Broad cloud

Final project report

Title:	Broad cloud
Author:	Marek Vargovčík, Michal Čič
Institution:	Professionshøjskolen UCN
Course:	Software Development
Class:	PSU0221
Semester:	3rd semester
Supervisor:	Simon Kongshøj
Company:	Broad I/S
Period:	21.4.2022 - 16.06.2022
Hand-in:	16.06.2022
Characters:	112 537

Abstract

This report summarizes the development of a hosting and management solution developed as a part of the Software development course at Professionshøjskolen UCN in collaboration with a digital agency called Broad I/S.

Preface

We would like to thank our supervisor Simon Kongshøj for excellent guidance throughout the development of this project and writing of this report. We would also like to thank Jason Güreş and Runi Johansen from Broad I/S for giving us the guidance and opportunity to develop a business-viable solution that will bring value to the company while providing a valuable learning experience.

Table of contents

1. Glossary

The glossary provides descriptions of abbreviations and specialized terms that are marked with a superscript number in parentheses: SSG [(1)]. To keep the level of visual clutter low, the glossary is referenced only the first time a term appears in this document.

(1) **SSG** – Static site generation is a website delivery mechanism that generates the HTML, CSS, and JavaScript of an entire website ahead of time, before being served to the user.

(2) **DNS** – Domain name system is a naming system that links IP addresses to human-readable domain names.

(3) **CDN** - Content delivery network is a group of distributed servers (usually globally) that delivers static assets with low response times by serving content to the user from a geographically close location.

(4) **NPM** – Node package manager is the default package manager for the JavaScript ecosystem.

(5) **CMS –** Content management system is software that enables users to add, modify and delete content without the need for technical knowledge of a programming language.

(6) **Google Lighthouse** – Google Lighthouse is a tool to measure performance of a website based on several weighted factors.

(7) **TLS –** Transport layer security defines a protocol to secure communication over HTTP.

(8) **API –** Application programming interface is a connection between computers or computer programs.

(9) **UI** – User interface is the layer of software that the user interacts with.

(10) **AWS** – Amazon Web Services is a subsidiary of Amazon focused on providing cloud services.

(11) **2FA** – Two-factor authentication is an extra security layer usually based on what the user possesses, an example is a phone with an authenticator application.

(12) **YAML –** YAML is a data-transfer format that aims to be easy for humans to read by using indentation levels like the python programming language.

(13) **JSON –** JavaScript Object Notation is a popular data-transfer format widely used for machine-to-machine communication.

(14) **REST** - Representational State Transfer is a standard that describes how computer systems should communicate.

(15) **GraphQL –** GraphQL is a querying language meant to be used with APIs.

(16) **OSS –** Open-source software makes its source code publicly available for distribution and modification.

(17) **AWS Availability zone –** An availability zone is an isolated location inside a region. A region is the physical location of AWS data centers.

(18) **CI/CD** - Continuous integration/Continuous delivery is a process that connects development and deployment of software.

(19) **Git –** Git is an open-source version control system.

2. Introduction

The internet is an inseparable part of our everyday lives in today's interconnected world. It is used for several reasons - to connect with other people, to access information or even shopping. It is an undeniable fact that the internet has changed the ways we, the consumers, expect businesses to operate. It is almost essential for a business to have an online presence, otherwise there is a risk of losing credibility, trustworthiness, and missing potential sales. People are often skeptical about businesses without digital information or reviews available and might instead choose a competitor that provides access to such information.

Business owners with requests to build a brand-new website or extend an existing one to provide better and/or new services to their customers are currently the main revenue stream in Broad I/S. Broad I/S is a startup digital agency offering tailor-made software solutions as well as providing continuous support throughout the life cycle of a product.

3. Problem area

Broad I/S offers, among other things, complete web application solutions and their management to clients. As a part of these solutions, it provides infrastructure for hosting such web applications via 3rd party services.

The main hosting service used by Broad I/S is Netlify. Netlify offers multiple services like Netlify Build to build, deploy, and manage web projects; a fast, resilient network for web apps named Netlify Edge; or serverless functions under the name Netlify Functions. [1] The mentioned services and the ability to register a domain under Netlify DNS keep the general management of a web application project 'under one roof' but cause vendor lock-in and promote inflexibility.

Most development projects at Broad I/S consist of a user-facing website built using a front-end framework that consumes data from a content management system. This architectural approach paired with SSG (1) is unofficially called Jamstack and Netlify builds its infrastructure around this concept. As Broad I/S expanded, developers in the company started facing larger scoped projects from bigger clients and Netlify's pricing model based around bandwidth usage and build time became too costly to justify.

Broad I/S wants to maximize profits by allocating resources and developer time to replace hosting services like Netlify with an in-house solution for web application deployment and management built on the AWS (10) platform. Building such a solution would enable Broad I/S to lower expenses for 3rd party services, offer their clients a more flexible and extensible solution and potentially extend its business into the hosting service territory.

4. Problem statement

The problem this project aims to solve is the lack of a sufficiently functional, extensible, and cost-effective CI/CD [(18)] pipeline that meets the business requirements of Broad I/S. It also aims to provide structured starting templates, integrated with the proposed pipeline, for developers to build on top of and deliver client websites in a fast, streamlined fashion. The proposed pipeline should automate the process of downloading source code from a specified repository, installing project dependencies, catching (build time) bugs to prevent the deployment of faulty software, running custom commands specified by the developer and finally deploying the built product onto hosting infrastructure. This project aims to answer the following questions:

- How can we lower the costs of hosting service infrastructure?
- How can we ensure deployment of websites and their associated resources without build-time errors and subsequent management of said websites and resources?
- How can we make the process of developing and deploying websites automated?
- How can we bundle related services into a single solution and lower the barrier of entry for developing and deploying websites?

5. Problem analysis

The following subsections will further analyze the problem briefly described in the Problem area (3.)

5.1. What is Netlify

Netlify, as explained in the Problem area (3.) is a platform as a service offering website hosting, a DNS (2) and a CDN (3), among other services.

5.1.1. Limitations of Netlify and its issues

During the time that Netlify is being used in Broad several issues were identified with either hosting, deployment process, or incompatibility with the design or technological choices that were made. At the time of writing, Netlify supports only static site hosting that is served to the public via a global content delivery network, ideal for client-side rendered JavaScript applications, that communicate with external APIs (8) to obtain data to display. Web applications, especially the ones built with technologies like React, Vue or Svelte, in most cases require a build step. Luckily, the Netlify process can build the project and automate the entire deployment process for supported build output types.

5.1.2. Low power provisioned instances

Netlify's business model offers a certain amount of build minutes in their pricing plans. This amount is often sufficient if the developed project is a simple website with only a handful of users that are making changes to it. It also depends on how complex the build process is, if and to what extent it can detect and cache output artifacts and many other factors that must be taken into consideration. At Broad the prevalent framework to build projects hosted on Netlify is Next.js, where the build process heavily depends on the number of pages that need to be pre-rendered and exported as static HTML files during SSG. The average build time increases linearly with the number of pages and heavily depends on the computing power of the system where the build process is running.

Unfortunately, Netlify provides weak compute instances with a maximum of 2 logical core CPUs. The average build time for medium sized projects at Broad is approximately 3 minutes (for around 80 pages to be rendered) excluding time it takes a builder instance to start. Therefore with 400 build minutes per month in a paid plan a website can only be deployed around 133 times which is a small number considering that deploy previews need to be built and deployed for editors to review changes before publishing them to a live audience, further

decreasing the number of times a website can be built. To continue operation and development of client websites, Broad is forced to buy more build minutes several times per month.

5.1.3. No support for file uploads in serverless functions

As previously mentioned, Netlify is a hosting service for static websites, and to fetch any dynamic data an API call must be made to query or mutate the resource in a way. Broad's customers often require a contact form to be embedded in their website with the possibility of uploading attachments, so their users have a way of reaching out. Netlify provides serverless functions that can be bundled in the same directory as a website. The integration is seamless, and the overall developer experience feels very polished due to Netlify CLI, a tool to simulate the Netlify stack locally. However, developers at Broad encountered issues where any file uploads made using Netlify functions ended up corrupted. Broad contacted Netlify's customer support, but they were unable to help, even after providing a repository with a function that reproduced the error Broad's developers were facing. As a result, Broad decided to switch to Google Cloud for any serverless computation.

5.1.4. Charging per team member

Some of the Netlify features, such as analytics and forms, are only accessible from within the Netlify administration panel. Similarly, it is necessary to have an account to be a team member when a person wants to be a bill payer or to manage the infrastructure (domain, build process, environment variables, etc.). Netlify charges 19$ for each team member for up to 6 users. After that, the plan must be upgraded to a business plan where the pricing goes from 19$ to 99$ per team member. This pricing model does not make sense in situations where even the bill payer or customer support personnel must have an account that is counted to the team member account limit. Sharing accounts with multiple people is not possible because of a need for different permissions and it also breaks Netlify's terms of use.

5.1.5. No insights into bandwidth usage

In a similar fashion of exceeding build minutes provided in the basic pricing plan, Broad also faced an issue with too much bandwidth being consumed and being charged for it. Netlify provides 400GB of bandwidth per month but the websites under Broad's management are assets heavy and quite popular, therefore consuming a lot of bandwidth in peak hours. Netlify charges 20$ for an additional 100GB of bandwidth and Broad's monthly bandwidth usage averages around 1TB per month.

5.1.6. Charging without approval

When any of the limits are exceeded Netlify automatically charges the bill payer via their default payment method. The fact that Netlify does not provide an option to verify and approve payments before charging hurts the provider-customer relationship.

5.2. What is NetlifyCMS

NetlifyCMS, an open–source product backed by Netlify, is a content management system [5] that uses Git [19] as the back-end – versioned persistent storage. It provides an effortless way for non-technical users to manage website content that can be consumed by any static site generator such as Next.js, Gatsby or any website template engine. NetlifyCMS provides developers with declarative syntax of defining schema of real-world entities as shown in figure 1 below. The CMS then generates a user interface (form with inputs) according to the schema as shown in figure 2.

```
{
  label: 'Title',
  name: 'title',
  widget: 'string',
},
{
  label: 'Subtitle',
  name: 'subtitle',
  widget: 'string',
},
{
  label: 'Description',
  name: 'description',
  widget: 'string',
},
{
  label: 'Excerpt',
  name: 'excerpt',
  widget: 'string',
},
{
  label: 'Thumbnail',
  name: 'thumbnail',
  widget: 'image',
},
{
  label: 'URL',
  name: 'url',
  widget: 'string',
},
```

Figure 1. NetlifyCMS data schema

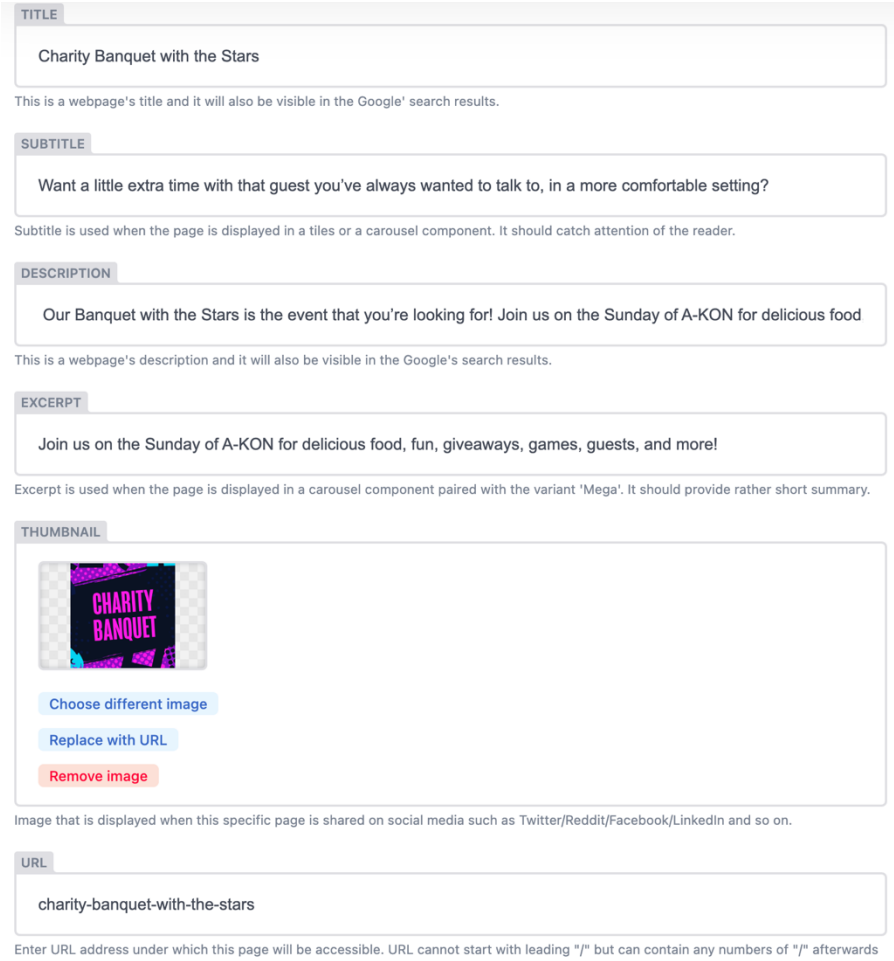

Figure 2. Auto generated administration interface from data schema

It is easy to get started and define various data types such as strings, arrays, sets, relationships between entities, files, and constraints such as minimum and maximum length, regular expressions, optionality, and more. Each data type has a corresponding UI [(9)] representation called a widget, that is displayed in the CMS administration panel. All widgets are open-sourced, therefore if there are unique requirements associated with displaying data, it is possible to extend or create a new data and widget type. Any change made in the NetlifyCMS administrator interface represents a Git commit and there are no application servers involved other than the one hosting Git. Git-based content management systems are very appealing as it is typically possible to host them for free, there is no server management involved, thus they are reliable and durable. However, developers at Broad ran into issues with certain types of workflows involving NetlifyCMS, and they will be discussed in the next section. Git-based CMSs are described in more detail in section 10.4.1 later in this report.

5.3. Limitations of NetlifyCMS and its issues

Most of the issues with NetlifyCMS are issues that are common with any Git-based content management system. But there are several other problems that are tied to NetlifyCMS as an open-source software.

5.3.1. Number of open issues and no maintainers

At the time of writing there are over 650 open issues and the last activity with a bugfix, or a feature release was published a month ago. There is also just a single maintainer. According to npm [(4)] statistics, the NetlifyCMS package is downloaded over 20 000 times per week, but the number of dependents (users of NetlifyCMS) might be higher due to Netlify providing a CDN download link which is a source to use within the HTML script tag instead of npm. However, there is no backing or sponsors other than Netlify themselves and most of the user base are individuals that do not contribute features upstream. This might be related to the fact that the NetlifyCMS system is written as a React application with class-based components that are now considered to be a bad practice, therefore, developers might feel hesitant to make any code changes. The other possible scenario is that for most individuals, NetlifyCMS works as is.

5.3.2. Vendor locked features

NetlifyCMS offers an authentication plugin that works out-of-the-box only when using the Netlify platform. It is possible to create a custom authentication back-end as a plugin but at the time of writing the documentation in this area is incomplete.

5.3.3. High number of features behind a beta flag

Many essential features are currently behind a beta flag, and it is unlikely to see any of them being stabilized and released as core features soon. However, without a local development workflow, open authoring, multi-language content (translations) or typed lists the CMS does not provide enough content management capabilities for medium and large-scale projects.

5.3.4. Lack of an application server for business logic

The appeal of Git-based content management systems is that they can be hosted without an application server. However, there are trade-offs because then the CMS lacks features like co-authoring, real-time collaboration, live content previews, dynamic data-fetching, publish scheduler and many more capabilities that are taken for granted in systems like WordPress or other API-based content management systems. The difference between API-based and Git-based content management systems is described in more detail later in this document in section 8.4.

6. Requirements

The requirements for this project are derived from the problem statement and extended problem formulation above. Stakeholders and user stories (Appendix 1) were also considered when gathering requirements for this project.

6.1. Stakeholders

The following section lists and describes the stakeholders of this project.

6.1.1. Development department at Broad

Developers at Broad are going to be interacting with the system throughout its whole lifecycle, from the first deployment, through development based on client needs, to maintenance, fixes as well as adding features once the product has been launched, published, or otherwise identified as done. Their main expectation of this product is good developer experience.

6.1.2. Directors at Broad

Jason and Runi - the directors at Broad - have high level decision power and business influence over the product. They are interested in utilizing the product to satisfy higher client demand faster and more efficiently. Their main expectation of this product is business growth and monetary gains.

6.1.3. General public

Public users are a group of people interacting with the presentation layer of this product. They are interested in the functional part of the presentation layer, together with appearance and speed.

6.1.4. Editors

Editors create, update and audit content through the content management system. Their main interests are ease of use and intuitiveness of the CMS. As seen from Broad's past projects, an editor might be the client themselves or an employee of the client.

6.1.5. Administrators

An administrator fulfils the role of managing editor accounts and their permissions in connection with the content management system. Administrators are editors with elevated permissions. As seen from Broad's past projects, both technical and nontechnical personnel should be able to make administrative changes in the content management system in case of

unavailability. The main interests of administrators are having granular control and pleasant user experience in the CMS.

6.2. FURPS+

Proposed by Robert Grady and Hewlett Packard, FURPS+ is a tool to define functional and non-functional requirements of a software project. [2] Each category of requirements will be described separately in the following subsections, with security requirements described under the "+" subsection.

6.2.1. Functionality

6.2.1.1. Front-end template kits

Template kits for the front-end will serve as a project entry point for developers at Broad. They will set a baseline for project structure; a common code standard and they will introduce best practices when creating applications or parts of applications facing the end user. Template kits will therefore provide a fast and straightforward way to scaffold a project in its initial stage.

6.2.1.2. Content management system

A CMS will provide data and assets to the front-end. It will serve as a publishing platform for editors. CMS administrators will have elevated permissions to manage editor permissions and accounts. The CMS will be modular - developers will be able to extend and adjust functionality of the CMS based on specific client needs. A content management system will be an option for clients that do not want to depend on developers being available and update the contents of a website or for other project stakeholders that do not want to or do not know how to interact with the codebase.

6.2.1.3. Custom API endpoints

Custom API endpoints will provide a platform to interact with external services that are not related to the CMS like a newsletter subscription function or a contact form submission function. Custom API endpoints should contain an encapsulated unit of business logic and can be scaled separately from the CMS, removing a resource overhead.

6.2.1.4. Custom CMS endpoints

Custom CMS endpoints will provide extended control over the CMS system such as a custom scheduler implementation. Their main purpose is to access the internals of the content management system without extensive configuration, compared to the generic API endpoints.

6.2.1.5. Persistent asset storage

Persistent asset storage will provide a place to store files like images, videos, or compressed asset folders to be used in the front-end and content management system. It is necessary to provision persistent asset storage because of data loss protection, granular access control and storage cost optimization.

6.2.1.6. Custom domains and subdomains

It is necessary to provide configuration options for creating new domains and subdomains for websites and their underlying content management systems that currently are not associated with a domain. The product will also provide a way to connect existing domain records to the deployed resources for clients that have already established userbases and want a smooth transition onto a new system.

6.2.1.7. Continuous delivery system

The continuous delivery system will enable developers to change existing functionality or implement new functionality and introduce it to production, development, testing, staging or other custom environments. This system will ensure that whenever changes are made to the system and are ready to be published, the deployment process will automatically be activated and will deploy a system that passes defined preconditions like the build process and tests passing successfully.

6.2.1.8. Background jobs

The product will include the option to define background jobs for tasks that take more than 20 seconds to execute. Background jobs can be utilized for heavy workloads that require a long time to execute, workloads where there is a need for continuous and periodical execution like resource monitoring, or even use cases where there is a need to wait for user input, like interaction with a chat bot.

6.2.1.9. Event-based triggers

The product will include an option to define triggers that are listening for events on various resources. This feature is helpful in situations where developers want to gather information in an encapsulated way, without depending on the initiating events. An example is error logs for web-service requests to provide instant notifications and feedback when an error occurs.

6.2.2. Usability

6.2.2.1. Documentation

Because software solutions are rarely stale and not evolving it is a necessity to educate new as well as seasoned users and potential contributors who can bring value back to the product and

make it better. Documentation fulfills the role of explaining the internal workings of the system as well as introducing basics to get started with using it. The documentation about the basics of the system will cover the following topics:

- How does the deployment system work?
- How to connect a remote Git repository?
- What resources can be used and what is their purpose?
- How to get started?
- Frequently asked questions

6.2.2.2. CLI toolkit

Initializing a new project is oftentimes a time-consuming process, therefore an easy-to-use tool that performs repetitive tasks when setting up a new project will be a part of the product. It will have a command line interface where the developer can choose which characteristics the system is supposed to have and can input data points that vary depending on the project like the name of the application. It will have the following options:

- Specifying the name of the application
- Specifying the region where the application is to be deployed
- Option to set up a custom domain
- Option to include a CMS
- Selection of a front-end template kit - a 'none' option included for a custom front-end implementation

6.2.3. Reliability

6.2.3.1. Monitoring of costs and utilization of resources

Implementing monitoring is necessary to limit the risk of overloading computing resources and therefore to limit budget overspending. Overloading of resources can be introduced by developer error or a rise in the popularity of the product. With monitoring in place, developers can prevent overloading and overspending by taking corrective actions.

6.2.3.2. Alarms and actions

Alarms and actions are usually connected to monitoring as described in the previous point. An example of utilizing alarms and actions is the CPU utilization of a computing resource. When

the CPU load reaches a set threshold, an action is fired to scale the resource up to accommodate higher load.

6.2.3.3. Aggregated logs and dashboards

Gathering information connected to reliability is necessary to ensure correct functioning of a system. This product will provide aggregated logs and dashboards to serve as a place to store information about monitoring as described previously in this section as well as a place to store logs of individual resources.

6.2.4. Performance

6.2.4.1. Moving content and functionality closer to the user

Modern web pages are expected to be loaded fast. One of the metrics Google Lighthouse [(6)] uses to measure website performance is server response time or time to first byte (TTFB). "Users dislike when pages take a long time to load. Slow server response times are one cause for long page loads." [3] One of the measures to reduce server response times is moving the content closer to the user. A content delivery network or CDN caches static assets across multiple servers in different physical locations called edge servers and reduces the geographical distance between the requestor and responder. The smaller the distance is between the two communicating entities, the lower the response time. Content delivery networks recently evolved from static asset caching only to functions caching as well, figure 3 below shows release years and names of this functionality by various providers. This gives developers the option to treat business logic like static assets as they are described earlier in this paragraph and utilize a CDN to move functions closer to the users.

Provider	**Service name**	**Release year**
Cloudflare	Cloudflare Workers	2018 [4]
Google	Cloud CDN	2016 [5]
Amazon	Lambda@Edge	2017 [6]

Figure 3. CDN hosted serverless services releases

6.2.4.2. Auto-scaling of performance critical services

Accommodating higher loads when demand rises is vital for performance. The product will include a configuration option to configure resource scaling under heavy load.

6.2.5. Supportability

6.2.5.1. Infrastructure testing

The solution will provide a suite of infrastructure tests that ensure the correct deployment and configuration of resources. These tests catch regressions or help with developer confidence when refactoring code because they keep the developer informed about the state of the build or deployment.

6.2.6. +

6.2.6.1. Security

Security is an integral part of any software application and needs to be addressed together with the main functionality of a product. The following subsections explain security requirements for this product.

6.2.6.1.a HTTPS (Hypertext Transfer Protocol Secure)

HTTPS communication will provide an encrypted and therefore secure channel between the CMS and front-end and front-end and the client device by issuing TLS [7] certificates.

6.2.6.1.b Least required privileges for deployment entities

Keeping the privilege level of deployment entities will ensure that these entities cannot make destructive or otherwise unwanted changes to the deployment system. This requirement prevents changes that can either be non-intentional (made by verified developer entities) or malicious (made by attackers that gained access to the system).

6.2.6.1.c Two-factor authentication for AWS accounts

Enforcing two-factor authentication or 2FA [11] for AWS users will decrease the possibility of an attacker gaining access to AWS resources and prevent malicious intent.

6.2.6.1.d Non-default ports on provisioned resources

Configuring ports for provisioned resources to use non-default values will decrease the chance of automated systems or human attackers gaining potentially exploitable information.

6.2.6.1.e Public/private subnets for provisioned resources

Systems handling sensitive information or functionality will be placed in private subnets to limit the chance of exploitation from the public side of the internet.

6.2.6.1.f Securely stored sensitive values

Sensitive values used for access to private resources will be placed in a centralized secrets store managed by a third party. This store will be easily accessible by the deployment pipeline.

6.2.6.1.g Protection against client-side attacks in front-end template kits

Protection against client-side attacks will be facilitated by setting up an appropriate code standard for the front-end to catch common vulnerabilities in the development stage.

6.2.6.1.h Secure CMS access

Access protection for CMS users will be implemented by setting appropriate authentication policies and adhering to security standards regarding authentication. The CMS needs protection to mitigate the risk of leaking potentially sensitive information.

6.2.6.1.i Rotating secrets

Secret values will be periodically changed to limit the amount of time an attacker has to exploit the product if they gain access to the system.

6.2.6.1.j DDoS protection

Protection against distributed denial of service attacks will be implemented by using a third-party service for botnet detection.

6.2.6.1.k Data backups

A periodical data backup system will be set up to prevent data loss in case of errors or malicious activity.

7. Projects stages

This project is divided into 3 stages to set clear milestones for progress and to reduce complexity in the development process. The following subsections provide a high-level overview of what is implemented in each stage. The defined project stages are developed chronologically, and the highest priority is placed on stages 1 and 2.

7.1. Stage 1

The first stage of the project focuses on recreating functionality like the Netlify platform. The main purpose of this stage is developing a platform to deploy statically generated websites by using an automatic build and deployment system. Due to the static nature of the system, this stage can accommodate Git-based content management systems, most popular between developers being NetlifyCMS - based on the metric of GitHub stars. [7] This stage also includes serverless functions to decouple business logic from the client-side application and provide a server-side environment without managing a server. This stage also features the option to register a new custom domain for the website with an attached TLS certificate for HTTPS support. Stage 1 can be deployed standalone with the CLI tool developed in stage 3 when configuring a new project as it is functional for clients that do not expect much change in their website contents. Stage 1 will also include DDoS protection for CDN connected resources, least required privileges for deployment entities and enforced 2FA for AWS users.

7.2. Stage 2

Stage 2 expands on the foundations laid by stage 1 by replacing the Git-based CMS with a customizable API-based CMS connected to a database and blob asset storage. Using an API-based CMS comes with its pros as well as cons. The extensibility and customizability of the system is one of its positives, because it can handle complex relations and business logic suited for clients with higher demands. On the other hand, it requires an actual server environment and therefore concerns like load balancing must be considered. Stage 2 also extends the build and deployment system with database migration support. Additional security requirements implemented in this stage are securely stored sensitive values and secure CMS access.

7.3. Stage 3

The last stage is an extension of stage 1 and 2. It adds additional measures for usability, advanced security, and scalability described in the subsections below. Stage 3 also includes the

implementation of a testing suite for deployment resources. Background jobs, event-based triggers and aggregated logs with dashboards are all implemented in stage 3.

7.3.1. Usability

Broad's aim is to streamline and automate development of new websites, the usability or in this case developer experience of project initialization and continued development have been taken into consideration to maximize efficiency.

7.3.1.1. CLI Tool

Initializing a new project is oftentimes a time-consuming process, therefore an easy-to-use tool that performs repetitive tasks when setting up a new project was developed. Introducing a project set-up process bundled into a tool will provide basic configuration options for a project, such as choosing a starting front-end template, what functionality should the website have (include CMS, include domain) while also setting a common code standard and enforcing best practices by configuring linting and formatting tools.

7.3.1.2. Starter kits

Starter kits are ready made, minimal website front-ends with pre-configured web frameworks or libraries. A starter kit can be chosen during project initialization based on client needs or developer preference.

7.3.1.3. Documentation

Providing documentation on how to extend and customize the product beyond the initialization phase is accomplished by clear and extensive descriptions of resources and processes used by and in the product. The documentation also includes a guide on how to get started with the product.

7.3.2. Advanced Security

Security basics have been covered in stage 1 and 2. The third stage introduces additional security layers such as placing resources in public or private VPC subnets depending on the resource, enabling data backups, setting up rotating secrets as well as configuring non-default ports on provisioned resources. These measures help further restrict access to sensitive resources for potential attackers.

7.3.3. Alarms and monitoring

Stage 3 also contains measures to prevent overspending of resources. Alarms and monitoring ensure that relevant stakeholders are notified when infrastructure costs are approaching a set maximum value.

7.3.4. Load balancing configuration

Stage 3 covers further load balancing configuration of the CMS in case of a high-demand traffic surge. Such configuration will create or destroy instances in case demand falls or rises, respectively.

8. Design

The following section will describe design considerations and will provide conclusions to decisions that were faced during the design process.

8.1. Cloud providers

The business requirement of Broad is to use the Amazon Web Services ecosystem as hosting infrastructure. While AWS is the current requirement, it might change in the future, and because the major cloud providers offer similar solutions, a few select ones are going to be compared in the following subsections.

Leading vendors in the cloud infrastructure industry are Amazon, Microsoft, IBM, Oracle, and Google. All the vendors mentioned have solutions for the problem the development team is trying to solve. Amazon, Microsoft, and Google with their respective services shared around 64% of the global cloud computing market share as of Q4 2021. [8] Because of their prevalence, these three providers are going to be compared in the following subsections based on which services they provide.

The requirements of this project define an abstract, high-level overview of the functionality that is to be achieved and can be dissected into the following types of services: static website hosting, continuous delivery pipeline, serverless functions, domain management, identity and access management, database hosting, persistent asset storage, web-service hosting, and content delivery network. An overview of comparable services can be found in figure 4 at the end of this section.

8.1.1. Amazon

Amazon Web Services or AWS has the highest global cloud infrastructure services market share as of Q4 2021 at 33%. [8] Many of the biggest companies on Earth are using AWS to fulfill their computing needs, such as McDonalds, The Volkswagen Group or Netflix. [9] [10].

8.1.2. Microsoft

Microsoft Azure is the second largest vendor as of Q4 2021 by cloud infrastructure services market share with 21% globally. [8] Azure services are used by multinational organizations as well as governments around the globe, some of the biggest are Bosch, HP, or Goodyear. [11]

8.1.3. Google

The last cloud infrastructure provider in this comparison is Google with its Google Cloud Platform. As of Q4 2021 it was the third biggest vendor among cloud infrastructure service providers with 10% market share. [8] Popular organizations are associated with Google Cloud, some of them are PayPal, Twitter, or Airbus. [12]

Service	Microsoft	Amazon	Google
Static website hosting	Static Web Apps	Amazon Simple Storage Service (S3) AWS Amplify	Firebase Hosting
Continuous delivery pipeline	Azure Pipelines	AWS CodeBuild AWS CodeDeploy AWS CodePipeline	Cloud Build
Serverless functions for custom API endpoints	Azure Functions	AWS Lambda	Cloud Functions
Domain management	Azure DNS	Amazon Route 53	Cloud DNS
Identity and access management	Microsoft Azure Active Directory	AWS Identity and Access Management	Identity and Access Management
Database	Azure SQL Database Azure Cosmos DB	Amazon Aurora Amazon RDS Amazon DynamoDB Amazon DocumentDB	Cloud SQL Firestore
Persistent asset storage	Azure Blob Storage	Amazon Simple Storage Service (S3)	Cloud Storage
Web-service hosting	App Service Virtual Machines	Amazon EC2 AWS Elastic Beanstalk	Compute Engine Cloud Run
Content delivery network	Azure Content Deliver Network	Amazon CloudFront	Cloud CDN

Figure 4. Cloud providers services comparison

8.2. Serverless computing

This project aims to be an upgrade to Broad's tooling by utilizing the ecosystem and solutions offered by AWS. Now, Broad manages several projects with servers being manually provisioned, maintained and scaled vertically by adding more hardware resources. Deployments to these servers are also manual, resulting in downtime and the likelihood of human error. Based on the described experience it was decided that to replace manual configuration, deployment, and maintenance with streamlined processes, most of the next generation projects at Broad are to be deployed to the cloud in a serverless manner. When serverless (IaaS) is not an option, then fully managed environments (PaaS) with all underlying infrastructure maintenance being taken care of are to be deployed, allowing end-users (Broad

in this case) to only focus on custom business logic of the product. This decision was made primarily to eliminate the need for human interactions during the process of deployment and delegate that to a computer.

Serverless computing, is a term that was defined by AWS (in the cloud context) during an AWS re:Invent event in 2014. The research publication "The rise of serverless computing" [13] defines the term as follows: "Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running". This type of computing is seeing an increased adoption year by year as it features a billing model of being charged only for resources, often a time-based calculation, that were used by the software. This is interesting for businesses because of the study "Usage Patterns and the Economics of the Public Cloud" [14] concluded, there is a large gap between allocated resources in the datacenters and resources that are being utilized. This often comes from a fear of under provisioning and experiencing unavailability and downtimes due to sudden bursts and spikes in usage. Businesses combat this problem by doing the opposite - overprovisioning, as downtimes could potentially hurt the business - customer relationship and trustworthiness of the product. By not using serverless or PaaS, the costs of human labor and expertise in server computing also must be factored in.

For this reason, it can safely be said that serverless computing is on the rise and cloud providers like Amazon, IBM, Microsoft, or Google now provide a wide range of serverless computing capabilities. The serverless paradigm takes all or at least most of the infrastructure concerns away from the developers. The key features of serverless computing are cost and elasticity [13]. The pricing model is subject to the policies of the cloud provider but commonly it is a model where only time spent running the code is billed. The second key feature - elasticity, is about the scalability of the executable business logic from zero to very large-scale applications, assuming there are no other bottlenecks like the budget or the throughput of other accompanying services. Serverless computation units are, in most common scenarios, small and isolated functions representing a business use case that can be invoked as a reaction to an event [13]. A serverless function is uploaded to the platform where it is fully handled by the provider. This opens many opportunities for the provider to do further optimization to reduce the cost and increase the profit margin [15]. Uploading code to the cloud without knowing where it is and what context does it share can be a trust issue and a potential security problem [15]. Serverless platforms do not persist state, therefore it is up to the code itself to make sure that its state (if needed) is persisted by an external resource such as a database [13].

8.2.1. Limitations

Serverless computation also has its downsides and limitations, especially for well-established businesses with existing products. Full adoption and conversion of an existing product to its serverless variation introduces many challenges and changes to software design. Highly coupled code must be split into isolated pieces of logic and then composed together [13]. As there is no automatic process to decompose a project into isolated functions, this represents a monumental task that is labor intensive and often cannot be justified as a business investment. Instead, slowly adapting new features as serverless units is preferred.

Additionally, serverless environments are currently not standardized and vary between cloud providers [13]. There often is not an option of swapping from one provider to a different one. This introduces a concern about vendor lock-in. A serverless solution might be utilizing a feature or concept that is not supported by different vendors, making the switch harder or impossible.

Serverless functions can also introduce higher response latency as it takes time to fully-spin up a new serverless unit due to the elasticity and scaling to zero. Cold start times of functions range from half a second up to multiple seconds in added response time. Cloud providers combat this by providing a certain number of minutes to keep the unit “warm” before the serverless container expires and shuts down [15]. This might pose a deal breaker for response time sensitive applications.

Another concern that comes with elasticity is monitoring. Since it is easy to scale, the budget now becomes a concern. A specific example comes from the company Milkie Way that launched their product while fully relying on Google Cloud compute engine. The shipped software introduced an infinite loop causing invocations of a function to be executed over 116 billion times [16]. The company received a bill for 72 thousand US dollars. If it were not for the elasticity of the serverless computation the server would have exhausted its memory and became unavailable [16].

8.2.2. Evaluation

Based on the information provided from "Usage Patterns and the Economics of the Public Cloud" [14] Broad favors the adoption of serverless mostly because of the fear of under or over provisioning of cloud resources. The solution to these two concerns is correct configuration of monitoring services like CloudWatch to prevent over provisioning and paying more than is necessary or under provisioning and sacrificing performance. The fact that there are no legacy

projects that would have to be adopted to this new paradigm and the benefit of not having to hire a full-time DevOps engineer are far outweighing the limitations and downsides of serverless.

Additionally, cold start times are currently not a concern as our projects do not fall into the category of applications which would require or depend on fast responses from an API. Initially this project should provide functionality of contact forms and signing up for a newsletter implemented in serverless fashion.

8.3. Infrastructure as code

"Infrastructure as Code (IaC) is the maintaining and provisioning of infrastructure through code instead of through manual processes." [17] The mindset in Broad is to automate anything that can be automated, and since this product requires multiple resources to be provisioned and configured at the same time, it was decided that the deployment processes of this product will be using infrastructure as code. IaC can be written in an imperative or declarative way; the following subsections will compare both approaches of deploying and configuring AWS resources.

8.3.1. AWS CloudFormation

AWS CloudFormation is a service that enables developers to compose and deploy resources in a declarative approach by treating the infrastructure as code. The format CloudFormation works with are files in YAML (12) or JSON (13), these files act as a blueprint for resources to be deployed to one or multiple CloudFormation stacks. Stacks are units of related resources that are deployed together. An example CloudFormation configuration written in JSON that deploys a virtual machine can be seen in figure 5 below. CloudFormation is a candidate for this project because it enables easy deployment, updates, rollback, and destruction of groups of resources in a single action.

```
"Ec2Instance" : {
  "Type" : "AWS::EC2::Instance",
  "Properties" : {
    "ImageId" : {
      "Fn::FindInMap" : [
        "AWSRegionArch2AMI",
        { "Ref" : "AWS::Region" },
        { "Fn::FindInMap" : [ "AWSInstanceType2Arch", { "Ref" : "InstanceType" }, "Arch" ] }
      ]
    },
    "KeyName" : { "Ref" : "KeyName" },
    "InstanceType" : { "Ref" : "InstanceType" },
    "SecurityGroups" : [{ "Ref" : "Ec2SecurityGroup" }],
    "BlockDeviceMappings" : [
      {
        "DeviceName" : "/dev/sda1",
        "Ebs" : { "VolumeSize" : "50" }
      },
      {
        "DeviceName" : "/dev/sdm",
        "Ebs" : { "VolumeSize" : "100" }
      }
    ]
  }
}
```

Figure 5. CloudFormation example

CloudFormation, compared to the Cloud Development Kit described in a subsection below, lacks a straightforward way to chain resource deployment where individual components depend on information available only after deployment like unique resource identifiers generated during resource creation. Another downside of CloudFormation compared to the Cloud Development Kit is its verboseness. JSON and YAML are formats intended for data transfer and communication between computers and can get quite complex for human readers to understand.

8.3.2. AWS Cloud Development Kit

AWS Cloud Development Kit (CDK) is an open-source project that introduces a programming language connection with CloudFormation. It abstracts resources into object-oriented paradigms like classes and objects and presents an imperative approach to infrastructure as code. Because the CDK builds on top of CloudFormation, it inherits the concept of stacks as described in the previous subsection, it also introduces more concepts, the most important being constructs and apps. A construct is a basic building block of AWS CDK that represents a "cloud component" and encapsulates everything AWS CloudFormation needs to create the component. [18] "A construct can represent a single AWS resource, like an Amazon Simple Storage Service (Amazon S3) bucket or it can be a higher-level abstraction consisting of multiple AWS related resources". [18] An app is a top-level construct that acts as a scope delimiter for stacks. An overview of the CDK architecture is shown in figure 6 below.

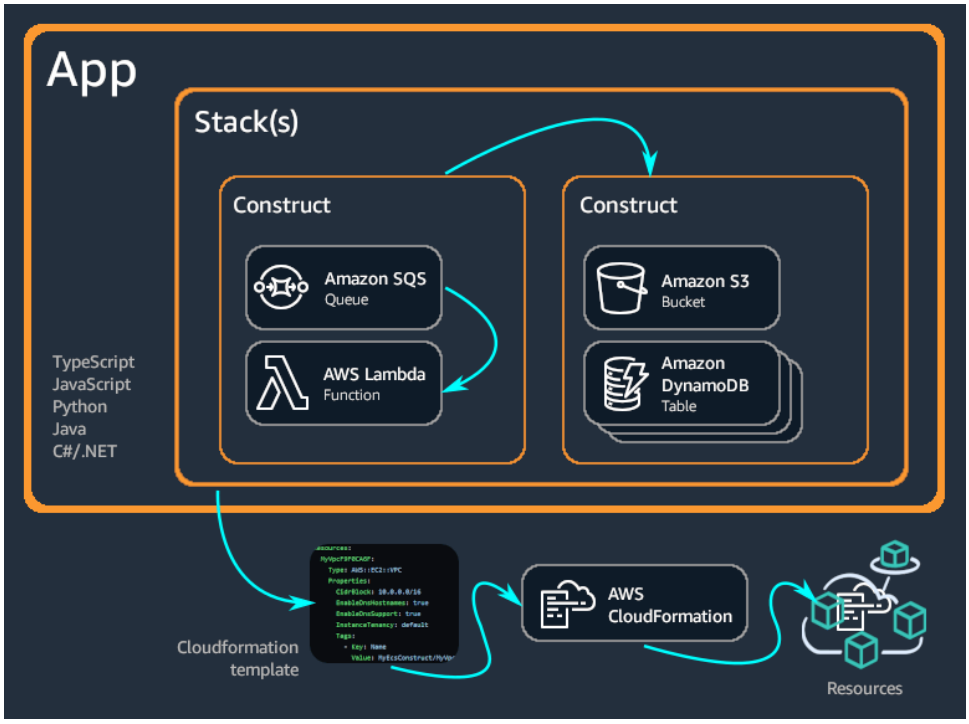

Figure 6. CDK architecture [19]

The Cloud Development Kit solves the infrastructure as code dilemma in an imperative way and it supports multiple languages - listed on the left side of figure 6 above. The core CDK source code is written in TypeScript and compiled into modules in other programming languages using jsii. [20] The advantage of the CDK over declarative configuration templates like in CloudFormation is the ability to add dependencies between resources, run testing suites, or use programming idioms like loops, conditionals, composition, or inheritance. Figure 7 shows a simplified example for deploying a virtual machine written in TypeScript.

```
const ec2Instance = new ec2.Instance(this, 'Instance', {
  vpc,
  instanceType: ec2.InstanceType.of(
    ec2.InstanceClass.T4G,
    ec2.InstanceSize.MICRO
  ),
  machineImage: ami,
  securityGroup: securityGroup,
  role: role,
})
```

Figure 7. AWS CDK example

8.4. Content management system

The most common use case for a content management system in projects that have been developed by Broad so far is adding, modifying and deleting visual content of a website. The main decision behind implementing a content management system into this product is to allow non-developer users to alter the contents of a website which has a side benefit of offloading work from developers to users. Having users alter the contents of a website also gives them more creative freedom as they are not dependent on a developer to implement changes, but they themselves can be the implementers and can iterate through ideas faster.

It is necessary to mention that this section and the whole project is focused on headless content management systems. A headless CMS serves only as a store of data instead of a more traditional approach of CMS implementation which contains both data and the presentation layer – an example of a traditional CMS is WordPress. Headless CMSs give the developers, as well as the designers, more freedom in developing a custom presentation layer, which many of Broad's clients require.

There are two main types of headless content management systems that are going to be compared in the following subsections – Git-based and API-based. These two types differ in ways of consuming and storing content.

8.4.1. Git-based content management systems

Git-based content management systems are built around the Git version control system and store content in files committed to a repository. A Git-based CMS' source code and content are a part of the repository together with the presentation layer source code. This type of structure has the benefit of direct access to content in a filesystem-like approach, but at the same time, it adds overall weight to the project – especially when assets like images or videos are also stored in the same repository. Utilizing Git as a backbone of a CMS has the benefit of easy history and rollback management or easy branching and therefore environment management - because every change made in the CMS is a new Git commit.

A characteristic of Git-based CMSs is that they are usually (but not exclusively) paired with a website that is exported as static HTML, CSS, and JavaScript. The main reason for this approach of website delivery is speed and durability. The approach of exporting a website every time a change is made becomes a negative once a project grows into having a larger number of pages (~80 as discussed in section 5.1.2) because of the increased duration of build times.

Storing content in a Git repository also hinders the possibility of reusing it for multiple platforms like mobile or desktop when the primary application is web-based because of the same limited data-querying options of remote Git repository providers.

The cost of Git-based CMSs is one of their positives, because hosting a remote private or public Git repository with GitHub is free and other major providers like GitLab and Bitbucket also offer free tiers in their plans, it should be noted, however, that all the providers introduce different limits in their plans. [21] [22] [23] One of the limits that the developers must be especially aware of is the repository size limit that can be reached by storing large assets like high-quality images or video directly in the repository. A problem like this can be overcome by introducing special services for remote asset storage and management like Cloudinary or Uploadcare.

8.4.2. API-based content management systems

API-based CMSs communicate with the presentation layer via an Application Programming Interface and are built as a backend web-service with a server environment connected to a database. Common API implementations include technologies like GraphQL (15) or standards like REST (14), with the communication happening over HTTP(S).

The cost for self-hosting an API-based CMS is composed of resources the system operates on. The price is dependent on factors like server usage, storage usage and database usage. The mentioned factors are individually broken down in Appendix 2.

Reading content from an API-based CMS can be done from multiple platforms, frameworks, and programming languages, because communication over the internet is supported in most if not all languages and frameworks. This makes API-based CMSs flexible if requirements for the presentation layer, programming language, framework or platform are diverse or are subject to change. Managing content that is universally accessible enables developers to choose the presentation layer technologies independently of the CMS, and so connecting to the CMS from a statically generated website during build time, from a dynamic, client-side rendered website, or a mobile application is an option – among others.

8.4.3. Candidates

This subsection will compare different API-based content management systems. The selection of CMSs is limited to API-based systems, because the current set-up with Git-based NetlifyCMS is a reasonable solution for small clients, but not for mid to large-size clients.

Another threshold in candidate selection is that the system can be self-hosted based on the requirements of extensibility, scalability, adjustability as well as overall fine-grained control over the system.

8.4.3.1. Strapi

Strapi is an open-source headless CMS written in JavaScript and running in Node.js. Based on GitHub stars (~45 000) it is the most popular system in this comparison. [24] Strapi fulfills the base requirements of being extensible, self-hostable and features customizable roles and permissions for editors and administrators. Data can be consumed via extendable REST or GraphQL APIs. [24] Strapi supports persisting data in the following databases: SQLite, MongoDB, MySQL, and PostgreSQL. [25] The documentation of the project also includes a detailed guide to deployment to AWS and other providers.

8.4.3.2. Directus

Directus is an open-source headless CMS written in JavaScript and running in Node.js. It provides an extensible platform for editors, developers, and administrators. It works with multiple relational databases: PostgreSQL, MySQL, Oracle Database, MSSQL, SQLite and CockroachDB. It also supports customizable roles and permissions for users, editors, and administrators. Querying data can be done via extendable GraphQL or REST APIs. [26] Directus also provides a guide on how to deploy the CMS to AWS as well as other hosting providers.

8.4.3.3. KeystoneJS

Keystone is an open-source headless CMS written in TypeScript running in Node.js. It provides an extendable GraphQL API to query data. Keystone also features custom roles and access control. [27] Keystone documentation provides guides on how to deploy the system to Heroku, Microsoft Azure or Railway, but does not provide one for AWS. Supported databases for Keystone are PostgreSQL and SQLite.

8.4.4. Conclusion

As implied in the subsections above, Strapi and Directus are similar in what they accomplish. It was decided to go with Directus because of its ability to wrap any supported relational database without needing migrations as well as not needing to introduce any proprietary structure into an already established database. This quality of Directus makes the product more flexible. Directus also provides better typings for TypeScript (autocomplete) compared to

Strapi. Keystone was not chosen because it lacks some functionality that both Directus and Strapi do provide.

8.5. AWS ecosystem

This project should utilize and benefit from AWS services that are by design interconnected as an ecosystem and complement each other by various degree of functionality while paying only for provisioned and used cloud resources - enabling faster time to the market and not sacrificing application performance or user experience [28]. AWS provides several ways how to design and utilize its ecosystem of services such as a combination of cloud resources and existing on-premises infrastructure in a network, how to store and distribute files and assets, how to utilize virtual private servers and their placement in a scalability group. The ecosystem is fast-growing, and new additions are announced annually at the AWS re:Invent conference [28]. In the following subsections, AWS services core to the use case of this project will be explained more in-depth to provide more context to the reader for an easier understanding of details discussed in the implementation section.

8.5.1. VPC (Virtual Private Cloud)

Any resource created in AWS is placed into a virtual private cloud (VPC). VPC is a virtual network that, to a high extent, works just as a standard local network with a certain network prefix – an aggregation of IP addresses. VPCs are logically isolated from each other by default [29]. Connecting multiple VPC together is possible through other AWS products and constructs like peering, transit gateway or private link [30]. Subnets are a core concept to networking and can be described and considered as a network within a network [29]. In other words, it is a logical subdivision of the bigger network to create multiple smaller networks. To create a VPC network one must be familiar with the basics of Classless Inter-Domain Routing (CIDR), network classes and subnetting masks. The default VPC provides 172.31.0.0 network with a prefix of 16 [30]. This allocates a total of 65536 available addresses. There are three default subnets that are defined with a network mask of 20 – each allocating 4096 IP addresses. This leaves enough room to expand the existing subnets or create new ones if it is deemed necessary as there are more than 50 thousand unallocated IP addresses available in the VPC. When designing and creating a VPC a means of communication between the resources and security must be kept in mind. Routing tables are an essential construct that enables fine-tuning of communication boundaries. To configure a routing table, it is necessary to add a new routing record with information about destination and target. Destinations and targets are defined in

the official AWS documentation as follows: “Destination is range of IP addresses where you want traffic to go (destination CIDR). For example, an external corporate network with the CIDR 172.16.0.0/12. Target is a gateway, network interface, or connection through which to send the destination traffic; for example, an internet gateway”. [29].

To avoid exposing private resources (virtual private servers, databases, etc.) to the internet, it is possible to hide them in a specific type of subnet – private subnet. To make a subnet private the routing table cannot allow communication of the allocated IP addresses to the internet gateway [29]. Such routing table can be defined as a default and all subnets within the VPC will be implicitly associated with it. To make a subnet public, an additional non-default route table with a rule to route traffic to the internet gateway can be used to explicitly associate subnets that are meant to be publicly available to the internet.

Broad cares about security. The intention is to place any sensitive resources (cache, databases, virtual private servers) in a private subnet and use load balancers to route traffic to them. This pattern enables fine-grained control over the incoming traffic and makes changes to the traffic rules easy.

8.5.2. Persistent asset storage

Amazon Web Services offers multiple solutions that provide persistent asset storage. The offered products range from filesystem-like to general-purpose blob storage. Persistent asset storage in this product is to be used by the content management system and should be as universal as possible while also being cost effective.

Amazon Simple Storage Service (Amazon S3) is a service that is universal and can be accessed across platforms through standards like HTTPS. This makes it a promising candidate for this product. Other storage services offered by Amazon like Amazon Elastic Block Store, Amazon Elastic File System or Amazon FSx are all tied to a specific resource like a server instance or an operating system which decreases their universality.

8.5.3. Serverless functions for custom API endpoints

Simple use cases like newsletter subscriptions or contact form submissions often require only a single unit of business logic to accomplish their goal. AWS offers a compute service called AWS Lambda that lets developers run code without provisioning or managing servers. Lambda automatically scales to meet demand and is therefore a solution fit for both small and large clients and their needs.

8.5.4. Databases

The following subsections will describe different types of databases used in the product.

8.5.4.1. Relational

A relational database is a type of database that stores and provides access to data points that are related to one another. [31] The need for a relational database comes from the content management system - Directus, that persists data about its internal state as well as content for the presentation layer in a SQL database. AWS offers two relational database services: Amazon Aurora and Amazon RDS. The main comparison factors for these two services are scalability and performance.

8.5.4.2. Amazon RDS

Amazon RDS is a managed relational database service that supports database engines shown in figure 8 below. RDS is running on virtual servers provisioned in the cloud with configuration options including replication, backup, and update management.

Amazon RDS variations
MySQL
PostgreSQL
MariaDB
Oracle
MSSQL
Custom

Figure 8. Amazon RDS engines

8.5.4.3. Amazon Aurora

Amazon Aurora is a relational database with MySQL and PostgreSQL compatibility. Instead of running only on virtual servers, Aurora is split into compute and storage layers to take advantage of the AWS cloud ecosystem. Aurora uses a distributed and shared storage architecture that is a key factor in performance, scalability, and reliability for Aurora clusters. [32]

8.5.4.4. Conclusion

The main reason for picking Amazon Aurora over RDS is scalability and performance that comes from optimizations done by the AWS team.

8.5.4.5. Key-value

A key-value database is a non-relational type of database storing values that are connected to unique string keys. In the case of this product, it is meant for storing data from interaction with custom API endpoints like the newsletter subscription endpoint. AWS offers only one key-value service – DynamoDB – that is used in this product.

8.5.5. Elastic Beanstalk

Before diving into Elastic Beanstalk, it is necessary to introduce Elastic Compute Cloud (EC2) as a low-level service and one of the oldest services offered by AWS. EC2 was released as a service to the public as an initial release back in 2006 [28] and enables provisioning of virtual private servers on-demand with a range of hardware configuration variations running on dedicated hardware or in virtual environments. Adding more instances if a task requires increased computation power and terminating them when demand is low can be done in a predictable manner, hence the name elastic compute.

Elastic Beanstalk is a product as a service (PaaS), an abstraction service over EC2 that adds orchestration and scalability features out of the box for more seamless and automated experience [33]. Additionally, Elastic Beanstalk integrates application health monitoring and logging [34]. Combination of the these out-of-the-box features makes it possible for end-users to focus on the critical business logic instead of manual cloud resource management. It is a fully managed service - meaning that the used environment is automatically being updated with the latest security and patch updates [34].

At the time of writing, there are 9 platform runtimes provided (listed in figure 9), the most popular being Node.js, Java, PHP, or Python, that could be selected during the getting started process [34]. 10th provided platform runtime is a capability of using Docker, an OS-level virtualization technology enabling software to be distributed in containers [34]. For the developers this means that if the required platform runtime is not amongst the currently supported ones, it is possible to provide a custom platform runtime in Docker.

Platform runtimes
.NET Core on Linux
.NET on Windows Server
Java SE
Node.js
PHP

Python
Ruby
Go
Tomcat

Figure 9. Currently supported Elastic Beanstalk platform runtimes

Elastic Beanstalk introduces the concepts of application and environment [34]. The intended approach is to have a single application with multiple environments that are fully isolated from each other meaning hardware resources of one environment are not dependent on each other in any way. Applications are associated with the platform runtime and environments are constructs associated with the actual source code [34]. Source code is supplied to the environment via different means by leveraging the ecosystem, for example using AWS build service CodeBuild or distributed as a ZIP file coming from a S3 bucket [33]. Each new ZIP file represents a new environment version. Application can switch between versions as deemed necessary [34]. This makes rolling back from a failed update easy.

An environment can also be extended by platform hooks or extension files to further configure or expand upon the AWS managed resource [34]. This is often used as a post-deployment step and can serve as a place to run other business tasks such as executing database migrations. Such an example is captured in figure 10. below.

```
container_commands:
  collectstatic:
    command: "django-admin.py collectstatic --noinput"
  01syncdb:
    command: "django-admin.py syncdb --noinput"
    leader_only: true
  02migrate:
    command: "django-admin.py migrate"
    leader_only: true
  99customize:
    command: "scripts/customize.sh"
```

Figure 10. Example of Elastic Beanstalk extension running commands for Django application

As explained in 8.2.2, Broad is confident about using serverless computing as a development and architecture paradigm for most of its business requirements. However, when a task is not possible to be hosted as a serverless computation unit (a function or multiple) then fully managed services are preferred as they take maintenance of cloud resources out of the equation. Such a use case exists in this project, the API-based content management system, which will

be fully explained in the following section dedicated to content management systems, requirements and candidates that fit the specified requirements. There are some already known parameters and features of the content management system (explained more in-depth in section 8.4) and figure 11 below shows how Elastic Beanstalk can be used to deploy such technological stack.:

Platform runtime	Node.js
Database	Relational (external service accessible over network)
Assets storage	S3 bucket
Hooks and extensions	Used to install dependencies and execute database migrations
Load balanced	Running in an autoscaling group

Figure 11. Known parameters and features of a CMS

8.5.6. AWS CodePipeline

It is in Broad's best interest, considering its small team size, to have a fully automated deployment process, so even in the absence of a development team responsible for the project, other developers can issue a fix with just a knowledge of the code, not the entirety of the CI/CD setup.

AWS CodePipeline is a continuous delivery service that can be used to model, visualize, and automate the steps required to automate continuous software releases. Various stages of a software release process can be modeled and configured using CodePipeline. [35]

To accommodate a wide range of use cases that may come as a requirement of a business decision, CodePipeline provides configuration options for distinct parts of the build and deployment process [36]. Understanding how CodePipeline works precedes the understanding of concepts that are present and used in AWS CodePipeline [36]. Further subsections contain concepts of CodePipeline together with their definitions that are quoted from the official documentation. These concepts and terms are introduced as they are essential for understanding the implementation described in later in this document (9.7.)

8.5.6.1. Pipeline

"A pipeline is a workflow construct that describes how software changes go through a release process. Each pipeline is made up of a series of stages." [36]

8.5.6.2. Stage

"A stage is a logical unit you can use to isolate an environment and to limit the number of concurrent changes in that environment. Each stage contains actions that are performed on the

application artifacts. Your source code is an example of an artifact. A stage might be a build stage, where the source code is built, and tests are run. It can also be a deployment stage, where code is deployed to runtime environments. Each stage is made up of a series of serial or parallel actions." [36]

8.5.6.3. Action

"An action is a set of operations performed on application code and configured so that the actions run in the pipeline at a specified point." [36]

8.5.6.4. Artifact

"Artifacts refers to the collection of data, such as application source code, built applications, dependencies, definitions files, templates, and so on, that is worked on by pipeline actions. Artifacts are produced by some actions and consumed by others. In a pipeline, artifacts can be the set of files worked on by an action (input artifacts) or the updated output of a completed action (output artifacts)." [36]

This project plans to utilize CodePipeline for defining the process of automating a trigger – to start execution of the pipeline when certain condition happens (push to a specified branch of a repository), build and test steps – to verify that software is functional and there are no regressions, and deployment – to deploy an updated version (or versions) of software to a specific environment.

8.6. Front-ends

Broad customers are commonly from business areas of gaming, esports, cryptocurrencies, and public events. Customers typically demand smooth user experience, rich interactivity, a modern look, snappy performance, and a feeling of immersion. Projects are typically not data heavy. Based on the requirements the most complexity lies in the front-end. Over time, Broad developers established a foundation of tools – a common starting ground – for all projects that fall into that complexity category. This toolset features a selection of in-house supported front-end frameworks, building and bundling processes, styling libraries, architectural patterns, configuration of static analysis tools (ESLint, Prettier) and other essential parts of front-end development. It was decided that front-end starter kits featuring different front-end frameworks are going to be a part of the initial release of this product, to firmly define the tools that Broad's developers use and define a baseline code standard.

8.6.1. Client-side vs server-side rendering

This subsection explains concepts of client-side and server-side rendering and its advantages, and disadvantages. It is important to know the differences to better understand Broad's decision-making process of which technology to pick for a client.

8.6.1.1. Server-side rendering

Server-side rendered websites are the traditional way of delivering a response to the visitor of a website by returning some data (in formats of XML, JSON, HTML, or plain text) back to its browser [37]. Functionality, like pulling data from an external data source, is done on side of the web server and the client that has requested the page has no knowledge of such processes taking place [37]. By navigating to the next page, visitors' browser requests another page and web server repeats the general process and delivers the response.

8.6.1.2. Client-side rendering

Client-side rendered websites are a recent addition to a developer's toolkit. It is a design approach of structuring the website or application where the presentation layer is delivered to the client up-front as JavaScript files or a bundle [37]. JavaScript code is then parsed by the browser. Client-side rendered websites are often called single page applications because page transitions do not cause the browser to reload [37]. Transition to other pages is more seamless. With this approach web servers are usually only concerned about responding to users' requests with data in formats like XML or JSON and do not include any opinion on how to present it.

The biggest benefit of client-side rendered websites is the seamless browsing experience. However, because the page transitions are now handled by JavaScript code, the first load of the website might be slower as it needs to download JavaScript code, parse, and execute it. SEO (search engine optimization) also takes a hit because the application, on the initial load, does not have all the content available for web-crawlers [37]. Usually, it displays just an empty shell hidden behind a loading indicator. The page will eventually be populated once the web server sends a response. SEO bots – crawlers – do not wait too long before moving on to the next page so the page might be incorrectly crawled and not appear in the search result. Therefore, client-side rendering is a better fit for web-based applications that operate behind authentication such as dashboards or management panels.

8.6.1.3. Conclusion

Nowadays, front-end frameworks tend to combine both approaches – a hybrid of server-side and client-side rendering, where the first initial load is served as a HTML response so content can be properly crawled, and the first paint of content is fast. At this step, the application is

visible to the user but is not interactive – all interaction is handled by JavaScript that has not loaded yet. JavaScript code is then downloaded, and a process called hydration takes place and starts connecting (applying) JavaScript to the visible HTML markup. After that, any page transitions are handled by the browser and JavaScript.

8.6.2. Starter kits

Broad currently supports and allows its developers to develop products in three front-end frameworks. All supported frameworks are built on top of React, utilizing it as a foundation and providing common features such as file-based routing, data-loading capabilities, server-based runtimes, API functions, and others. There is also an ongoing initiative to support a very recent framework: Remix.js. This framework advertises itself as a web framework rather than a front-end framework [38]. It still uses React for its template declaration but tries to fully transform experience of writing React code into more server/client model [38]. In the following subsections the mentioned frameworks will be explained more in-depth and some of the concepts as it is important for the implementation and its integration to this product. These are shown in figure 12 below.

Framework	**Supported version**	**Status**	**Server-side rendering**
Vite (Vanilla React)	v2 (latest)	Stable	Experimental
Next.js	v11 and v12	Stable	Yes
Remix.js	v1 (latest)	Experimental	Yes

Figure 12. Framework supported features

8.6.3. Vite (Vanilla React)

Vite is OSS [(16)] and does not fall entirely into the category of frameworks nor libraries. It is best described as a tooling software that significantly improves developer experience when writing React code. It is developed by the creator of the Vue front-end framework, Evan You. At the time of writing, it is a package with just a shy of a million weekly downloads. The motivation for development of yet another JavaScript build tool is that previous generation of tools of the same purpose are becoming slower as applications are getting bigger due to amount of JavaScript, like external dependencies that the application relies on, increasing exponentially [39]. Previous generation tools, like Webpack, Rollup or Parcel, were written entirely in JavaScript and could not leverage the latest additions to the browser - availability of ES modules, because the standards did not exist at the time or were not implemented [39].

Additionally, they were often slow and inefficient with hot module replacement where the detection of changes was both slow and lacking [39]. Vite integrates an efficient bundling tool, esbuild, that claims it manages to be a hundred times faster at bundling JavaScript than competing tools Webpack, Rollup or Parcel [40]. Figure 13. below presents a benchmark of bundling three.js library with various tools. Esbuild comes as a clean winner in speed.

Bundling tool	**Time (s)**
esbuild	0.33s
parcel 2	32.48s
rollup + terser	34.95s
webpack 5	41.53s

Figure 13. The time to do a production bundle of 10 copies of the three.js library from scratch using default settings, including minification and source maps [https://esbuild.github.io/]

Vite's strategy to provide fast development experience when working with React is that it differentiates from dependencies and actual source code. Dependencies are unlikely to change and can be pre-bundled once and cached. Source code is then served to the browser as individual ES modules and changes to a single module can be detected, therefore only this specific module is to be replaced, thus achieving a fast and efficient hot module replacement [40]. Below is a figure (14) of out-of-the-box supported Vite features.

Supported features
Hot module replacement for faster development
Plugins and extension to customize the build process
TypeScript support
Server-side rendering (experimental)

Figure 14. Vite (Vanilla React) - supported features

Broad uses Vite for projects that do not require any prerendering and/or SEO capabilities, usually internal tools, dashboards, management software behind authentication. In other words, projects which do not need server-side rendering. Deployment to AWS should be fully automated and an updated version should be pushed to a CloudFront distribution from an S3 storage bucket and served from a global CDN network. Therefore, each release should create a cache invalidation.

8.6.4. Next

Next is another OSS project that was founded by Guillermo Rauch and had its initial release, version 1.0, in 2016 [41]. Since then, Next.js underwent many changes and shaped in

accordance with the needs of the developers. The latest stable version, at the time of writing, is version 12 [42]. According to npm trends, where the Next package has over 2.7 million weekly downloads, Next is the most popular React framework amongst JavaScript developers [42].

Next's main objective and purpose is to provide an all-in-one tool experience for developers to accommodate different (pre)rendering strategies that applications might need. There are 3 different approaches available now – static generation where HTML is generated during build step, server-side rendering where HTML is generated on each request and a third one being a combination of both static generation and server-side rendered pages [43]. Additionally, it is possible to use static generation to generate an HTML template, for example in a loading state, and load dynamic data on client-side instead [43].

To create a new page associated with a URL it is only necessary to create a file with React code in the pages directory (unless configured differently) - this is possible because of file-based routing [43].

There are many other features that are favored by developers such as built-in CSS support (CSS or SCSS files, CSS modules, CSS-in-JS solutions), layouts, built-in image optimization strategy (transforming images on build time or on-fly to reduce size), API functions and more. See figure 15 below where more built-in features of Next are listed.

Supported features
Hot module replacement for faster development
Plugins and extension to customize the build process
TypeScript support
Client-side and server-side rendering
File-based routing
Data loaders
API functions
Page layouts
ESLint integration

Figure 15. Next.js - supported features

Next is being used for companies like Netflix, GitHub, HBO, Nintendo, or Hulu [44]. This increases Broad's trust in choosing Next as a foundation framework for its client's projects. For all projects so far, Broad has only chosen an all-in static generation approach of generating

HTML files on build time. However, the new cloud tooling should accommodate all three approaches of how to render a Next application. For a static generation of HTML files, these files should be distributed via CloudFront together with other static assets (JavaScript files, images, fonts). For a server-side rendering and hybrid approach the MVP should run Next in an Elastic Beanstalk environment.

8.6.5. Remix

Remix was already briefly introduced in 8.6.2 as a web framework, an innovative approach for developing web-based applications with React as its foundation [38]. Remix does not have multiple rendering approaches. [38] It only supports server-side rendering where HTML is generated for each request. Remix is not a server per se but can be described as a handler that is given to an actual JavaScript runtime [45]. This abstraction allows Remix to be deployed to any provider supporting JavaScript serverless functions [38].

Similarly to Next, a framework that was described in the previous subsection, Remix also features file-based routing [46]. However, since Remix is a web framework rather than only a front-end framework it mixes concepts of a view and controller in each route (file) [38]. Remix automatically checks for presence of an exported function called loader and action. Loader, as the name suggests, is responsible for loading the initial data and action is handling dispatched actions by the view.

Remix is leveraging a client/server model and tries to minimize the complexity of maintaining JavaScript state in single page applications [45]. These tend to suffer from out-of-sync data issues and state management in general becomes increasingly hard to manage. Instead, Remix builds abstraction over fetch request API (built-in browser standard) and standard HTML forms and handles invalidations and re-fetching of data automatically [45]. An action can be invoked simply just by submitting the form. It is never exposed to the front-end, therefore mutations such as database connections can be established without leaking secrets such as database credentials or any sensitive information to the end user.

Because of embracing and reliance on web standards in Remix, developers can benefit from a side-effect of their projects working without using any JavaScript [45]. This is possible because Remix, with JavaScript disabled, will handle any data manipulation and fetching just as traditional server-side application would work – by using HTML form actions, submitting the data, and refreshing the page to re-fetch the new data.

Remix has many other features, and the most significant ones are listed in figure 16 below.

Supported features
Hot module replacement for faster development
TypeScript support
Only server-side rendering
File-based routing
Data loaders
Page layouts
Works with no JavaScript enabled
Error handling (one way to handle both front-end and back-end)
Data writing with HTML forms

Figure 16. Remix.js - supported features

Broad is excited about the potential of Remix, its unique approach to building web-based applications and variety of deployment targets. It is planned to be incorporated into the initial release of MVP by deploying the Remix handler as an AWS Lambda function. Static assets are to be served from an S3 bucket. However, it must be monitored closely as it is currently unknown what the monthly cost of running this service could be.

8.7. Means of distribution

This software is not aimed to be directly used by end users, but it is built to accommodate business needs of Broad or other developers outside of Broad to make setting up infrastructure a straightforward process to launch any web-based project if it utilizes the same or similar choice of technologies. The development team at Broad is full of people who are enthusiastic towards open-source software and try to contribute back into OSS in multiple ways, like upstreaming features that were tailor built for its customers, reporting discovered bugs, suggest features or simply joining the community to discuss and help other developers. This infrastructure project has no proprietary secrets or know-hows, and the project consists of tying multiple cloud services together into a resilient infrastructure. Therefore, at this moment, Broad is in favor of open sourcing the project as reusable AWS constructs at later stages of the development process.

8.7.1. Codebase

Based on the requirements, the project should eventually be made available as a public repository for anyone to reuse, fork, change and redistribute. Broad already hosts all its projects on GitHub under its own organization. There are many competing Git-based code hosting

online services like GitHub, GitLab or Bitbucket. Broad started its existence after Github was acquired by Microsoft in the beginnings of 2019 and shortly after the acquisition Github extended its free tier to offer unlimited private repositories [47]. Out of the other mentioned Git platforms, GitLab and Bitbucket, at that time GitHub was already hosting code for incredibly significant projects in the open-source sphere like React and React Native, Visual Studio Code, Tensorflow or Kubernetes. Based on the developer survey conducted by Jailton Coelho, Marco Tulio Valente in their academic publication "Why Modern Open-Source Projects Fail" [48] it can be said that GitHub, with its 73 million users and 200 million repositories, is the most favored Git-based hosting service for both open-source and private product development. One of the drawbacks of GitHub is that it is introducing a vendor lock as its code is not open-sourced and cannot be self-hosted. GitLab offers the option to fully self-host the GitLab environment (Git repository server and the management UI) on-premises or in a private cloud [49]. However, due to the nature of projects at Broad where security and total independence is not a priority, it is much more important for the business to offload any non-core development tasks (in this case code hosting) to a third-party provider so Broad can focus on the business critical, revenue generating development tasks instead of overcoming repository hosting obstacles. GitHub provides a strong set of social features that make development easier such as integration of concepts like issues, pull requests, code reviews, code review assignments, project boards, announced releases, notifications and more [50]. These social features, without a doubt, are the main reasons why it is possible to see rapid development of open-source products that accept outside contributions [51]. Broad, with its plans to release this product as open source, does not have any expectations to see many contributors but sees it as an opportunity for other developers to upstream their additions and changes and create meaningful discussions. For these reasons, there are currently no plans or initiatives to move to other it-based code hosting services.

8.7.2. NPM package and CLI tool

As an addition to offering a publicly accessible repository with an open-source license this product will include a way to scaffold projects quickly and include prompts to enter dynamic data such as the name of the project, resources to include, and other variables. Many open-source projects offer a tool that automatically sets up a minimal working example that comes with a directory structure and necessary dependencies to quickly get started. The following figure (17) shows a handful of projects that provide such tools and commands to start the process.

Project	Bootstrapping command
React	npm create react-app
Next	npm create next-app
Remix	npm create remix
Vite	npm create vite

Figure 17. OSS projects and their CLI bootstrapping commands

All the OSS projects and their bootstrapping CLI tools listed in figure 17 above are available to download and install from the Node package manager. Prior to release and widespread adoption of Node.js, a runtime that enabled developers to run JavaScript outside of browsers, browser packages were distributed simply by downloading and placing them next to other static assets or later by using Bower, a package manager for the web. Nowadays, npm hosts more than 1.3 million packages [52] and provides both a public and a private scoped registry. The JavaScript ecosystem consists mostly of small independent libraries that provide a limited amount of functionality - so-called micro packages. Packages can depend and be dependent on other packages forming an ecosystem. A study "Empirical Study of the npm JavaScript Ecosystem" warns about fragility of such ecosystems where the removal of a package can cause other dependent packages to be broken, causing a domino effect [53]. There are also security concerns, but these will be discussed in its own section related to security instead (10.8). Given the circumstances, Broad is convinced that to receive any open-source contributions and to promote its open-source software it must be distributed over npm. npm, luckily, provides necessary primitives built into its distributable. Broad will utilize both npm package publishing mechanisms built into their own CLI tool (npm publish) [54] and its initialization primitive (npm init) to scaffold a package [55]. Both will be discussed more in depth in a section dedicated to implementation.

8.8. Architecture overview

The diagram shown in figure 18 below describes the application in its deployed state. The presentation layer frameworks displayed in red are optional and only one is going to be deployed at a time.

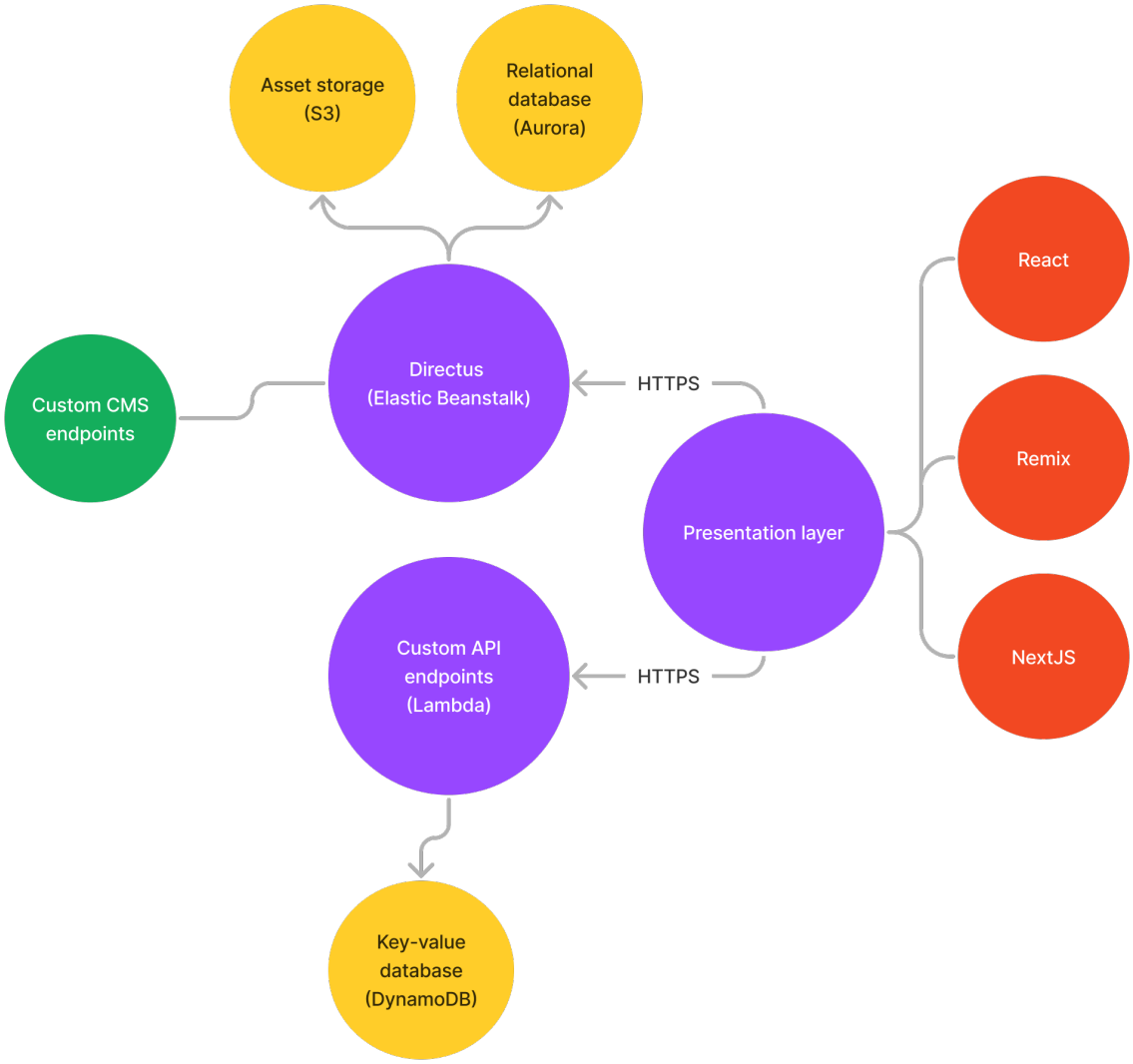

Figure 18. Architecture overview diagram

9. Implementation

The following subsections will go into detail about distinct parts of the implementation process with specific examples and parts of the codebase.

9.1. Project initialization

As mentioned in the design section 8.7.1 about the codebase, versioning of the project is managed by Git and the repository is hosted on GitHub.

9.1.1. Monorepo

Monorepos or monolithic repositories are single repositories that contain multiple projects, related or unrelated, sharing the same dependencies. [56] In the case of this product, the repository contains multiple folders and configuration files as shown in figure 19 below. This project uses Turborepo because of the setup simplicity and the features it provides, but there are multiple other tools with very similar feature sets that can be used to manage a monorepo.

Figure 19. Project folder and file structure overview

9.1.1.1. Apps

The apps folder, shown in figure 20 below, contains the core logic for the CDK, CMS, custom API endpoints and different frontend starter templates. Each of the folders is a separate project that can be run with commands defined in the top-level package.json file. An example of a group of scripts used to manage the custom API endpoints is shown in figure 21.

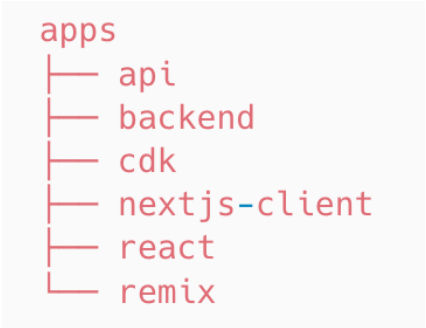

Figure 20. Apps overview

```
"api:build": "turbo run build --scope=\"@broadlify/api\"",
"api:dev": "turbo run dev --scope=\"@broadlify/api\"",
"api:lint": "turbo run lint --scope=\"@broadlify/api\"",
"api:lint:fix": "turbo run lint:fix --scope=\"@broadlify/api\""
```

Figure 21. Top-level package.json scripts example

9.1.1.2. Node modules

The folder with node modules contains shared dependencies downloaded and installed from npm for apps in the monorepo. Non-shared dependencies are in the respective apps folders. This structure is in place so that the overall monorepo size is not bloated due to dependency duplication and is managed by Turborepo.

9.1.1.3. Packages

Files and folders to be shared between projects are defined in the packages folder. The initial content is the config folder shown in figure 22 that contains the base configuration for static analysis that is described in more detail in section 11.1.2 about static analysis tools below.

```
packages
└── config
    ├── eslintrc.base.json
    ├── README.md
    ├── package.json
    └── tsconfig.base.json
```

Figure 22. Packages overview

9.1.2. Static analysis tools

Static analysis tools define code standards and perform type checking across individual applications in this product. The tool that performs static type checking is TypeScript and the tool for static analysis is ESLint because of their popularity and documentation.

9.1.2.1. TypeScript

TypeScript is a strongly typed programming language that builds on JavaScript. [56] As a superset of JavaScript, it provides static type checking for compile-time errors and helps Broad ship more crash-proof applications. The base configuration of TypeScript is done in the “tsconfig.base.json” file that serves as a starting point for configuration of other TypeScript apps in the repository. This configuration can be extended based on the project or development team, or it can be disable if the app is not written in TypeScript. The full content of the base TypeScript configuration file can be found in appendix 3.

9.1.2.2. ESLint

ESLint is a tool for identifying and reporting on patterns found in ECMAScript/JavaScript code, with the goal of making code more consistent and avoiding bugs. [57] It works by defining rules that lead developers to use proven practices. ESLint is a tool used extensively in Broad to set a comprehensive code standard. It is extensible and offers a plethora of plugins built by the core team as well as the community.

The base configuration package used in this project is a modified version of eslint-config-canonical and is chosen based on previous experience and usage in several Broad projects, it has over 1000 rules that target the JavaScript, TypeScript and Node.js environments. The modifications of the core package are done through a configuration file - "eslint.base.json" - and consist mostly of disabled rules, the full content of the file can be found in appendix 3. The configuration of each project can be extended based on the project requirements and needs of the development team.

9.2. CDK

The Cloud Development Kit implementation in this project is written in TypeScript. The entry point to the project is the "infrastructure.ts" file in the bin folder where the main infrastructure stack is defined and is, together with the CDK app (as described in section 8.3.2), instantiated. Code for all individual resources is placed in the infrastructure.ts file in the lib folder. The entirety of the CDK folder is shown in figure 23 below.

```
cdk
├── README.md
├── bin
│   └── infrastructure.ts
├── cdk.context.json
├── cdk.json
├── lib
│   ├── buildspecs.ts
│   └── infrastructure.ts
├── package.json
└── tsconfig.json
```

Figure 23. CDK folder structure

9.3. VPC

CIDR block described in the configuration as 10.0.0.0/16 means that the VPC can hold 2^{16} IPv4 addresses from 10.0.0.0 to 10.0.255.255. The VPC is also configured with 2 private and

2 public subnets. Private subnets enable communication between resources inside the VPC and public subnets (paired with AWS gateways) enable communication to and from the internet and the VPC. The maxAzs configuration parameter specifies at most how many AWS availability zones [17] the VPC can span. Simplified configuration of the VPC is shown in figure 24 below.

```
const vpc = new ec2.Vpc(this, 'vpc', {
  cidr: '10.0.0.0/16',
  maxAzs: 2,
  subnetConfiguration: [...],
  vpcName: 'broadlify-vpc',
})
```

Figure 24. VPC instantiation

9.4. Issuing validated certificates to domains and subdomains

TLS certificates enable secure HTTP communication. Issuing certificates is done using AWS Certificate Manager. The certificates are attached to domains and subdomains during resource deployment in the CDK. Certificates are attached to the CloudFront CDN distribution of the presentation layer or the CMS load balancer. An example of certificate attachment is shown in figure 25 below, this example is specific to the React front-end starter kit.

```
const reactCertificate = new acm.DnsValidatedCertificate(
  this,
  'reactCertificate',
  {
    domainName: REACT_DOMAIN_NAME,
    hostedZone: zone,
    region: 'us-east-1',
    subjectAlternativeNames: [addWWWPrefix(REACT_DOMAIN_NAME)],
  }
)

const reactDistribution = new cloudfront.Distribution(
  this,
  'reactDistribution',
  {
    certificate: reactCertificate,
    defaultBehavior: {
      allowedMethods: cloudfront.AllowedMethods.ALLOW_GET_HEAD_OPTIONS,
      origin: new cloudfrontOrigins.S3Origin(reactBucket, {
        originAccessIdentity: cloudfrontOAI,
      }),
      viewerProtocolPolicy: cloudfront.ViewerProtocolPolicy.REDIRECT_TO_HTTPS,
    },
    defaultRootObject: 'index.html',
    domainNames: [REACT_DOMAIN_NAME, addWWWPrefix(REACT_DOMAIN_NAME)],
    enableLogging: true,
  }
)
```

Figure 25. React TLS certificate attachment

9.5. Lambda functions

Code for AWS Lambda functions is organized as depicted in figure 26 below.

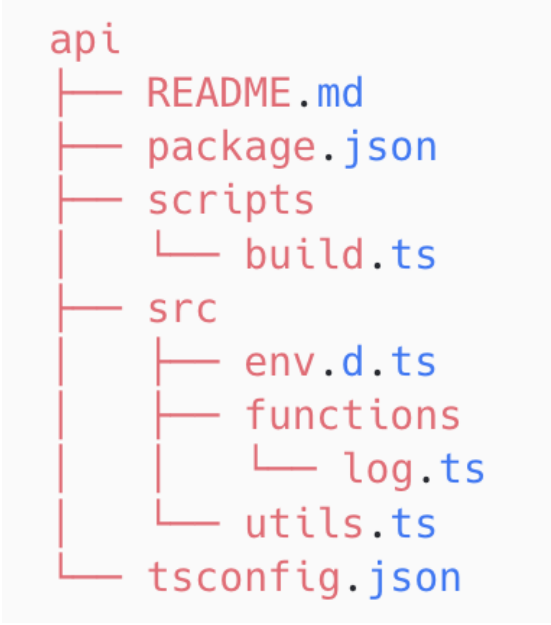

Figure 26. Serverless API folder structure

9.5.1. Use cases

Common use cases defined in the requirements include a newsletter subscription function, which inserts a record with an email address into a provisioned DynamoDB table and a function that sends an email from a contact form.

9.5.2. Bundling

The function code is written in TypeScript and to be used as an AWS Lambda handler, it must first be compiled to JavaScript. The npm command for building functions is shown in figure 27 below.

```
"build": "tsc && ts-node scripts/build.ts"
```

Figure 27. API functions build command

The build command is comprised of two steps. The first step is TypeScript compilation with the "tsc" command that type-checks files based on configuration in "tsconfig.json". The second step is running a build script with "ts-node scripts/build.ts," this command runs code defined in the "build.ts" file. The main part of "build.ts" is calling the esbuild.build function that will take TypeScript files as "entryPoints," transform them into the format specified in figure 28 and output them into the specified output directory. The "entryPoints" parameter receives an array of paths pointing to function TypeScript files.

```
await esbuild.build({
  allowOverwrite: true,
  bundle: true,
  entryPoints: [ENTRY_FILE],
  outdir: OUTPUT_DIRECTORY,
  platform: 'node',
  sourcemap: 'inline',
  target: 'node14',
})
```

Figure 28. ESBuild function call

Another important part is zipping files from the "OUTPUT_DIRECTORY" into a "source" zip file. The commands used to accomplish this are shown in figure 29 below.

```
const commands = [
  `cd ${OUTPUT_DIRECTORY}`,
  'mkdir source',
  'cd source',
  'zip -Ar source ../*.js',
]
```

Figure 29. Command for zipping the source code of serverless functions

9.5.3. Uploading

The function registration is a two-step process defined in the CDK code. The first step is uploading the bundled function source code into a S3 bucket provisioned beforehand. The second step is registering the Lambda function as shown in figure 30 below.

```
new lambda.Function(this, 'Lambda', {
  code: lambda.Code.fromBucket(apiBucket, 'source'),
  environment: lambdaEnvironment,
  handler: 'log.handler',
  runtime: lambda.Runtime.NODEJS_14_X,
})
```

Figure 30. Lambda function registration in the CDK

The CDK code also handles registering the build process used for updating the function once it has been initially deployed. The build process is fired every time a commit is pushed to the repository.

9.6. Elastic Beanstalk

Deploying Directus to AWS Elastic Beanstalk is done through the CDK and it has two stages that are explained in more detail in their respective subsections. The first stage is initial deployment, and the second stage is CI/CD pipeline integration for further development.

9.6.1. Initial deployment

The initial deployment of Directus is done by bundling the source code into a ZIP archive, uploading the archive into a S3 bucket, and then pointing the Elastic Beanstalk environment source to the archive in the bucket. Configuration of the Elastic Beanstalk environment includes the choice of EC2 instance types, autoscaling configuration in case of demand rises and drops. Further configuration options are the VPC ID for the environment to be included in a VPC, and connection configuration for persistent asset storage and relational database. The Elastic Beanstalk environment depends on the database and persistent asset storage S3 bucket

to finish provisioning before starting the deployment of the environment itself, it is explicitly stated in code as shown in figure 31 below.

```
env.node.addDependency(database)
```

Figure 31. Elastic Beanstalk environment dependency on database provisioning

9.6.2. CI/CD Pipeline integration

The CI/CD integration enables continuous development and is vital to keep Directus extensible and adjustable. The build process is triggered by a commit being pushed to the remote GitHub repository and repeats the steps of bundling Directus source code, deploying it to an S3 bucket and refreshing the Elastic Beanstalk environment to use the newest build.

9.6.3. CMS customization

It is a functional requirement for the CMS to be customizable, an example being custom CMS endpoints. The customization for Directus is done by developing respective extensions in the "src/extensions" folder as it is shown in figure 32 below.

```
src
└── extensions
    ├── displays
    ├── endpoints
    ├── hooks
    ├── interfaces
    ├── layouts
    ├── modules
    └── panels
```

Figure 32. Directus extensions folder

A custom Directus endpoint for newsletter subscriptions has been developed in TypeScript and a simplified version of it is shown in figure 33 below. TypeScript code must be compiled into JavaScript and placed in the correct folder (extensions/endpoints in this case) to be recognized by Directus. The process of compiling is the same as for custom API endpoints in section 11.5.2.

```
export default defineEndpoint((router) => {
  router.post("/newsletter", async (req, res) => {
    const { email } = req.body;

    try {
      // ValidateEmail
      validateEmail(email);
      // Save email into database
      await saveEmail(email);
      res.status(200).send("Email saved");
    } catch (error) {
      if (isCustomError(error)) {
        res.status(error.statusCode).send(error.message);
      } else {
        res.status(500).send("Internal server error");
      }
    }
  });
});
```

Figure 33. Custom Directus endpoint for newsletter subscriptions

9.7. AWS CodePipeline implementation

The pipeline is responsible for reacting and updating all parts of the application in an automated process. The theory, terms and concepts introduced in 8.5.6 are put into practice in the current implementation. The pipeline CDK construct is available under the corresponding aws-cdk-lib package. The following figure (34) shows how pipeline is first constructed.

```
const pipeline = new codepipeline.Pipeline(
  this,
  'pipeline',
  {
    pipelineName: 'AWS CodePipeline',
   }
);
```

Figure 34. Creating CodePipeline in CDK

A pipeline does nothing on its own. For a pipeline to work and react to triggers, it needs to have defined stages and stages must have actions associated to perform a task. The code of this project lives in a GitHub repository. Connection with GitHub must be established, and permissions must be sufficient for AWS to be granted access to the repository. Luckily, AWS CodePipeline CDK constructs are provided as layer 2 constructs that automates the necessary

background steps of connecting and creating a webhook that listens for events in a Github repository. Code that represents this process is captured in figure 35 below.

```
const sourceStage = new codepipelineActions.GitHubSourceAction({
  actionName: 'Checkout',
  branch: 'main',
  oauthToken: githubOauthToken,
  output: repositorySource,
  owner: 'broadgg',
  repo: 'broadlify',
  trigger: codepipelineActions.GitHubTrigger.WEBHOOK,
});

pipeline.addStage({
  actions: [sourceStage],
  stageName: 'Source',
});
```

Figure 35. Establishing connection between pipeline and Github as a pipeline action

Pipeline execution is now being invoked whenever the defined action (like a push event) is detected by GitHub in a specified branch – main in this case. Next, the project defines multiple output artifacts, as shown in figure 36, that will be used as output buckets from a build stage. The build action is constructed from yet another layer 2 construct that provides out-of-the-box integration with CodeBuild. CodeBuild is given instructions on what actions to perform with the source code via build specification in a proprietary format of AWS CodeBuild – buildspec file. Refer to Appendix 4 for the full content of the file.

```
const buildStage = new codepipelineActions.CodeBuildAction({
  actionName: 'Build',
  input: repositorySource,
  outputs: [
    apiOutput,
    reactOutput,
    nextjsClientOutput,
    remixBuildOutput,
    remixAssetsOutput,
  ],
  project: new codebuild.PipelineProject(this, 'project', {
    buildSpec: codebuild.BuildSpec.fromObject(
      buildspecs.createBuildBuildspec(),
    ),
    environment: {
      buildImage: codebuild.LinuxBuildImage.STANDARD_5_0,
    },
    projectName: 'Build',
  }),
});

pipeline.addStage({
  actions: [buildStage],
  stageName: 'Build',
});
```

Figure 36. Adding a new stage with CodeBuild action

Build stage outputs artifacts for each project (CMS, Lambdas, frontends) as archives (ZIP) and saves them to S3. Next stage is responsible for moving and, in some cases, extracting archives so they are stored in the correct S3 buckets so the next actor, the deploy stage, has permission sufficient permissions for accessing and further acting upon the source code.

The last step of the process, the deploy stage, is done by a CodeBuild action and is instructed what commands to execute and what actions to perform based on another buildspec file. This buildspec file and its commands are generated during runtime, based on the number of arguments provided. Its job is to update serverless Lambda functions, invalidate cache of the CloudFront CDN and to release an updated version of Elastic Beanstalk. Refer to Appendix 4 for the full code of the deploy stage.

9.8. Front-end starter kits

The following section will describe the implementation of front-end starter kits for this product. Each of the starter kits is a separate project as described in section 9.1.1 about the monorepo setup. The limited amount of time only permitted the implementation of minimal working

examples for every starter kit. The future prospect of these starter kits is to introduce folder structures and configurations that set a standard for further development and promote consistency across projects. Further development of starter kits should also include a standardized way of consuming data from the CMS. An example of a minimal starting kit is shown in figure 37 below, specifically the React (Vite) project.

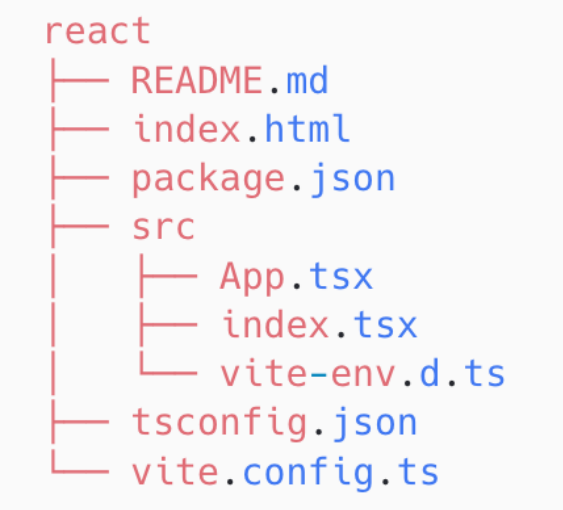

Figure 37. React (Vite) starter kit folder structure

9.9. CLI tool

The CLI tool implementation is not feature complete during the writing of this document as it is missing options to provide dynamic parameters. The CDK code is currently written in a single file under a single construct. During the implementation, it is quite easy to fall into creating unnecessary abstractions with the intention to simplify things but because of a lot of unknowns, all implementation, at the moment, is written in a sequence – one after another in a single file. Once all the defined requirements are implemented written infrastructure will be split into multiple constructs/stacks (as JavaScript classes) in their own files with parameters for all dynamic data so they can be provided from the upper scope. After this change is in place, the CLI tool can utilize JavaScript package called prompts that is available in the package registry, utilized to ask the user for all necessary data such as name of the project, project type (Git-based CMS or API-based CMS), deployment region and more. The current implementation of the CLI tool is simply cloning the repository to the local drive in the current working directory. The CLI tool is also not published in the package registry now because the repository is not yet public.

10. Security

The following section will address security concerns connected to this product. Planned security requirements are defined in section 8.2.7.1, however, not all of them are addressed in the product because of the limited timeframe the development team was given. What requirements were and were not implemented is explained in the conclusion (14.)

10.1. HTTPS (Hypertext Transfer Protocol Secure)

This product uses TLS certificates to facilitate secure communication over HTTP. Certificates are managed by AWS Certificate Manager (ACM) and are installed on individual services – each CDN resource, backend load balancer, and each custom API endpoint handler. The pairing of certificates and resources is handled by the CDK during deployment. Once the certificates are installed, AWS Certificate Manager handles renewals automatically.

10.2. Least required privileges for deployment entities

The deployment of the system sets permissions for deployed resources. Strict access policies have been implemented to mitigate unwanted changes. Implemented policies include specific actions for objects stored in deployment buckets, CloudFront distributions and serverless functions. The policy implemented for these resources is shown in figure 38 below.

```
new iam.PolicyStatement({
  actions: [
    'cloudfront:CreateInvalidation',
    'lambda:UpdateFunctionCode',
    's3:GetObject',
  ],
  resources: [
    reactDistributionArn,
    nextJsClientDistributionArn,
    remixDistributionArn,
    logLambda.functionArn,
    remixLambda.functionArn,
    `${apiBucket.bucketArn}/source`,
    `${remixBuildBucket.bucketArn}/source`,
  ],
}),
```

Figure 38. Deployment entity permissions

10.3. Two-factor authentication for AWS accounts

AWS Identity and Access Management provides a managed policy that denies all permissions for a user account except managing their own credentials or setting up a two-factor authentication device. This policy has been set up for every non-root user account and must be set up for every new user to keep the AWS organization secure.

10.4. Securely stored sensitive values

Sensitive values like the relational database credentials or GitHub OAuth token are stored in AWS Secrets Manager. Secret values stored in AWS Secrets Manager are encrypted and can be retrieved via encrypted API calls. Retrieving values in the CDK deployment pipeline is done by using the aws-secrets-manager construct as shown in figure 39 below.

```
const secrets = secretsmanager.Secret.fromSecretNameV2(
  this,
  'secrets',
  'broadlifySecrets'
)

const githubOauthToken = secrets.secretValueFromJson('githubOauthToken')
const rdsUsername = secrets.secretValueFromJson('rdsUsername')
const rdsPassword = secrets.secretValueFromJson('rdsPassword')
```

Figure 39. Secrets manager access in the CDK

10.5. Protection against client-side attacks in front-end template kits

Mitigating client-side vulnerabilities is done by alerting developers when the usage of unsafe practices is detected by using a linting tool - ESLint. An example is a linting rule that prevents the usage of “dangerouslySetInnerHTML” in React applications that is show in figure 40 below. The linting rule displays an error in the IDE and prevents the application from compiling and building production source code unless the issue is resolved. Other implementations of vulnerability mitigation include linter rules preventing unsafe Document Object Model (DOM) manipulations like using methods “write” and “innerHTML” on the document object.

```
<div>
  <h1
    // eslint(react/no-danger): Dangerous property 'dangerouslySetInnerHTML' found
    dangerouslySetInnerHTML={{
      __html: '<script>alert("Danger");</script>',
    }}
  />
</div>
```

Figure 40. Linter error in the IDE

10.6. Secure CMS access

Hashing passwords is handled by Directus, using the Argon2i cryptographic hashing algorithm. Argon2i is one of two variants of Argon2 and is optimized for high-performance password hashing.

Directus provides multiple settings for password strength policy as well as setting the number of maximum failed login attempts. The password policy is set to “Strong,” meaning that a password needs to have at least one uppercase letter, lowercase letter, number, special character and needs to be at least 8 characters long. Maximum number of login attempts is set to 5, once a user uses all of them, an administrator needs to unlock the account for further use.

10.7. DDoS protection

Resources distributed via Amazon’s CloudFront CDN are protected by AWS Shield, which is a managed Distributed Denial of Service protection service. It prevents attacks against the network and transport layers of the OSI model usually done by an automated bot network.

10.8. Unsecure dependencies

Most projects, that utilize the JavaScript programming language, depend on packages that are external to the developed system, usually distributed via npm. JavaScript packages might introduce vulnerabilities by simply becoming outdated and not adhering to newest security standards or by malicious intent of the package developer. To mitigate the risk of unsecure dependencies, developers need to (carefully) update or even replace dependencies that might introduce a vulnerability. Commands like npm outdated or npm doctor help locate faulty, out-of-date, or malicious dependencies. There are several other approaches and tools to automate this process and are further described in the maintenance section about out-of-date dependencies (13.1).

11. Maintenance

In this section, the process of maintenance of the developed software and its parts will be explained. Maintenance is a core part of the software development life cycle, and it is important both for the developers and for the stakeholders to keep the product up to date to preserve competitive advantage and not fall into a category of obsolete, legacy or even insecure systems. Broad identified scenarios that can happen and prepared a plan to combat them. The following subsections describe the learning goals that were set and whether they were achieved.

11.1. Out of date dependencies

The product itself is a development effort in the JavaScript ecosystem that relies on and utilizes a package manager and its repository of public and private packages that can be installed and used in any standard JavaScript based environment. The foundation is an open-source project called the Cloud Development Kit (CDK) developed by the AWS team. The CDK is also distributed and available as a package in the package registry. Any package in the registry has a version and developers can publish a new release that other developers depending on the package might need to adapt to. Packages most commonly follow semver versioning, with this assumption it should be theoretically possible to update to newer minor and patch versions without breaking the functionality of the software. In the Broad developers' team, a tool called npm-check-updates is frequently used to check and update to the latest dependency versions available if desired. However, this is a manual operation and would require writing a scheduled action to automate the process. Luckily, there is a fully automated tool, dependantbot, that scans the dependencies of a project and opens a pull request with a suggested dependency (or dependencies) bump. Dependabot is an open-source software developed by GitHub and offers rich configuration options to accommodate common workflows such as intervals, scheduling, types of updates (major, minor, patch) and others. Out of date dependencies might also introduce major security vulnerabilities and therefore need to be attended to in a structured and repetitive manner.

11.2. Adoption of new software to landscape changes and product ownership

A product is at risk of being gradually considered legacy software if it does not, during its lifecycle, adopt changes and new emerging tools in their respective technological landscape. Similarly, if there is no attention paid to the rollout of new tools and the process of phasing out the old ones goes unnoticed it might have catastrophic consequences for the business and

potentially risk losing customers due to loss of trust. Considering the product that Broad is investing in and its nature of not being a standalone solution but instead fully relying on a cloud vendor, AWS in this case, the product must be maintained over time and the latest changes and technologies must be adopted to prevent downtimes and security vulnerabilities. AWS does not provide any guarantee of backwards compatibility for any of its services. The only guarantees that AWS provides are so called service level agreement guarantees which differ between all AWS offered services. Many of the service guarantees are only concerned about uptime percentage. However, if, theoretically speaking, AWS would push an update with an unexpected breaking change, without effectively communicating it with its customer it would severely impact parts of the customer base resulting in its SLAs violation. To combat any of the catastrophic scenarios, Broad as a company has every intention to both educate its current employees in the cloud landscape as it thrives to establish a culture of shared ownership and potentially establish a dedicated infrastructure team overseeing all cloud operations such as uptime, retention, redundancy, load distribution and more.

12. Conclusion & future work

The conclusion will reflect upon the project and provide answers to the questions asked in the problem statement (4.). The developed product fulfilled milestones presented as stage 1 (7.1) and stage 2 (7.2) except database migrations from stage 2. This project provided a valuable learning experience about creating a platform for hosting web applications that has potential to become a core product of Broad. The development team has managed to implement a deployment pipeline using infrastructure as code backed by the AWS Cloud Development Kit. This product connects several AWS resources to create a streamlined development experience of building websites backed by a content management system. The outcome of this project is considered a success by the development team consisting of Marek Vargovcik and Michal Cic.

The following paragraphs will answer problem statement questions.

- How can we lower the costs of hosting service infrastructure?

The cost of the hosting infrastructure can be lowered by investing time and resources to develop an in-house hosting solution that replaces hosting services like Netlify with bundled AWS services.

- How can we ensure deployment of websites and their associated resources without build-time errors and subsequent management of said websites and resources?

The deployment of websites was reinforced by introducing testing suites, linting and formatting infrastructure that mitigate the possibility of build-time errors and ensure deployment of correctly functioning products. The management of websites and resources was achieved by integrating automated and transparent build processes that can be modified based on specific project needs and ensure a clear path to further continuous development and improvement of the end product.

- How can we make the process of developing and deploying websites automated?

The process of developing and deploying websites was automated by implementing and documenting processes and tools like CodeBuild and CodePipeline. The combination and correct configuration of these tools provided a clear path to automation of deployment processes.

- How can we bundle related services into a single solution and lower the barrier of entry for developing and deploying websites?

Bundling related services was accomplished using tools such as Amazon Cloud Development Kit to link related services together and create a streamlined process of deployment for different services.

12.1. Future work

The following section contains requirements that, due to the limited amount of time allocated, were not implemented into the product at the time of writing this document. Most unimplemented requirements are related to stage 3 as described in the project stages section (7.). The following requirements are planned to be worked on in the future by Broad's developers.

- DDoS protection for the backend
- Data backups
- 2FA for the CMS
- Public/private subnets for provisioned resources
- Testing
- CLI tooling
- Documentation

13. Appendix 1

User stories

User stories are an important part of the requirement gathering process, because of the stakeholder action insight they provide. User stories help us identify high level goals from the perspective of the user.

Developer

As a developer, I want to have an out-of-the-box working template with a presentation layer and a content management system, so I can initialize new projects quickly.

As a developer, I want to connect a remote repository to this system, so that every commit that I push to this remote repository is built and deployed automatically.

As a developer, I want to be sure that the commit I just pushed is built without errors, so the website is accessible to users and does not crash.

As a developer, I want to be sure that the website is running and accessible, so users can always connect to it.

Editor

As an editor, I want to preview content as it will be displayed to the user, so I can ensure its correctness and visual cohesion.

As an editor, I want to schedule when changes are published.

As an editor, I want to publish multiple changes at once.

As an editor, I want to reset my password, so I could access my account if I forgot my password.

As an editor, I want to update information in my editor profile.

Administrator

As an administrator, I want to restrict user permissions, so junior editors cannot delete content.

As an administrator, I want to remove editor accounts, so people that do not work here anymore do not have access to our content management system.

As an administrator, I want to create new editor accounts, so I can give accounts to people that we might hire in the future.

14. Appendix 2

Content management system costs breakdown

Server usage for AWS is calculated based on hourly rate of the server running, data transfer in and out of the server, load balancing if multiple servers are utilized and server monitoring. [58] Storage usage for AWS is calculated based on the size of stored objects, how long an object has been stored and data transfer in and out of the storage. [59] Database usage for AWS is calculated based on inputs and outputs from the database, storage size, data transfer and backup storage. [60]

15. Appendix 3

15.1. TypeScript base configuration – tsconfig.base.json

```
{
  "compilerOptions": {
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "incremental": true,
    "isolatedModules": true,
    "resolveJsonModule": true,
    "skipLibCheck": true,
    "strict": true,
  },
  "exclude": [
    "node_modules"
  ],
}
```

15.2. ESLint base configuration – eslintrc.base.json

```
{
  "extends": [
    "canonical",
    "canonical/typescript",
    "canonical/node",
    "plugin:prettier/recommended"
  ],
  "ignorePatterns": [
    "dist"
  ],
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": 2021,
    "project": [
      "**/tsconfig.json"
    ],
    "sourceType": "module"
  },
  "plugins": [
    "sort-destructure-keys",
    "sort-exports"
  ],
}
```

```
{
  "rules": {
    "@typescript-eslint/explicit-member-accessibility": "off",
    "@typescript-eslint/naming-convention": "off",
    "@typescript-eslint/no-extraneous-class": "off",
    "@typescript-eslint/no-misused-promises": [
      "error",
      {
        "checksVoidReturn": false
      }
    ],
    "@typescript-eslint/no-shadow": "off",
    "@typescript-eslint/no-unused-vars": "warn",
    "@typescript-eslint/triple-slash-reference": "off",
    "@typescript-eslint/unbound-method": "off",
    "canonical/destructuring-property-newline": "off",
    "canonical/export-specifier-newline": "off",
    "canonical/filename-match-exported": "off",
    "canonical/filename-match-regex": "off",
    "canonical/id-match": "off",
    "canonical/import-specifier-newline": "off",
    "class-methods-use-this": "off",
    "consistent-return": "off",
    "fp/no-class": "off",
    "fp/no-this": "off",
    "id-length": "off",
    "id-match": "off"
  }
}
```

```
{
  "rules": {
    "import/extensions": "off",
    "import/no-cycle": "off",
    "import/no-extraneous-dependencies": [
      "error",
      {
        "bundledDependencies": true
      }
    ],
    "import/no-named-default": "off",
    "import/no-unassigned-import": "off",
    "import/order": [
      "error",
      {
        "alphabetize": {
          "caseInsensitive": true,
          "order": "asc"
        },
        "newlines-between": "always"
      }
    ],
    "import/unambiguous": "off",
    "max-nested-callbacks": "off",
    "no-catch-shadow": "off",
    "no-console": "warn",
    "no-else-return": "off",
    "node/callback-return": "off",
    "node/no-extraneous-import": "off",
    "node/no-process-env": "off",
    "padding-line-between-statements": "off",
    "prefer-promise-reject-errors": "off",
  }
}
```

```
{
  "rules": {
    "prettier/prettier": [
      "error",
      {
        "jsxSingleQuote": true,
        "singleQuote": true,
        "trailingComma": "all"
      }
    ],
    "require-unicode-regexp": "off",
    "sort-destructure-keys/sort-destructure-keys": [
      "error",
      {
        "caseSensitive": false
      }
    ],
    "sort-exports/sort-exports": "error",
    "sort-imports": [
      "error",
      {
        "allowSeparatedGroups": true,
        "ignoreCase": true,
        "ignoreDeclarationSort": true,
        "ignoreMemberSort": false,
        "memberSyntaxSortOrder": [
          "none",
          "all",
          "multiple",
          "single"
        ]
      }
    ],
    "spaced-comment": [
      "error",
      "always",
      {
        "markers": [
          "/"
        ]
      }
    ],
    "unicorn/no-array-reduce": "off",
    "unicorn/prefer-object-from-entries": "off",
    "unicorn/prevent-abbreviations": "off"
  }
}
```

16. Appendix 4

16.1. Build specification (buildspec.yml)

```
const createBuildBuildspec = () => ({
  artifacts: {
    'secondary-artifacts': {
      api: {
        'base-directory': 'apps/api/dist',
        files: ['*.js'],
      },

      nextjsClientOutput: {
        'base-directory': 'apps/nextjs-client/dist',
        files: ['**/*'],
      },
      reactOutput: {
        'base-directory': 'apps/react/dist',
        files: ['**/*'],
      },
      remixAssetsOutput: {
        'base-directory': 'apps/remix/public',
        files: ['**/*'],
      },
      remixBuildOutput: {
        'base-directory': 'apps/remix/build',
        files: ['**/.js'],
      },
    },
  },
  phases: {
    build: {
      commands: [
        'npm run api:build',
        'npm run react:build',
        'npm run nextjs-client:build',
        'npm run remix:build',
      ],
    },
    install: {
      commands: ['npm install -g npm@latest', 'npm install'],
      'runtime-versions': {
        nodejs: 14,
      },
    },
  },
  version: 0.2,
});
```

```
type CreateDeployBuildspecParams = {
  distributions: Array<{
    id: string;
    path: string;
  }>;
  elasticbeanstalks: Array<{
    bucket: string;
    key: string;
    name: string;
  }>;
  functions: Array<{
    bucket: string;
    key: string;
    name: string;
  }>;
};

const createDeployBuildspec = ({
  distributions,
  elasticbeanstalks,
  functions,
}: CreateDeployBuildspecParams) => {
  const lambdaUpdateCommands = functions.map(
    (fn) =>
      `aws lambda update-function-code --function-name ${fn.name} --s3-bucket ${fn.bucket} --s3-key ${fn.key} --
region us-east-1`,
  );
  const distributionInvalidationCommands = distributions.map(
    (distribution) =>
      `aws cloudfront create-invalidation --distribution-id ${distribution.id} --paths "${distribution.path}"`,
  );
  const elasticbeanstalkCommands = elasticbeanstalks.map(
    (elasticbeanstalk) =>
      `aws elasticbeanstalk create-application-version --application-name ${
        elasticbeanstalk.name
      } --version-label ${Date.now()} --source-bundle S3Bucket=${
        elasticbeanstalk.bucket
      },S3Key=${elasticbeanstalk.key}`,
  );
  return {
    phases: {
      build: {
        commands: [
          ...lambdaUpdateCommands,
          ...distributionInvalidationCommands,
          ...elasticbeanstalkCommands,
        ],
      },
    },
    version: 0.2,
  };
};
```

16.2. Deploy stage

```
const deployActionProject = new codebuild.PipelineProject(this, 'deploy', {
  buildSpec: codebuild.BuildSpec.fromObject(
    buildspecs.createDeployBuildspec({
      distributions: [
        {
          id: reactDistribution.distributionId,
          path: '/*',
        },
        {
          id: remixDistribution.distributionId,
          path: '/*',
        },
        {
          id: nextjsClientDistribution.distributionId,
          path: '/*',
        },
      ],
      elasticbeanstalks: [
        {
          bucket: backendBucket.bucketName,
          key: 'output.zip',
          name: env.applicationName,
        },
      ],
      functions: [
        {
          bucket: apiBucket.bucketName,
          key: 'source',
          name: logLambda.functionName,
        },
        {
          bucket: remixBuildBucket.bucketName,
          key: 'source',
          name: remixLambda.functionName,
        },
      ],
    }),
  ),
  environment: {
    buildImage: codebuild.LinuxBuildImage.STANDARD_5_0,
  },
  projectName: 'Deploy',
});
```

```
const reactDistributionArn =
`arn:aws:cloudfront::${props.accountId}:distribution/${reactDistribution.distributionId}`;
const nextJsClientDistributionArn =
`arn:aws:cloudfront::${props.accountId}:distribution/${nextjsClientDistribution.distributionId}`;
const remixDistributionArn =
`arn:aws:cloudfront::${props.accountId}:distribution/${remixDistribution.distributionId}`;

deployActionProject.addToRolePolicy(
  new iam.PolicyStatement({
    actions: [
      'lambda:UpdateFunctionCode',
      'cloudfront:CreateInvalidation',
      'elasticbeanstalk:CreateApplicationVersion',
      's3:GetObject',
    ],
    resources: [
      reactDistributionArn,
      nextJsClientDistributionArn,
      remixDistributionArn,
      logLambda.functionArn,
      remixLambda.functionArn,
      `${apiBucket.bucketArn}/source`,
      `${remixBuildBucket.bucketArn}/source`,
    ],
  }),
);

const deployAction = new codepipelineActions.CodeBuildAction({
  actionName: 'Deploy',
  input: repositorySource,
  project: deployActionProject,
});

pipeline.addStage({
  actions: [deployAction],
  stageName: 'Deploy',
});
```

17. Bibliography

[1] "The Netlify Platform - Products for Modern Web Development." https://www.netlify.com/products/ (accessed May 01, 2022).

[2] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, New Edition. Englewood Cliffs, NJ: Prentice Hall, 1992, p. 282.

[3] "Reduce server response times (TTFB) - Chrome Developers." https://developer.chrome.com/docs/lighthouse/performance/time-to-first-byte/ (accessed Jun. 10, 2022).

[4] "Everyone can now run JavaScript on Cloudflare with Workers." https://blog.cloudflare.com/cloudflare-workers-unleashed/ (accessed Jun. 10, 2022).

[5] "Cloud CDN release notes | Google Cloud." https://cloud.google.com/cdn/docs/release-notes (accessed Jun. 10, 2022).

[6] "Introducing CloudFront Functions – Run Your Code at the Edge with Low Latency at Any Scale | AWS News Blog." https://aws.amazon.com/blogs/aws/introducing-cloudfront-functions-run-your-code-at-the-edge-with-low-latency-at-any-scale/ (accessed Jun. 10, 2022).

[7] "netlify/netlify-cms: A Git-based CMS for Static Site Generators." https://github.com/netlify/netlify-cms/ (accessed Jun. 09, 2022).

[8] "• Chart: Amazon Leads $180-Billion Cloud Market | Statista." https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ (accessed Jun. 07, 2022).

[9] "Amazon Web Services - AWS in the Enterprise." https://aws.amazon.com/enterprise/ (accessed Jun. 07, 2022).

[10] "Amazon EC2 customers." https://aws.amazon.com/ec2/customers/ (accessed Jun. 07, 2022).

[11] "Microsoft Customer Stories." https://customers.microsoft.com/en-us/ (accessed Jun. 10, 2022).

[12] "Customers | Google Cloud." https://cloud.google.com/customers (accessed Jun. 10, 2022).

[13] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun ACM*, vol. 62, no. 12, pp. 44–54, Nov. 2019.

[14] C. Kilcioglu, J. Rao, A. Kannan, and R. P. McAfee, "Usage Patterns and the Economics of the Public Cloud," in *Proceedings of the 26th International Conference on World Wide Web*, Apr. 2017, pp. 83–91.

[15] G. McGrath and P. R. Brenner, “Serverless computing: design, implementation, and performance,” in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, Jun. 2017, pp. 405–410, doi: 10.1109/ICDCSW.2017.36.

[16] S. Chauhan, “We Burnt $72K testing Firebase + Cloud Run and almost went Bankrupt,” Dec. 08, 2020. https://blog.tomilkieway.com/72k-1/ (accessed Jun. 06, 2022).

[17] “What is Infrastructure as Code (IaC)?” https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac (accessed Jun. 11, 2022).

[18] “Constructs - AWS Cloud Development Kit (CDK) v2.” https://docs.aws.amazon.com/cdk/v2/guide/constructs.html (accessed Jun. 09, 2022).

[19] “Getting started with the AWS CDK - AWS Cloud Development Kit (CDK) v2.” https://docs.aws.amazon.com/cdk/v2/guide/getting_started.html (accessed Jun. 11, 2022).

[20] “Introduction - The jsii reference.” https://aws.github.io/jsii/ (accessed Jun. 11, 2022).

[21] “Pricing · Plans for every developer.” https://github.com/pricing (accessed Jun. 11, 2022).

[22] “Bitbucket - Pricing | Atlassian.” https://www.atlassian.com/software/bitbucket/pricing (accessed Jun. 11, 2022).

[23] “GitLab Pricing.” https://about.gitlab.com/pricing/ (accessed Jun. 11, 2022).

[24] “strapi/strapi: 🚀 Open source Node.js Headless CMS to easily build customisable APIs.” https://github.com/strapi/strapi (accessed Jun. 12, 2022).

[25] “strapi.io/.” https://strapi.io/ (accessed Jun. 12, 2022).

[26] “Introduction | Directus Docs.” https://docs.directus.io/getting-started/introduction/ (accessed Jun. 12, 2022).

[27] “Why KeystoneJS - Keystone 6.” https://keystonejs.com/why-keystone#features (accessed Jun. 12, 2022).

[28] A. Mishra, *Amazon Web Services for Mobile Developers: Building Apps with AWS*, 1st ed. Indianapolis, NY: Sybex, 2017, p. 792.

[29] A W S, “How Amazon VPC works.” https://docs.aws.amazon.com/vpc/latest/userguide/how-it-works.html (accessed Jun. 09, 2022).

[30] A W S, “Building a Scalable and Secure Multi-VPC AWS Network Infrastructure.” https://docs.aws.amazon.com/whitepapers/latest/building-scalable-secure-multi-vpc-network-infrastructure/welcome.html (accessed Jun. 09, 2022).

[31] “What Is a Relational Database | Oracle.” https://www.oracle.com/database/what-is-a-relational-database/ (accessed Jun. 13, 2022).

[32] “Amazon Aurora storage and reliability - Amazon Aurora.” https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Aurora.Overview.StorageReliability.html (accessed Jun. 13, 2022).

[33] R. Aapakallio-Autio, “From Elastic Beanstalk to Lambda: A comparative case study on the AWS tools,” Master thesis, 2021.

[34] A W S, “AWS Elastic Beanstalk Documentation.” https://docs.aws.amazon.com/elastic-beanstalk/index.html (accessed Jun. 07, 2022).

[35] A W S, “What is AWS CodePipeline?” https://docs.aws.amazon.com/codepipeline/latest/userguide/welcome.html (accessed Jun. 14, 2022).

[36] A W S, “CodePipeline concepts.” https://docs.aws.amazon.com/codepipeline/latest/userguide/concepts.html (accessed Jun. 14, 2022).

[37] T. Fadhilah Iskandar, M. Lubis, T. Fabrianti Kusumasari, and A. Ridho Lubis, “Comparison between client-side and server-side rendering in the web development,” *IOP Conf. Ser.: Mater. Sci. Eng.*, vol. 801, no. 1, p. 012136, May 2020, doi: 10.1088/1757-899X/801/1/012136.

[38] Remix, “Remix | Philosophy.” https://remix.run/docs/en/v1/pages/philosophy (accessed Jun. 12, 2022).

[39] O S S, “Why Vite.” https://vitejs.dev/guide/why.html#the-problems (accessed Jun. 11, 2022).

[40] O S S, “esbuild - An extremely fast JavaScript bundler.” https://esbuild.github.io/ (accessed Jun. 11, 2022).

[41] “Next.js - Wikipedia.” https://en.wikipedia.org/wiki/Next.js (accessed Jun. 12, 2022).

[42] npm, “next - npm.” https://www.npmjs.com/package/next (accessed Jun. 12, 2022).

[43] Vercel, “Basic Features: Pages | Next.js.” https://nextjs.org/docs/basic-features/pages (accessed Jun. 12, 2022).

[44] Vercel, “Showcase | Next.js.” https://nextjs.org/showcase (accessed Jun. 12, 2022).

[45] Remix, “Remix | Technical Explanation.” https://remix.run/docs/en/v1/pages/technical-explanation (accessed Jun. 12, 2022).

[46] Remix, “Remix | Routing.” https://remix.run/docs/en/v1/guides/routing (accessed Jun. 12, 2022).

[47] Github, “New year, new GitHub: Announcing unlimited free private repos and unified Enterprise offering,” Jan. 07, 2019. https://github.blog/2019-01-07-new-year-new-github/ (accessed Jun. 06, 2022).

[48] D. Kubitza, M. Bockmann, and D. Graux, “Towards Semantically Structuring GitHub,” *CEUR Workshop Proc*, pp. 141–144, Oct. 2019.

[49] Gitlab, “Install self-managed GitLab.” https://about.gitlab.com/install/ (accessed Jun. 06, 2022).

[50] Github, “The tools you need to build what you want.” https://github.com/features (accessed Jun. 06, 2022).

[51] C. Tozzi, “Open Source’s Killer Features: What Makes Open Source So Popular?,” Nov. 27, 2017. https://www.channelfutures.com/strategy/open-sources-killer-features-what-makes-open-source-so-popular (accessed Jun. 06, 2022).

[52] Wikimedia Foundation Inc., “npm (software).” https://en.wikipedia.org/wiki/Npm_(software) (accessed Jun. 06, 2022).

[53] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, “On the Impact of Micro-Packages: An Empirical Study of the npm JavaScript Ecosystem,” *arXiv*, 2017, doi: 10.48550/arxiv.1709.04638.

[54] Npm Inc., “npm publish.” https://docs.npmjs.com/cli/v8/commands/npm-publish (accessed Jun. 06, 2022).

[55] Npm Inc., “npm init.” https://docs.npmjs.com/cli/v8/commands/npm-init (accessed Jun. 06, 2022).

[56] “TypeScript: JavaScript With Syntax For Types.” https://www.typescriptlang.org/ (accessed Jun. 13, 2022).

[57] “Getting Started with ESLint - ESLint - Pluggable JavaScript linter.” https://eslint.org/docs/user-guide/getting-started (accessed Jun. 13, 2022).

[58] “Amazon EC2 Pricing - Amazon Web Services.” https://aws.amazon.com/ec2/pricing/ (accessed Jun. 12, 2022).

[59] “Amazon S3 Simple Storage Service Pricing - Amazon Web Services.” https://aws.amazon.com/s3/pricing/ (accessed Jun. 12, 2022).

[60] “Amazon Aurora Pricing | MySQL PostgreSQL Relational Database | Amazon Web Services.” https://aws.amazon.com/rds/aurora/pricing/ (accessed Jun. 12, 2022).