service called Legacy User Content Service (UCS). Raw page hit data is sent to PHC through Legacy Back End (BE) when users trigger page hit events in websites and mobile applications. The processed data served by PHC is used to construct content popularity listings. PHC is located in a private network (Amazon VPC), so any requests to it have to go through a proxy service called Core Service (CS) which has a public facing API.

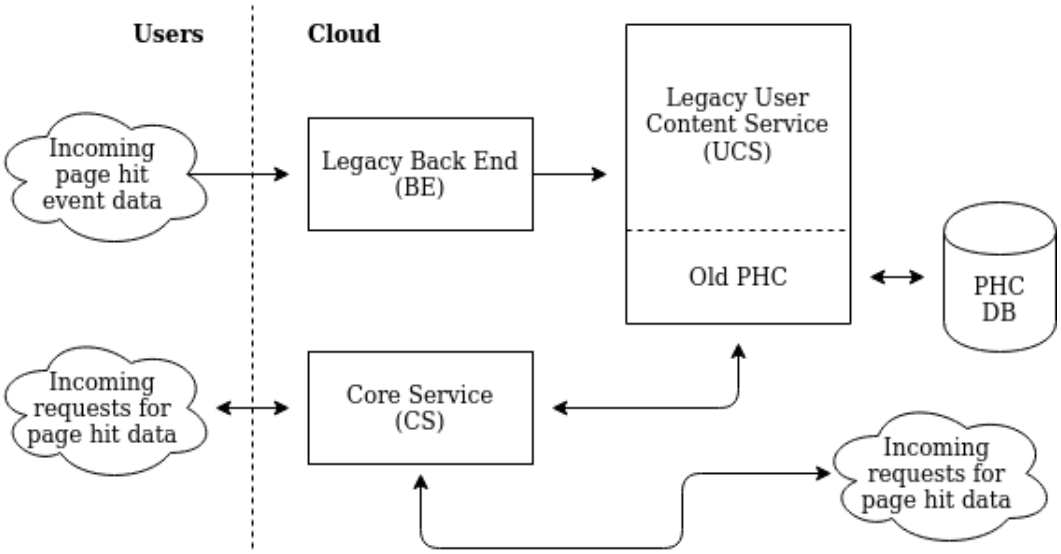


Figure 4.1: Overview of old PHC architecture

4.3.1 Requirements for page hit counter and adjacent services

The new PHC architecture has a requirement to replace Legacy Back End (BE) with Event Processor (EP) in the existing pipeline. BE is a legacy application which should be deprecated, therefore any new service should not use it. EP has access to almost all the data needed for page hit counting so it makes sense to use that existing functionality to implement new PHC. EP is described in section 4.8.

The functional requirements for new PHC are the following:

- Process the incoming page hit event data coming from EP

- Count the page hits for individual assets over different time windows
- Store the page hit counts
- Serve page hit count requests coming from CS

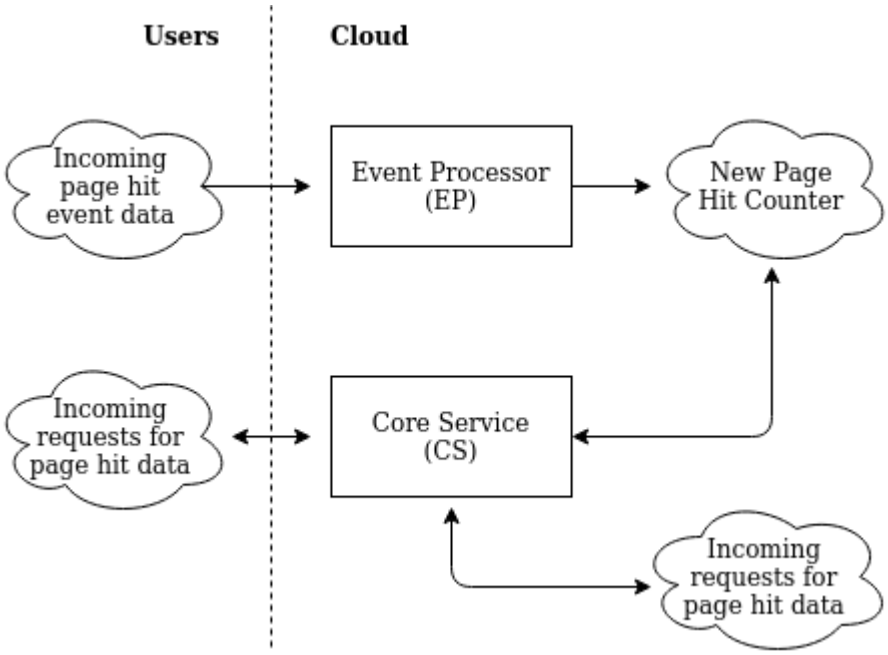


Figure 4.2: Overview of the new PHC architecture with the adjacent services

4.4 Time windows in page hit counter

In old PHC the most popular content listings that are displayed on the websites and mobile applications contain page hit counts from one of three periods: past five minutes, past day and past week. These content listings do not need to be updated in real-time so tumbling time windows are generally sufficient, but a most popular listing for the past week which is updated only once per week is a bit too vague to be useful during most of the week. This is why old PHC calculates the one week listing from tumbling windows with one day granularity. Same is true for five minutes listing and tumbling windows with granularity of one minute and one day listing and tumbling windows with granularity of one hour.

4.4.1 Requirements for time windows

New PHC should support these the same three periods, but the underlying time windows and granularities do not have to be the same. Other time windows and granularities can be chosen as long as the most popular asset listings are updated at a pace which keeps the contents of the popular listings relevant for the users. This is not a well defined requirement, and thus making the time window lengths and granularities configurable at later times would be ideal. Another thing to take into account with the choice of time windows is their impact of running and storage costs of the system.

4.5 Page hit counter API

The old PHC can be queried over a HTTP API with url query parameters (*Query String* 2020). PHC will return the page hits matching the query parameters.

Any combination of the following parameters can be defined in a HTTP request:

```
period: "justnow" (past 5 min) or "day" (past day) or "week" (past week)
resourceType: string
section: number
paidType: string
```

`period` defines which period's hits are wanted. By default `justnow`.

`resourceType` defines which type of assets' hits are wanted. If none are defined any type will do. If multiple are defined the asset has to match one type.

`section` defines in which section the asset has be in. If none are defined any section will do. If multiple are defined the asset has to match one section.

`paidType` defines which type of assets' hits are wanted. If none are defined any type will do. If multiple are defined the asset has to match one type.

4.5.1 Old PHC API call analysis

I analyzed old PHC API traffic by running Amazon Athena SQL queries on the AWS ALB logs inside the company Amazon VPC (*Querying Application Load Balancer Logs* 2020).

This traffic data informed some of the design decisions made, as can be seen in subsection 6.2.1.

Figure 4.3 shows the query I ran on the UCS ALB logs to find all the unique request urls called with GET method in year 2020 along with the number of times they were called. I continued processing this data further with Python scripts (*Python* 2018) by filtering out non-PHC request and then reducing the remaining data to query parameters and the matching percentage of requests. I got the results seen in table 4.1.

```
SELECT request_url, COUNT(1) as total_requests
FROM prod
WHERE YEAR = '2020'
    AND elb_status_code = '200'
    AND request_verb = 'GET'
    AND domain_name LIKE 'ucs-prod.com'
    AND request_url LIKE 'https://ucs-prod.com'
GROUP BY request_url
```

Figure 4.3: Amazon Athena SQL querying Amazon ALB for information about UCS API calls.

From table 4.1 it can be seen that a specific `paidType` is not requested very often. `section` and `resourceType` are used approximately as often, and a `period` is always present. The table does not show it but `paidType` and `resourceType` request parameters almost always had only a single type defined if they were present. `section` request parameters had 13 sections defined on average if it was present in a request. Approximately half of the request with `section` defined had only one section defined.

Table 4.1: Number of UCS API calls by query parameter combinations.

query parameter combination	percentage of requests
`period&resourceType&paidType`	1,3%
`period&paidType`	5,4%
`period&resourceType`	6,0%
`period&section`	6,9%
`period`	15,2%
`section&resourceType&sectionId`	65,1%

From this data I can also see that this service receives millions of calls per month which is information that is useful when comparing different design choices in 5.4.1.

4.5.2 Requirements for page hit counter API

The new page hit counter should implement the same API as the old one with the addition of being able to support any brand of the company. Old PHC was brand specific meaning that there was one PHC instance running for each brand of the company's products, new PHC will have one instance shared between brands.

The most popular listings shown in the websites and mobile applications show between 5 to 100 list items. New PHC does not need to send back more than 100 page hit results back to the requester even if there are more.

4.6 Incoming event data

Incoming data are sent to old PHC in HTTP POST requests as batches of page hit objects encoded in JSON. Each page hit object has the following structure.

```
Pagehit = {
    assetId: long;
    hits: integer;
    sectionId: integer;
    resourceType: string;
    paidType: string;
}
```

4.6.1 Requirements for event data model

The data model should contain at least the same data as the old model along with some information to distinguish between the different brands. Other than that there is no specific requirements for new PHC event data model as long as the required functionality of the service can be fulfilled with it.

4.7 Page hit database

Old PHC uses a relational database to store the page hit counts.

```
CREATE TABLE page_hit (
  id BIGSERIAL NOT NULL PRIMARY KEY,
  hit_amount BIGINT,
  asset_id BIGINT,
  resource_type VARCHAR(256),
  section_id BIGINT,
  hit_stmp TIMESTAMP with time zone,
  paid_type VARCHAR(32),
  unique (asset_id, resource_type, section_id, hit_stmp, paid_type)
);
```

Figure 4.4: Old PHC database schema for periodical page hits.

Figure 4.4 shows the table schema that the different page hit periods share. There is a different table with a different name for each period. Page hits stored in these tables follow the granularity rules described in section 4.4.

Page hits are always fetched for a specific period. For period `justnow` the hits are found in the rows with the timestamp in the past five minutes. Same is true for period `day` and the past 24 hourly timestamps, and period `week` and the past 7 daily timestamps. Filtering might be done in SQL if extra constraints are part of the request.

Page hits are stored with an upsert operation. Upsert inserts a new page hit if there is no existing row matching the unique constraint (`asset_id`, `resource_type`, `section_id`, `hit_stmp`, `paid_type`). Upsert updates the `hit_amount` if a row with the same unique constraint already exists.

Page hits are automatically deleted after they have become stale by rolling out of the active tumbling time windows.

4.7.1 Requirements for page hit database

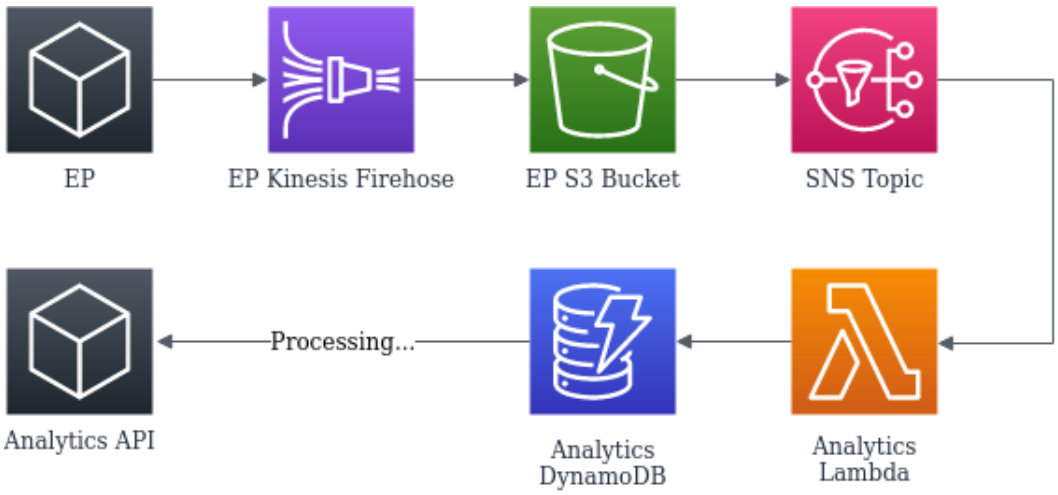
The use case of the database does not require any specific storage solution. The company prefers Amazon DynamoDB and Amazon Aurora as storage solutions so the choice should be done between them (*Amazon DynamoDB* 2020; *Amazon Aurora FAQs* 2020). The chosen technology needs to be able to support the same functionality as the relational database in old PHC: select data with different constraints, upsert page hits and get rid of stale data.

The choice of storage technology, method and schema can greatly impact the running and storage costs of new PHC, thus a lot of thought should be put into the decision.

4.8 Event Processor

Event Processor (EP) acts as a public consumer API for the event data sent by the websites and mobile applications. These events contain all sorts of data such as information about page visits which are gathered for analytics purposes. The page visit event data can be used almost as it is for the page hit calculations.

Figure 4.5: AWS data pipeline from Event Processor to Analytics API.



EP is part of a bigger analytics pipeline shown in figure 4.5. EP forwards the processed events to a Amazon Kinesis Firehose responsible for events of that type, which in turn forwards them to a corresponding S3 bucket for intermediate storage (*Amazon Kinesis Data Firehose FAQs* 2020; *Amazon S3* 2020). New items arriving to a S3 bucket trigger a corresponding Amazon SNS topic which messages the AWS Lambdas subscribed to that topic to process the new item (*Amazon SNS FAQs* 2020; *AWS Lambda Features* 2020). AWS Lambdas then pull the item from the S3 bucket and insert it into a DynamoDB table (*Amazon DynamoDB* 2020). The rest of the analytics pipeline uses data from the DynamoDB to serve incoming request to Analytics API.

4.8.1 Requirements for Event Processor

New PHC is required to use EP as its source of data. This way BE does not need to act as the source of page hit data, thus taking the architecture one step closer to deprecating BE. Page visit event data forwarded by EP needs to contain all the fields required by PHC. At the moment EP does not have any information about the `resourceType` or the `brand` of the asset to which the page visits belong to, this has to be changed by integrating EP with CS which has more information about the assets.

4.9 Spam filtering

Historically spamming has referred to sending unsolicited messages to one or more recipients, usually with the goal of scamming the recipient or advertising something. In the context of the PHC system our interest lies in event data spam. Event data spam can be generated by a user repeatedly reloading the same web page or by a bot sending data directly to EP. This sort of event data does not represent the normal usage of the websites or the mobile applications and therefore it skews the gathered event data. To make sure that the event data used for page hit counting is authentic I am interested in trying to identify and filter out this sort of spam.

To filter spam it must be first identified and then separated from valid event data. Identifying spam can be done through many means ranging from simple lists of blocked users and request origins, all the way to complex machine learning models. For this use case the latter is more interesting as the event data does not necessarily have a static source.

4.9.1 Requirements for spam filtering

Try to filter out event data spam. The filtering can be done anywhere in the pipeline between events getting generated and processed into storable page hits.

5 Design comparisons

In this chapter I describe the different designs that fit the requirements laid out in chapter 4. I discuss PHC system in smaller *parts* with the intention of making it easier to understand and compare the different design choices. These *parts* were decided on by studying the design of the old PHC along with the project requirements.

5.1 PHC parts and design problems

For each incoming page hit PHC needs to process the event, recalculate page hit counts and store them. For each incoming request PHC has to parse the request, fetch the correct hits from the page hit storage and send them back to the requester. PHC also has to periodically run a maintenance task to drop stale data from the storage. These processes can be abstracted into the following *parts*:

- Page hit `counter` calculates page hits.
- Page hit `consumer` stores the page hit data.
- Page hit `storage` is the storage solution, includes the storage schema.
- Page hit `maintainer` drops stale hits from the storage.
- Page hit `API` serves incoming requests by fetching the right data from storage and sending it to the requester.

All valid designs for new PHC have to have the right components to take care of the tasks of these *parts*. It is possible for one component to take care of the tasks of multiple *parts* in some designs.

Here are some questions all the different designs need to answer:

1. At which point in the EP pipeline should the event data be read from?
2. Which AWS services to use for the implementation? (Constrained by the wishes of the company and the estimated running costs.)

3. How to keep the hit counts up to date in the current time window? (Adding new hits and removing old ones.)

4. What kind of database and schema to use? (Needs to be able to allow service to serve all possible requests, see 4.5.2.)

5.2 Delivering data to PHC

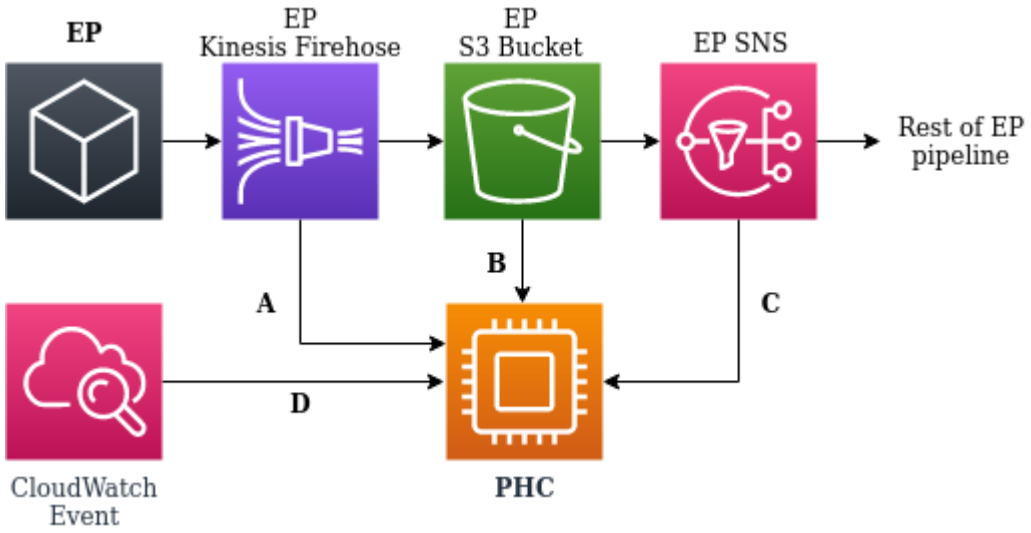


Figure 5.1: Possible ways to integrate PHC with the EP pipeline.

Before looking further into the implementation of the different *parts* I had to think about how to integrate PHC into the EP pipeline. The letters A-D in figure 5.1 correspond to four different ways of sending the page hit event data to PHC. Options A-C are part of the existing EP pipeline and thus they would not require any extra work. Here are the four options explained:

(A) Set PHC as one of the destinations for EP Kinesis Firehose.

(B) Set PHC to trigger in response to new files being uploaded to EP S3 bucket.

(C) Subscribe PHC to a EP SNS topic that forwards messages of new page hit items arriving in the EP S3 bucket.

(D) Create a timed CloudWatch Event which triggers PHC. After being triggered PHC can read all the new items from the EP S3 bucket.

Options A-C are very similar and there does not seem to be any advantage in picking any specific one. In the context of this project option C makes the most sense as it is in line with the rest of the EP pipeline design which receives it's data from the EP SNS topic. The EP SNS topic would send the data to PHC in a batch once every couple minutes, because it takes a while for new data to reach the S3 bucket as EP and EP Kinesis Firehose do some buffering of the data.

5.3 AWS services as components

In this section I introduce the designs that fit the requirements of the different *parts* for the new PHC implementation.

Table 5.1: The AWS services which can be used to implement specific parts of new PHC.

`Counter`	AWS Lambda, AWS Fargate, Amazon Kinesis Analytics
`Consumer`	AWS Lambda, AWS Fargate
`Storage`	Amazon DynanoDB, Amazon Aurora
`Maintainer`	AWS Lambda, AWS Fargate, Amazon DynanoDB, Amazon Aurora
`API`	AWS Lambda, AWS Fargate

Table 5.1 contains the necessary *parts* to construct the new PHC along with the AWS services that could be used to implement them. The choices are to use AWS Lambda or AWS Fargate for computing and Amazon DynamoDB or Amazon Aurora for storage with the additional choice of utilizing Kinesis Analytics for counting and spam detection (*AWS Lambda FAQs* 2020; *AWS Fargate FAQs* 2020; *Amazon DynamoDB* 2020; *Amazon Aurora FAQs* 2020; *Amazon Kinesis Data Analytics FAQs* 2020).

5.3.1 Computing with AWS Lambda

The AWS Lambda computation would be split into two different Lambdas: processing Lambda and API Lambda. These Lambdas would be triggered either by the EP integration triggering the processing of new page hit event data or by incoming API requests coming from CS. Processing the new page hit events could be triggered by option C or D of the figure 5.1.

Option D requires the `consumer` to contain additional logic to figure out which event data in the EP S3 bucket was added there since the last time the `consumer` was triggered.

This would cause problems for AWS Lambda as they are ”stateless” (*AWS Lambda FAQs* 2020). The previous state has to be either stored somewhere (the S3 bucket would work) or the the AWS Lambda has to be called at predestined times to make it possible to find the new page hit data.

A potential upside with option D is that it could be used to control when the AWS Lambda gets triggered. The CloudWatch Event could for example trigger once every five minutes (which is the length of the shortest time window in PHC) to update the data in the active tumbling time windows (*Amazon CloudWatch FAQs* 2020). At the same time there is no obvious upside in doing page hit counting like this compared to consuming the hits in batches as they are sent by the EP SNS topic. Option D does not seem to offer any tangible benefits over option C which is simpler in nature and does not require extra logic in the AWS Lambda code.

`maintainer` work could be left for the database as doing it as a part of either the processing or the API computations would introduce extra complexity to the system and increase the likelihood of introducing bugs into the code base.

5.3.2 Computing with Kinesis Analytics

Additional design choice to consider with the AWS Lambda approach is whether to add a Kinesis Analytics application to the pipeline in between the EP Kinesis Firehose and the PHC to act as the `counter`. Kinesis Analytics has in-built functionality for working with streaming data in tumbling time windows which suits PHC's use case very well (*Streaming SQL Concepts* 2020). The Kinesis Analytics application would be implemented with streaming SQL rather than Java as that one is more expensive to run (*Amazon Kinesis Data Analytics pricing* 2020).

Kinesis Analytics is capable of doing anomaly detection on the streaming page hit event data which could potentially help with the spam detection and filtering. Using a Kinesis Analytics application would decouple the counting from the rest of the PHC implementation and could make it easier to change the counting logic in the future (*Amazon Kinesis Data Analytics FAQs* 2020).

Kinesis Analytics costs will not be significant as the flow of event data coming from EP is steady which should make it possible to consistently run the application with low number of vCPUs (*Amazon Kinesis Data Analytics pricing* 2020).

5.3.3 Computing with AWS Fargate

The AWS Fargate based solution would be a standard server side application continuously running and handling the functionality of all the parts except the `storage`. It would receive the page hit event data from EP by subscribing to a SNS topic containing the page hit event data (*Amazon SNS FAQs* 2020). There is no upside to using option D of 5.1 if the AWS Fargate service is a monolith containing the functionality of both the page hit consuming and the API in the same application. If the application is split into two parts then the upsides and downsides for using option D described in 5.3.1 apply here too. In any case the `API` would get called directly by CS.

Implementing all the required functionality for `counter`, `consumer`, `maintainer` and `API` in a standard server side application would not be too complicated. The application would parse the incoming patches of data and upsert them into the `storage`. It could have an internal state for managing the maintenance of the `storage` (unless the `storage` does it itself). The `API` would be a standard HTTP API. Overall the implementation would be very similar to the old PHC.

5.3.4 Storage options

PHC needs a real database to act as the page hit `storage`. The options are Amazon DynamoDB and Amazon Aurora as described in section 4.7.1 (*Amazon DynamoDB* 2020; *Amazon Aurora FAQs* 2020).

Amazon Aurora could utilize the existing database schema as it is. Amazon Aurora does not have a simple in-built way to drop stale data, but it could be done with a SQL trigger that drops old data whenever new data is added.

Amazon DynamoDB would require some schema design to make it work as the page hit `storage`. DynamoDB has in-built mechanism for dropping old data with TTLs.

The page hit data is very cacheable as it does not change until it becomes stale, thus the `storage` I/O usage should be minimized with memoizaition of the queries (Norvig, 1991). The fewer different queries PHC makes the higher the cache hit rate will be.

5.4 Comparing the AWS service costs

Earlier in this chapter I described how it is possible to use both the AWS Lambda or AWS Fargate to implement the computing parts of new PHC. In the same way the `storage` and `maintainer` can be implemented either with Amazon DynamoDB or Amazon Aurora. To make the decision between which ones to pick I need to compare the projected costs of each of these approaches.

I will not be using the real cost estimates as those are company secrets. However, I will demonstrate how the price differences can be evaluated by comparing generic setups which are analogous to the setups that would be used for the real project implementation.

5.4.1 AWS Lambda vs AWS Fargate

I will run the cheapest AWS Lambda setup possible which is the one with the smallest memory size of 128MB and no additional services. The work done by the processing service and the API service should not need any more memory. The pricing for such a setup is 0.0000002083$ per 100ms of Lambda execution (min running time is 100ms) and additional 0.20$ per 1 million requests (*AWS Lambda Pricing* 2020).

I compare the AWS Lambda setup against the cheapest AWS Fargate setup which has 0.25 vCPU and 0.5GB of RAM. The costs for running this setup are $0.04048\$ * 0.25$ per hour for the 0.25 vCPU plus $0.004445\$ * 0.5$ for the 0.5GB storage per hour. AWS Fargate costs are calculated per second of running with a minimum running time of one minute per invocation (*AWS Fargate Pricing* 2020).

Figure 5.2 shows comparison of costs between AWS Lambda and AWS Fargate running batch jobs over a month (30 days) where the service is called once every 1,5 minutes which corresponds approximately to how often the `consumer` part needs to be run. That means there are $60/1,5 * 24 * 30 = 28800$ computations per month. The X-axis shows different running times in milliseconds and the Y-axis shows the corresponding cost over a month assuming all executions take the same time which is of course not the case in reality. Even if the calculation is just a rough estimation one can clearly see that the AWS Lambda is much cheaper for these sort of batch jobs as long the running times stay under one minute. The running time will depend on the size of the page hit event data input to the computing service.

I can also make another comparisons against an AWS Fargate instance running continu-

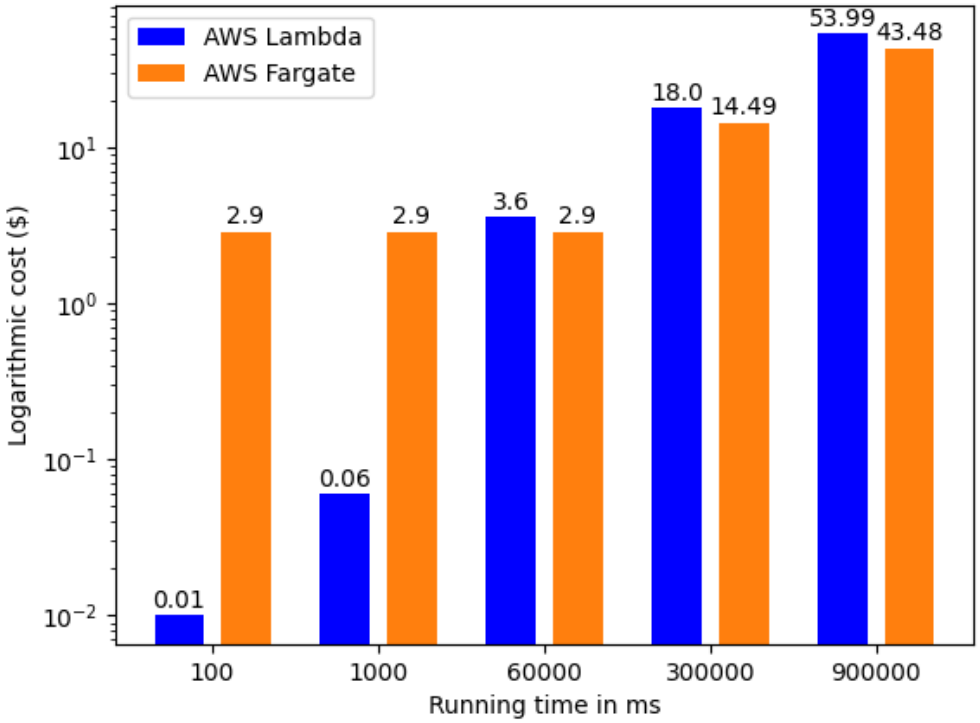


Figure 5.2: Cost comparison between the cheapest AWS Lambda and AWS Fargate setups for processing page hits.

ously which is how the `API` service would have to work as it needs to respond to queries quickly. The cost is $0.01980394 * 0.25 * 30 * 24 * 60 + 0.00217462 * 0.5 * 30 * 24 * 60 \approx 260,85$ per month. Comparing the approximate cost to the different AWS Lambda costs shown in 5.3 shows that either the execution times need to be very long or there needs to be an unrealistic amount of traffic (compared to the traffic amount based on 4.5.2) before AWS Fargate is clearly cheaper for the `API`.

In practise running a monolithic AWS Fargate service might make the most sense. In that case even if one were to forget about the AWS Fargate costs in 5.2 the AWS Lambda would seem cheaper still. Additionally, I remain sceptical of the service quality of the least expensive AWS Fargate setup. As such the cost estimation for the monolith AWS Fargate service are probably lower than they would be in reality.

Any way one considers the cost aspect of AWS Lambda versus AWS Fargate it seems that AWS Lambda is always cheaper and therefore the better choice.

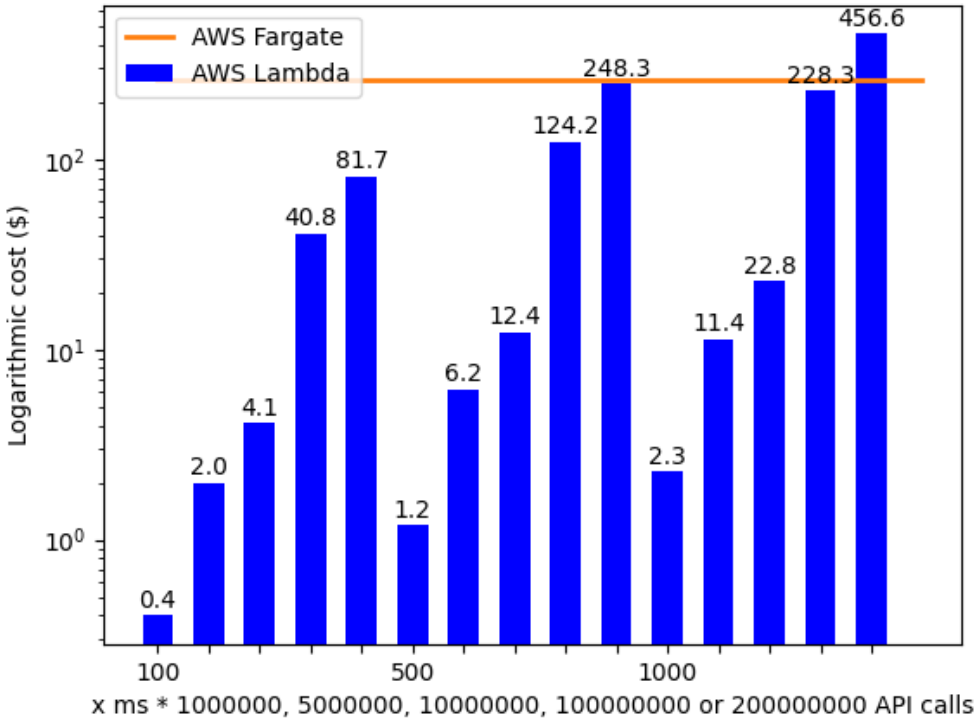


Figure 5.3: Cost comparison between the cheapest AWS Lambda and AWS Fargate setup for API.

5.4.2 Amazon DynamoDB vs Amazon Aurora

In this section I will introduce an provisioned Amazon DynamoDB setup which I estimate to have enough capacity to perform as the page hit storage without throttling. Then I will show that an equivalent Amazon Aurora setup will cost more and thus Amazon DynamoDB is more cost effective. I do the comparisons with fabricated I/O levels for both the storage reads and writes. Importantly these I/O levels have the same fraction of reads to writes as the estimated I/O in the production system which is heavily slanted towards reads. The number of incoming read requests will be 20 per second and the number of incoming write requests is 1 per second. The `storage` does not need to store that much data as it should be continuously dropping stale data, thus I will use 3 GB as the maximum amount of storage required.

An important factor to keep in mind is the caching of the `storage` queries which can greatly impact the I/O required. I will not take the caching costs into account as they are similar for both of the services. To maximize the cache hit rate the `storage` queries should be as homogeneous as possible. This kind of uniform querying does not leverage

the capabilities of relational databases and thus suits NoSQL databases better.

I estimate that with caching Amazon DynamoDB needs approximately 250 RCU and 50 WRU to safely support the incoming traffic. The queries do not need to be transactional. With 3 GB of storage this will cost approximately 55$ per month (*Amazon DynamoDB Pricing* 2020).

I estimate that with caching the Amazon Aurora instance consumes about 50 IOPS. The cheapest *db.t3.medium* instance type on-demand would costs about 85$ per month. If the capacity was provisioned in advance for three years the costs would drop all the way down to 25$ per month. That might be something to think about in the future but at the time of designing the system I do not have the data required to safely provision the database for one or three years in advance (*Amazon RDS Pricing* 2020). There is also the question around if the cheapest instance type would be able to serve the traffic in the production environment, I think not. Therefore the Amazon Aurora cost estimate might be lower than it should be.

Taking into account the desire to memoize the page hit queries and the estimated running costs Amazon DynamoDB seems like the better choice.

6 Implementation

In this chapter I will describe the design of the implementation of the new PHC. Additionally I will describe some of the challenges I ran into during the implementation along with the changes to the design that were motivated by those challenges. Figure 6.1 shows the overview of the chosen design. The box labeled PHC is the new page hit counter service.

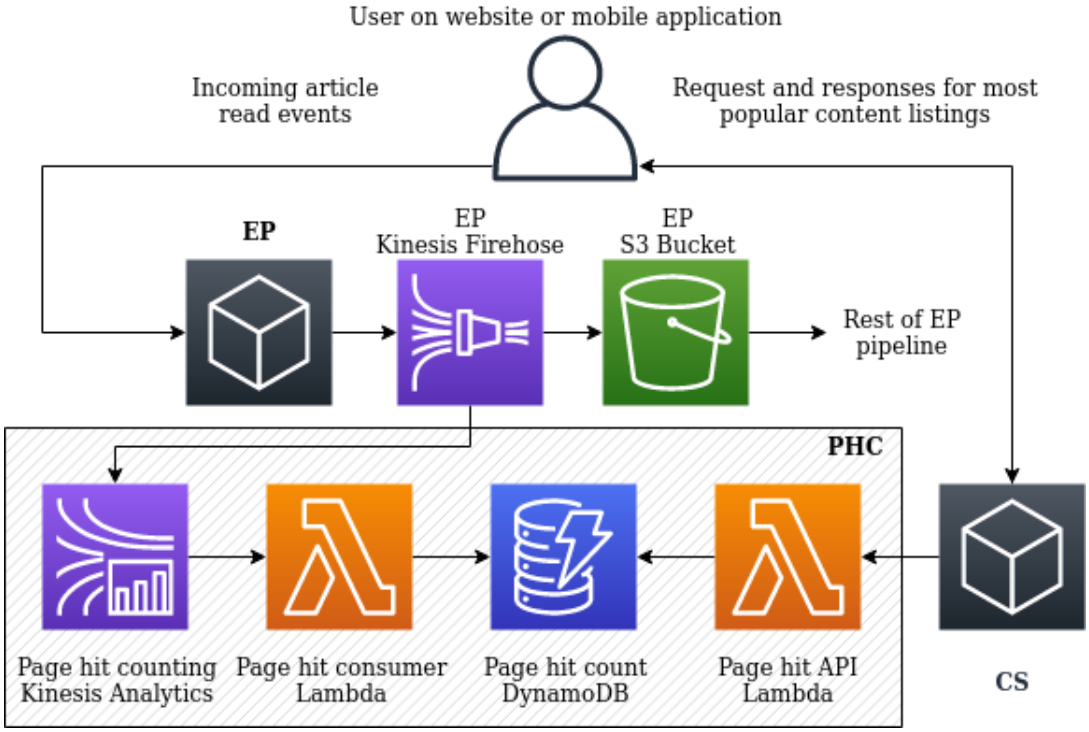


Figure 6.1: New PHC pipeline with EP and CS integrations.

New PHC is formed from four pieces:

1. *Counter*: Amazon Kinesis Analytics application (*Amazon Kinesis Data Analytics FAQs* 2020) which continuosly counts the page hits in different tumbling time windows.

2. *Consumer*: AWS Lambda (*AWS Lambda FAQs* 2020) which inserts the precalculated hit counts to Amazon DynamoDB (*Amazon DynamoDB* 2020).

3. *Storage*: Amazon DynamoDB without additional services. Amazon DynamoDB handles the dropping of stale data.

4. *API*: AWS Lambda which parses incoming requests, reads data from DynamoDB and returns the results matching the request.

I did not have time to implement page hit event spam filtering. Spam filtering is further discussed in chapter 8.

6.1 Counter

Counter is a Amazon Kinesis Analytics application that receives its input data from the EP Kinesis Firehose. The input data is directed to three in-application streams which are counting page hit counts over the three required time periods. From the in-application stream the data is sent to a corresponding destination stream whenever the current time window in that stream finishes (*Amazon Kinesis Data Firehose FAQs* 2020; *Amazon Kinesis Data Analytics FAQs* 2020).

Kinesis Analytics does not allow multiple in-application streams to use a single AWS Lambda as a shared destination. To get around this the *Consumer* Lambda is given three different aliases which are used as the destinations. This set up makes the architecture a bit more complex. However, it does not add additional costs as most of AWS Lambda costs are based on the total execution time of all the Lambda executions which does not change whether there are three ”separate” ones or just a single one.

The streaming SQL in 6.2 shows the implementation of the five minute time window in Kinesis Analytics. The SQL code passes through the required attributes and adds data about the time period in question along with the hit count. The period codes are used by the *Consumer* AWS Lambda to distinguish between different time windows. The two other time windows are implemented in the same fashion with the following changes: the interval time and period are changed to `INTERVAL '1' HOUR` and period `'H'` for the time window of one day, the same is true for `INTERVAL '1' DAY` and period `'D'` for time window of one week. The windows lengths were chosen according to the requirements described in 4.4.

The hit count are calculated for each unique combination of `brand`, `assetId`, `sectionId`, `paidType`, `resourceType` and `timestamp` (period time window timestamp). This grouping matches the grouping in the old PHC database schema with the addition of `brand`. Rows lacking `resourceType` are filtered out.

Figure 6.2: Streaming SQL for page hit counting in five minute tumbling windows.

```
CREATE OR REPLACE STREAM "FIVE_MIN_DESTINATION_SQL_STREAM" (
    "brand" VARCHAR(4),
    "period" VARCHAR(1),
    "assetId" BIGINT,
    "sectionId" INTEGER,
    "paidType" VARCHAR(20),
    "resourceType" VARCHAR(50),
    "hitCount" INTEGER,
    "timestamp" TIMESTAMP
);

CREATE PUMP "FIVE_MIN_AGGREGATED_SQL_PUMP" AS
    INSERT INTO "FIVE_MIN_DESTINATION_SQL_STREAM"
    SELECT STREAM
        "brand",
        'F' AS "period",
        "assetId",
        "sectionId",
        "paidType",
        "resourceType",
        COUNT(*) AS "hitCount",
        STEP("PAGEHITS_SOURCE_001".ROWTIME BY INTERVAL '5' MINUTE)
            AS "timestamp"
    FROM "PAGEHITS_SOURCE_001"
    WHERE "resourceType" IS NOT NULL
    GROUP BY
        "brand",
        "assetId",
        "sectionId",
        "paidType",
        "resourceType",
        STEP("PAGEHITS_SOURCE_001".ROWTIME BY INTERVAL '5' MINUTE);
```

6.2 Storage

Page hit counter DynamoDB follows a schema where each row has the attributes: `hashKey`, `rangeKey`, `sectionId`, `resourceType`, `paidType` and `expirationTime`. The `hashKey` and `rangeKey` attributes are explained in subsections 6.2.1 and 6.2.2. DynamoDB is queried with the `Query` operation which can utilize the additional range key attribute (*Working with Queries in DynamoDB* 2020). `expirationTime` is the TTL attribute marked as a UTC+0 integer which is set to ten minutes after the last point in time when the page hit record in the row is still in an active time window. Table 6.1 shows an example of the sort of rows that might be in the DynamoDB.

Table 6.1: Example of page hit storage table contents.

hashKey	rangeKey	sectionId	resourceType	paidType	expirationTime
abc#F#2020-05-06T11:05	00000258#132141398	563	article	free	1588764000
def#H#2020-05-06T11:00	00001187#387294989	1347	recipe	paid	1588767000
abc#D#2020-05-06T00:00	00012349#983447323	34	advertorial	metered	1588810200

6.2.1 Hash key

For each valid Kinesis record processed by *Consumer* Lambda one row is added into DynamoDB. The hash key is built from the brand name, the period and the timestamp concatenated together separated by `#` characters. `#` is used as it is not found in any of the hash key parts. For example one hash key might be `xyz#D#2020-06-12T12:00`. This hash key format was chosen as it allows PHC to answer all possible queries quite efficiently by post-processing the fetched rows.

I debated on whether to have single hash key or multiple different hash keys per page hit record to optimize the querying of the data. These keys would contain information about the page hit record's `resourceType`, `section` and/or `paidType` inside the hash keys in different combinations. In the end I decided against this and went for the simpler approach because of the following reasons.

Firstly, a multiple hash keys schema would have increased the number of keys per page hit record from one up to maximum of eight depending which kinds of combination keys are supported. For example writing the base key (the hash key without anything else besides brand, period and timestamp) and all the combinations of `resourceType`, `section` and

`paidType` would have meant that for each page hit record there would have been eight hash keys as shown in formula 6.1.

$$1 + \binom{3}{1} + \binom{3}{2} + \binom{3}{3} = 1 + 3 + 3 + 1 = 8 \tag{6.1}$$

Secondly, according to data described in subsection 4.5.1: a specific `paidType` is not requested regularly and `section` is often requested with multiple values. When request have multiple sections defined PHC either has to read with the hash keys for all the sections in the request, or with a hash key with no section and filter out the rows with wrong sections after fetching the data. For requests with a couple of sections reading the data per section hash key might be a good idea, but for requests with more sections this would cause the RCU usage to increase significantly. Additionally, when reading with the sectionless hash keys there is the added benefit of reading the data with a descending page hit count order. A downside to this approach is that DynamoDB always uses the same amount of RCU when looking for specific sections which might increase the amount of RCU usage for many requests when compared to being able to search for a specific section.

Thirdly, utilizing heavy caching with the simplified schema should cut off a significant amount of the RCU usage when fetching the page hits from *Storage*, as fewer unique hash keys in the cache leads to higher chance of a cache hit.

6.2.2 Range key

`rangeKey` is the hit count padded with zeroes to length eight concatenated with the `assetId` and separated by `#`. This hash and range key schema makes sure that each key combination created is a unique primary key so that later page hits do not overwrite earlier ones.

The range key makes it possible to fetch hits from the table in a descending order based on the hit counts. This works because the rows are sorted character by character based on their range key's UTF-8 string values when the range keys are the same length. To make sure that the keys are the same length the page hits value which is an integer is stored as a string and prepended with zeroes. The `assetId` in a page hit records always has the same length in characters.

To make sure that the hash key and range key created a unique primary key I had to add

the `assetId` in the primary key. `assetId` could not be put into the hash key because then there would not be a way to fetch specific rows without knowing the `assetId` in advance. Also, I did not want to make the range key solely out of the `assetId` as I want to fetch the page hit records from DynamoDB in descending page hit order. By concatenating these values both of the goals can be achieved.

6.3 Consumer

Page hit *Consumer* AWS Lambda is triggered when Kinesis records are sent to it by *Counter* when one of the tumbling time windows finishes. *Consumer* stores records to DynamoDB after they are validated and parsed into a form that fits the table schema.

Kinesis Analytics expects a response containing a list of processing results for each record it sends forward. *Consumer* responds to a record either with an `Ok` or a `DeliveryFailed` status code, depending on whether the record was successfully processed or not. Records that have a `DeliveryFailed` status code get resent to *Consumer* after a short wait time. I use this property of Kinesis as the retry mechanism for storing records to DynamoDB in case `PutItem` fails for some records. Malformed records which cause a validation error get an `Ok` status code so that they are not resent to *Consumer* thus avoiding an endless loop.

During development of new PHC I analyzed *Consumer* application logs with Amazon CloudWatch Logs Insights and noticed that a small portion of the assets had most of the page hits. This is significant because the most popular content listings only care about maximum of 100 most popular assets (see 4.5.2). These facts lead to the idea of optimizing the costs of writing to and reading from DynamoDB by filtering out records with small hit counts as they do not contribute to the overall popular content listings in a meaningful way.

The cutoff point for filtering vary per time period as assets in longer time windows tend to get more hits. I manually tuned the cutoff points and compared the query results from the old PHC and new PHC to make sure that the results still matched. This turned out to not work that well as the number of hits changes throughout the day. I changed to using a simpler filter which filters out all the records that have hit counts lower than the hit count of the top 500th asset ordered by hit count. This approach saves a lot of DynamoDB WCU and RCU and the query results are good when compared to the old PHC.

6.4 API

Page hit *API* answers requests based on the request parameters. It validates and then parses the request into correct DynamoDB hash keys to fetch the rows containing the requested page hits or the rows that could contain the requested page hits which are then post processed to find the relevant rows.

For example the request:
`<phc-domain>?brand=asd&period=day&resourceType=article&section=121` made at 2020-06-01 16:32 would create the following hash keys to fetch data.

`asd#H#2020-06-01T15:00`, `asd#H#2020-06-01T14:00` all the way back to `asd#H#2020-05-31T16:00`.

The fecthed data is then filtered down to the rows where the `resourceType` is article and `sectionId` is 121.

There are multiple caches in the way from client to PHC API. *API* caches rows in a LRU cache when DynamoDB is accessed. CS caches responses from PHC *API* for a short time. CloudFront CDN caches data when CS responds to requests. The reason for having so many caches is to lower the computing costs of the whole architecture. Figure 6.3 shows how data is cached in the path of a request going to PHC through CS.

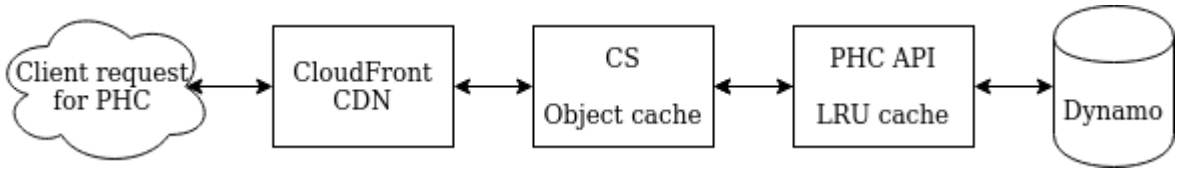


Figure 6.3: Client to new PHC API caching.

PHC HTTP responses contain the header field `Cache-Control: max-age=X` which defines the maximum length of time a response returned by it should be cached for. PHC varies the duration of the `max-age` based on the period defined in the request as seen in table 6.2.

DynamoDB is wrapped inside a LRU cache in the PHC API so that whenever data is fetched from DynamoDB the cache is first checked for a cache hit. In case of a cache miss the data is read from DynamoDB and stored into the LRU cache for subsequent requests. Using the cache greatly reduces the amount of read requests made to DynamoDB. The cache is an in-memory cache utilizing the excess memory that the AWS Lambda executions

Table 6.2: PHC API Cache-Control: max-age per period.

period	max-age
justnow	0,5min
day	2min
week	5min

have. The in-memory caches do not get garbage collected between executions as the *API* is called very often (*Simple Caching in AWS Lambda Functions* 2019).

Amazon CloudFront is sensitive to the order in which the query parameters appear in requests when looking for CDN cache hits. It is therefore very important to make sure that all calls to PHC use the same ordering of the query parameters (*Caching Content Based on Query String Parameters* 2020).

7 Measurements and results

In this chapter I describe performance measurements taken from the production data of the new PHC. Additionally I compare the monthly running costs of the old and new PHC systems, the exact numbers are sensitive information and therefore will not be revealed. For the sake of making the comparison I will use the average total monthly running costs of the old system from the past year as the number which I compare the running costs of the new system against.

7.1 Performance measurements

Figure 7.1 shows the five minute average of provisioned WCU (top line) and consumed WCU (bottom line) of the PHC DynamoDB from 10/7/2020 to 16/7/2020. From the figure it can be seen that the consumed WCU peaks heavily during the night when all the data from the daily tumbling window is written into the table. This does not cause a problem as DynamoDB is able to auto-scale quickly enough to avoid throttling the writes.

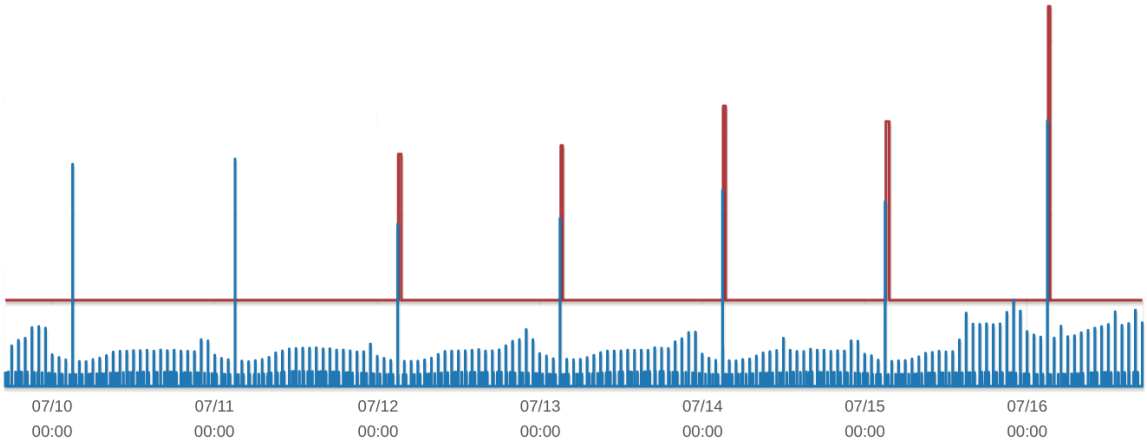


Figure 7.1: New PHC DynamoDB WCU usage from week 29 of 2020.

The RCU usage of the DynamoDB is negligible as we can have almost 100% cache hit rate with our API Lambdas instances' in-memory caches. The high cache hit rate is possible because the page hit data does not change after it has been written and therefore it does not have to be re-read by the same Lambda execution after it has read it once. This is

very good from cost performance standpoint as it reduces the DynamoDB running costs significantly while allowing us to utilize the memory capacity of the Lambdas to the fullest.

Figure 7.2 shows the five minute average execution duration of the PHC consumer Lambda from 10/7/2020 to 16/7/2020. We can see from the figure that the execution times vary based on the time of the day, but otherwise they are quite uniform.

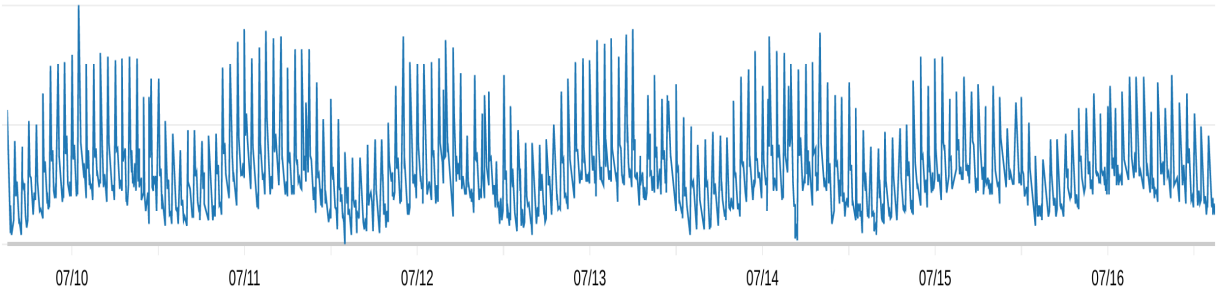


Figure 7.2: New PHC consumer Lambda execution durations from week 29 of 2020.

The API execution durations are stable with the occasional outlier taking much longer than the average time. The number of concurrent executions follows the behavior of RCU consumption of the DynamoDB meaning that the execution times stay short and there is not a reason to run an increasing number of API Lambdas in parallel.

The Kinesis Analytics application workloads follow the same kind of fluctuation as the Consumer Lambda throughout the day. Happily this fluctuation does not have an impact on the amount of vCPUs used by the Kinesis Analytics application which would increase the running costs.

7.2 Results

The running costs of the old PHC are calculated from a portion of total UCS running costs, this portion is as big as the portion of the traffic going to old PHC out of the all traffic coming to UCS. This is done to approximate the running costs of the old PHC part of UCS. I did not take into account the BE costs in the same way as it is complicated to figure out how those costs relate to PHC. This means that the cost estimates for the old PHC are smaller than they are in practice. The cost comparison was done between the average monthly running cost of old PHC from 1.9.2019 to 30.4.2020 and the total running cost of new PHC from the month of September in 2020.

The running costs for the new system turn out to be approximately 40% of that of the

old PHC. While the result numbers are not exact they still show that the stated goal of having the running cost of the new system be smaller than the running cost of the old one was achieved. Additionally, the goal of moving PHC from UCS to a separate service and thus bringing UCS and BE one step closer to being fully deprecated was completed.

8 Conclusions

In this thesis I have shown how to implement a page hit counter system by using Amazon Kinesis, AWS Lambda and Amazon DynamoDB. This system consumes streaming event data in real time and serves an API for querying the pre-calculated counts of different assets in different tumbling time windows.

The research questions laid out for the work in chapter 1 were the following:

1. What is the best way to implement the new PHC service with the stated requirements and goals in mind?
2. How well does the new implementation perform when compared to the old one in running costs and other metrics?
3. Did the new service manage to satisfy the stated requirements and goals?

Research question 1. was answered in chapter 5 with the component and design comparisons made there. The combination of Amazon Kinesis Analytics, AWS Lambda and Amazon DynamoDB matched the project requirements the best out of the different options considered.

Research question 2. was answered in chapter 7 with strong results showing that the running cost of the new PHC are less than half of that of the old one.

Research question 3. is answered in this paragraph. The new PHC implementation meets all of the requirements laid out in chapter 4 with the exception of the spam filtering functionality.

8.1 Future work

Even though I managed to satisfy the stated requirements and goals almost in full there is still some work left for the future. In this chapter I will look into different improvements that could be made to new PHC DynamoDB. I will also discuss event data spam filtering.

8.1.1 DynamoDB

DynamoDB writes could be optimized as they account for most of the DynamoDB running costs. The primary problem with the WCU usage of the current architecture is that the writes are always done in burst when a tumbling time window finishes. This behavior can be seen in the figures in subsection 7.1. One easy way to reduce the bursts would be to add small delays in between items being written to DynamoDB.

To reduce both WCU and RCU consumption DynamoDB data could be compressed before it is stored and decompressed after it is read. This would reduce the amount of RCU consumed by `Query` operations (*Amazon DynamoDB read/write capacity mode* 2020). In practise this might not bring significant cost savings as the cache hit ratio in the API Lambda is high and thus reducing the size of the read items does not matter much.

In the future after other problems have been dealt with and enough data about the DynamoDB table capacity consumption has been gathered I will look into reducing the running costs by using reserved capacity for the table (*Amazon DynamoDB Pricing* 2020).

I have to be careful to not underprovision the write capacity of the table when doing the optimization work as that could cause throttling which can lead to the Kinesis record processing in the consumer Lambda to start failing. The consumer Lambda failing will in turn cause the Kinesis Analytics application to resend the records which can lead to a snowballing effect. This was something I ran into during the implementation of the project. The snowballing effect can cause both Kinesis Analytics and DynamoDB to partly fail for up to ten minutes, thus preventing the system from writing down the newest page hits.

8.1.2 Spam filtering

I did not have the time to implement the page hit event data spam detection and filtering with Kinesis Analytics like discussed in chapter 5 or by other means. This is something that needs to looked into in the future.

The following are some other ideas around the topic of detecting and filtering the spam.

1. More data validation could be added to the consumer Lambda.
2. The events sent by the websites and the mobile applications could include meta data that is hard to counterfeit so that EP or PHC could recognize real user events from fake ones.

3. Anomalous data detected by Kinesis Analytics could be sent into a separate S3 bucket for further analysis.

4. Anomalous data detected by Kinesis Analytics could be filtered out once there is confidence that the detection works well.

Since `RANDOM_CUT_FOREST` and `RANDOM_CUT_FOREST_WITH_EXPLANATION` can only deal with numeric data I want to encode the non-numeric attributes of page hit data into numeric data. So that Kinesis Analytics can count the anomality score from a greater number of features per page hit event data sent from EP to PHC and thus hopefully reach a better quality of anomality detection (*RANDOM_CUT_FOREST_WITH_EXPLANATION* 2020; *RANDOM_CUT_FOREST* 2020).

References

5 key takeaways about the state of the news media in 2018 (2018). https://www.pewresearch.org/fact-tank/2019/07/23/key-takeaways-state-of-the-news-media-2018/. (Visited on: 2020-04-13).

A Decade Of Innovation (2016). https://perspectives.mvdirona.com/2016/03/a-decade-of-innovation/. (Visited on: 2020-06-15).

Adzic, G. and Chatley, R. (2017). "Serverless Computing: Economic and Architectural Impact". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, pp. 884–889. ISBN: 9781450351058. DOI: 10.1145/3106237.3117767.

Amazon Athena FAQs (2020). https://aws.amazon.com/athena/faqs/?nc=sn&loc=6. (Visited on: 2020-06-23).

Amazon Aurora FAQs (2020). https://aws.amazon.com/rds/aurora/faqs/?nc=sn&loc=6. (Visited on: 2020-07-07).

Amazon CloudFront FAQs (2020). https://aws.amazon.com/cloudfront/faqs/?nc=sn&loc=5&dn=2. (Visited on: 2020-06-07).

Amazon CloudFront Key Features (2020). https://aws.amazon.com/cloudfront/features/?nc=sn&loc=2. (Visited on: 2020-05-25).

Amazon CloudWatch FAQs (2020). https://aws.amazon.com/cloudwatch/faqs/. (Visited on: 2020-06-23).

Amazon DynamoDB (2020). https://aws.amazon.com/dynamodb/. (Visited on: 2020-04-12).

Amazon DynamoDB core components (2020). https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html. (Visited on: 2020-06-10).

Amazon DynamoDB data types (2020). https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_AttributeValue.html. (Visited on: 2020-06-10).

Amazon DynamoDB Pricing (2020). https://aws.amazon.com/dynamodb/pricing/. (Visited on: 2020-06-16).

Amazon DynamoDB read/write capacity mode (2020). https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html. (Visited on: 2020-06-10).

Amazon DynamoDB Transactions: How It Works (2020). https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html. (Visited on: 2020-06-11).

Amazon Elastic Container Service FAQs (2020). https://aws.amazon.com/ecs/faqs/. (Visited on: 2020-07-02).

Amazon Elastic Container Service pricing (2020). https://aws.amazon.com/ecs/pricing/. (Visited on: 2020-07-06).

Amazon Fargate (2020). https://aws.amazon.com/fargate/. (Visited on: 2020-07-02).

Amazon Kinesis Data Analytics FAQs (2020). https://aws.amazon.com/kinesis/data-analytics/faqs/. (Visited on: 2020-04-05).

Amazon Kinesis Data Analytics pricing (2020). https://aws.amazon.com/kinesis/data-analytics/pricing/. (Visited on: 2020-07-23).

Amazon Kinesis Data Firehose Data Transformation (2020). https://docs.aws.amazon.com/firehose/latest/dev/data-transformation.html. (Visited on: 2020-06-12).

Amazon Kinesis Data Firehose FAQs (2020). https://aws.amazon.com/kinesis/data-firehose/faqs/. (Visited on: 2020-06-17).

Amazon Kinesis Data Firehose pricing (2020). https://aws.amazon.com/kinesis/data-firehose/pricing/. (Visited on: 2020-06-12).

Amazon Kinesis Data Streams FAQs (2020). https://aws.amazon.com/kinesis/data-streams/faqs/. (Visited on: 2020-04-05).

Amazon RDS Features (2020). https://aws.amazon.com/rds/features/. (Visited on: 2020-07-07).

Amazon RDS Multi-AZ Deployments (2020). https://aws.amazon.com/rds/features/multi-az/. (Visited on: 2020-07-07).

Amazon RDS Pricing (2020). https://aws.amazon.com/rds/pricing/. (Visited on: 2020-07-07).

Amazon RDS Read Replicas (2020). https://aws.amazon.com/rds/features/read-replicas/. (Visited on: 2020-07-09).

Amazon Redshift (2020). https://aws.amazon.com/redshift/. (Visited on: 2020-05-21).

Amazon S3 (2020). https://aws.amazon.com/s3/faqs/. (Visited on: 2020-06-10).

Amazon SNS FAQs (2020). https://aws.amazon.com/sns/faqs/. (Visited on: 2020-07-06).

Amazon Virtual Private Cloud (2020). https://aws.amazon.com/vpc/. (Visited on: 2020-07-06).

Application programming interface (2020). https://en.wikipedia.org/wiki/Application_programming_interface. (Visited on: 2020-04-05).

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (Apr. 2010). "A View of Cloud Computing". In: *Commun. ACM* 53.4, pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672.

AWS Fargate FAQs (2020). https://aws.amazon.com/fargate/faqs/?nc=sn&loc=4. (Visited on: 2020-07-17).

AWS Fargate Pricing (2020). https://aws.amazon.com/fargate/pricing/. (Visited on: 2020-07-17).

AWS Lambda FAQs (2020). https://aws.amazon.com/lambda/faqs/. (Visited on: 2020-06-16).

AWS Lambda Features (2020). https://aws.amazon.com/lambda/features/. (Visited on: 2020-06-16).

AWS Lambda Pricing (2020). https://aws.amazon.com/lambda/pricing/. (Visited on: 2020-06-16).

AWS Pricing (2020). https://aws.amazon.com/pricing/. (Visited on: 2020-05-25).

AWS SES FAQs (2020). https://aws.amazon.com/ses/faqs/. (Visited on: 2020-07-19).

AWS Snowmobile (2020). https://aws.amazon.com/snowmobile/. (Visited on: 2020-07-19).

AWS vs Azure vs Google Cloud Market Share (2020). https://www.parkmycloud.com/blog/aws-vs-azure-vs-google-cloud-market-share/?utm_medium=referral&utm_source=medium&utm_campaign=medium%20blogs&utm_content=aws-vs-azure-vs-google-cloud-market-share. (Visited on: 2020-04-05).

Azure global infrastructure (2020). https://azure.microsoft.com/en-us/global-infrastructure/. (Visited on: 2020-07-11).

Bernstein, D. (2014). "Containers and Cloud: From LXC to Docker to Kubernetes". In: *IEEE Cloud Computing* 1.3, pp. 81–84.

Caching Content Based on Query String Parameters (2020). https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/QueryStringParameters.html. (Visited on: 2020-07-06).

Caching Overview (2020). https://aws.amazon.com/caching/. (Visited on: 2020-07-10).

Castro, P., Ishakian, V., Muthusamy, V., and Slominski, A. (2017). "Serverless Programming (Function as a Service)". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2658–2659.

Choosing the Right DynamoDB Partition Key (2020). https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/. (Visited on: 2020-07-11).

Content Distribution Network (2020). https://www.akamai.com/uk/en/resources/content-distribution-network.jsp. (Visited on: 2020-06-30).

Data Privacy FAQ (2020). https://aws.amazon.com/compliance/data-privacy-faq/. (Visited on: 2020-07-19).

Docker (2020). https://www.docker.com/. (Visited on: 2020-07-02).

Economies of scale (2020). https://en.wikipedia.org/wiki/Economies_of_scale. (Visited on: 2020-04-25).

Elastic Load Balancing (2020). https://aws.amazon.com/elasticloadbalancing/. (Visited on: 2020-07-06).

Gilbert, S. and Lynch, N. (June 2002). "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *SIGACT News* 33.2, pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601.

Hayes, B. (July 2008). "Cloud Computing". In: *Commun. ACM* 51.7, pp. 9–11. ISSN: 0001-0782. DOI: 10.1145/1364782.1364786.

Here's How The Way We Read Newspapers Has Changed (2020). https://www.huffpost.com/entry/newspaper-evolution-change-infographic-digital-twitter-online_n_5367077. (Visited on: 2020-04-13).

Heroku (2020). https://www.heroku.com/. (Visited on: 2020-06-30).

JSON (2020). https://en.wikipedia.org/wiki/JSON. (Visited on: 2020-05-01).

Kablan, M., Caldwell, B., Han, R., Jamjoom, H., and Keller, E. (2015). "Stateless Network Functions". In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. HotMiddlebox '15. London, United Kingdom: Association for Computing Machinery, pp. 49–54. ISBN: 9781450335409. DOI: 10.1145/2785989.2785993.

Leavitt, N. (2010). "Will NoSQL Databases Live Up to Their Promise?" In: *Computer* 43.2, pp. 12–14.

McNaughton, R. and Yamada, H. (1960). "Regular Expressions and State Graphs for Automata". In: *IRE Transactions on Electronic Computers* EC-9.1, pp. 39–47.

Mirkovic, J. and Reiher, P. (Apr. 2004). "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms". In: *SIGCOMM Comput. Commun. Rev.* 34.2, pp. 39–53. ISSN: 0146-4833. DOI: 10.1145/997150.997156.

MongoDB (2020). https://www.mongodb.com/. (Visited on: 2020-07-19).

MySQL (2020). https://www.mysql.com/. (Visited on: 2020-07-07).

Network Address Translation (2020). https://en.wikipedia.org/wiki/Network_address_translation. (Visited on: 2020-07-06).

Norberg-Schultz Hagen, S., Verne, G., and Bratteteig, T. (2020). ""All Celebrities and Sports on Top": Prototyping Automation for and with Editors". In: *Proceedings of the 16th Participatory Design Conference 2020 - Participation(s) Otherwise - Volume 1*. PDC '20. Manizales, Colombia: Association for Computing Machinery, pp. 22–32. ISBN: 9781450377003. DOI: 10.1145/3385010.3385025.

Norvig, P. (Mar. 1991). "Techniques for Automatic Memoization with Applications to Context-Free Parsing". In: *Comput. Linguist.* 17.1, pp. 91–98. ISSN: 0891-2017.

Overview of Access Management: Permissions and Policies (2020). https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction_access-management.html. (Visited on: 2020-05-25).

Overview of Cloud Bigtable (2020). https://cloud.google.com/bigtable/docs/overview. (Visited on: 2020-07-19).

PostgreSQL (2020). https://www.postgresql.org/. (Visited on: 2020-07-07).

Python (2018). https://www.python.org/. (Visited on: 2020-06-23).

Query String (2020). https://en.wikipedia.org/wiki/Query_string. (Visited on: 2020-06-09).

Querying Application Load Balancer Logs (2020). https://docs.aws.amazon.com/athena/latest/ug/application-load-balancer-logs.html. (Visited on: 2020-06-23).

RANDOM_CUT_FOREST (2020). https://docs.aws.amazon.com/kinesisanalytics/latest/sqlref/sqlrf-random-cut-forest.html. (Visited on: 2020-07-20).

RANDOM_CUT_FOREST_WITH_EXPLANATION (2020). https://docs.aws.amazon.com/kinesisanalytics/latest/sqlref/sqlrf-random-cut-forest-with-explanation.html. (Visited on: 2020-07-20).

Regions and Availability Zones (2020). https://aws.amazon.com/about-aws/global-infrastructure/regions_az/. (Visited on: 2020-05-25).

Replication (computing) (2020). https://en.wikipedia.org/wiki/Replication_(computing). (Visited on: 2020-07-09).

Rimal, B. P., Choi, E., and Lumb, I. (2009). "A Taxonomy and Survey of Cloud Computing Systems". In: *2009 Fifth International Joint Conference on INC, IMS and IDC*, pp. 44–51.

Simple Caching in AWS Lambda Functions (2019). https://rewind.io/blog/simple-caching-in-aws-lambda-functions/. (Visited on: 2020-08-16).

Streaming SQL Concepts (2020). https://docs.aws.amazon.com/kinesisanalytics/latest/dev/streaming-sql-concepts.html. (Visited on: 2020-06-12).

The Base16, Base32, and Base64 Data Encodings (2006). https://tools.ietf.org/html/rfc4648. (Visited on: 2020-06-12).

Tin Kam Ho (1995). "Random decision forests". In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. Vol. 1, 278–282 vol.1.

Using Redis as an LRU cache (2020). https://redis.io/topics/lru-cache. (Visited on: 2020-07-10).

UTF-8 (2020). https://en.wikipedia.org/wiki/UTF-8. (Visited on: 2020-06-10).

Vapnik, V. N. (1999). "An overview of statistical learning theory". In: *IEEE Transactions on Neural Networks* 10.5, pp. 988–999.

Vaquero, L. M., Rodero-Merino, L., Caceres, J., and Lindner, M. (Dec. 2009). "A Break in the Clouds: Towards a Cloud Definition". In: *SIGCOMM Comput. Commun. Rev.* 39.1, pp. 50–55. ISSN: 0146-4833. DOI: 10.1145/1496091.1496100.

Vogels, W. (Jan. 2009). "Eventually Consistent". In: *Commun. ACM* 52.1, pp. 40–44. ISSN: 0001-0782. DOI: 10.1145/1435417.1435432.

Webhook (2020). https://en.wikipedia.org/wiki/Webhook. (Visited on: 2020-07-06).

What is AWS (2020). https://aws.amazon.com/what-is-aws/. (Visited on: 2020-04-05).

What Is Load Balancing? (2020). https://www.nginx.com/resources/glossary/load-balancing/. (Visited on: 2020-06-30).

What is SaaS? (2020). https://azure.microsoft.com/en-us/overview/what-is-saas/. (Visited on: 2020-06-30).

What is streaming data? (2020). https://aws.amazon.com/streaming-data/. (Visited on: 2020-05-21).

Why Cloud Lock-In is a Myth: The Openness of AWS (2018). https://www.contino.io/insights/why-cloud-lock-in-is-a-myth-the-openness-of-aws. (Visited on: 2020-07-19).

Why Graph Databases? (2020). https://neo4j.com/why-graph-databases/. (Visited on: 2020-07-19).

Working with Items and Attributes (2020). https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html. (Visited on: 2020-06-30).

Working with Queries in DynamoDB (2020). https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.html. (Visited on: 2020-07-11).