Moving a Three(+) Tier app to Cloud-Native

Abstract

Deployment of layered applications can be cumbersome. How to ensure the integrity of the components on managed infrastructure, how to ensure the gradual update of network configurations post-deployment, how to size the servers, and prepare the system for handling a large volume of requests, while keeping the highest standards of application security?

In my paper, I am detailing a real-world application (based on my BSc thesis), built upon the Three(+) Tier Architecture[1] model, deployed to Cloud-Native on Amazon Web Services. I am writing about many of the considerations that took place when I repurposed and rebuilt my application to be aligned with the latest standards of enterprise-grade software products. Mentioning containerization, Continuous Integration and Continuous Delivery, serverless databases and caches, the Elastic Container Service platform including load balancing and autoscaling configuration as well as how to ship native (C++) code to the cloud.

I am writing some bits and pieces about security by design principles and how I achieved a green cloud security audit result. Since the project is still in private alpha, I am closing with a financial evaluation, some calculations and some test results regarding the solution's performance and scalability.

About the author

Árpad Kiss is a student at Óbuda University John von Neumann Faculty of Informatics, Budapest, Hungary, in Business IT MSc, from 2023. He holds a BSc in Engineering IT from the same faculty, from 2014. He is the Chief Executive Officer of GreenEyes Artificial Intelligence Services, LLC., Lewes, Delaware, United States of America.

arpad@greeneyes.ai
https://www.linkedin.com/in/arpi11

Consultants & contributors

Balázs Buri, AWS Cloud Architect / Engineer, Zürich, Switzerland
https://www.linkedin.com/in/balazsburi/

Klára Méndez-Osgyáni BSc, Freelance Software Developer at GreenEyes Artificial Intelligence Services, LLC., Hungary
https://www.linkedin.com/in/klara-anita-mendez-osgyani/

Gergő Barta, PhD, Senior Manager & AI lead at Deloitte, Budapest, Hungary
https://www.linkedin.com/in/gerg%C5%91-barta-ph-d-953b9976/

[1] What is three-tier architecture? (https://www.ibm.com/topics/three-tier-architecture), last visit: 12th of October, 2023

Moving a Three(+) Tier app to Cloud-Native

Table of contents

1) Introduction....4
1.1) The prototype from 2014....4
1.1.1) Purpose of the system....4
1.1.2) System model....5
1.2) Evolution of Deployment models on Public and Private infrastructures, 2009-2023....6
1.2.1) Traditional on-premises deployment (Pre-2009):....6
1.2.2) Early cloud adoption (2009-2013):....6
1.2.3) Hybrid cloud (2013-2017):....6
1.2.4) Containerization and Kubernetes (2014-2018):....6
1.2.5) Serverless computing (2016-2019):....6
1.2.6) Edge computing (2018-2023):....6
1.2.7) Multi-cloud strategy (2019-2023):....7
1.2.8) DevOps and CI/CD (2010s-2023):....7
1.2.9) Security-first approach (2010s-2023):....7
1.2.10) AI and Machine Learning integration (2010s-2023):....7
1.2.11) Nowadays and the future....7
1.3) Introduction to Amazon Web Services and Cloud-Native....7
2) System specification....9
2.1) System components....9
2.1.1) System architecture of the prototype....9
2.1.2) REST API....10
2.1.3) Database and cache....10
2.1.4) Object storage....10
2.1.5) A native (C++) CLI as asynchronous worker....10
2.2) Cloud components, the building blocks....11
2.2.1) Virtual Private Cloud, Route53 and Certificate Manager....11
2.2.2) Elastic Container Service and AWS Fargate....11
2.2.3) Aurora and ElastiCache....11
2.2.4) Simple Storage Service and Elastic File System....12
2.2.5) AWS Lambda....12
2.3) Simplified network topology and access configuration....13
2.4) Continuous- Integration and Delivery from developer's angle....14
2.4.1) Containerization and Elastic Container Registry....15
2.4.2) Version control in Gitlab and usage of GitlabCI....15
2.5) Reimplementation and changes for Version 1.0, briefly....15
3) Deploying the system to AWS Public Cloud....17
3.1) Security considerations and vulnerability remediation strategy....17
3.2) Zero-time deployment, load balancing and autoscaling in ECS....18
3.3) Parallelizing asynchronous worker calls on Lambda....18
3.4) The dilemma on deploying the components to multiple availability zones....19

By: Árpád Kiss, Óbuda University John von Neumann Faculty of Informatics, Business IT MSc 2023
Contact: arpad@greeneyes.ai; Version 1.2; Last revision: 21th of October, 2023.

4) Evaluating the results..... 20
4.1) Financial implications..... 20
4.2) System response times on various datasets..... 23
5) Summary and closing..... 26
6) References..... 27

Moving a Three(+) Tier app to Cloud-Native

1) Introduction

1.1) The prototype from 2014

During my studies I specialised in computer vision, and as I was a hobby and part time web developer for over a decade by then, I came up with an idea of a client-server application[2] about a pictorial search engine: identifying books from a pictorial database by user generated inputs.

For this I designed and developed a Three Tier application server (API) in NodeJS (v0.11), a vector database storing clustered CIELAB[3] responses of 18 dimensions, in memory, and SURF[4] (Haar-wavelet) responses of 64 dimensions on disk, of stored and processed book covers. The API was able to extract these vectors from raw book cover photo uploads, and perform a recognition operation against stored book cover photos, by performing further operations on the uploaded photo. These operations were such as edge detection, polygon approximation, dimensionality reduction (to estimate the detected book cover's orientation), and orthographic projection to transform a detected rectangles to a flattened shape. Once the flattened shape was extracted and the vectors were (partially) computed, the API performed a filter in memory, using the CIELAB responses, then performed a k-Nearest Neighbors[5] search query on the reduced group of potential matches.

For the image processing and machine learning algorithms I implemented a small, lightweight, and parameterized CLI in C++ (v11), using libs like boost[6] and OpenCV[7], besides implementing plenty of low-level algorithms on my own. The client was an Objective-C, native iOS mobile app, but that is not a topic of this paper. If you seek further details, see my thesis[2].

1.1.1) Purpose of the system

As of 2023, with the emergence of generative AI new horizons were opened for the ordinary people, making them able to use advanced technologies for all sorts of creative works. These technologies are easily accessible and the practical use-cases keep growing. AI is no longer the privilege of the educated and the elite, who are able to fund the research and operations of these models, while exploiting the technology's capabilities for their benefit, sometimes even in a questionable way. More and more regulations around the topic are getting addressed by global legislators, see for example the AI Act by the European Union[9]. As we see by ChatGPT and other Large Language Models, these new models require enormous amounts of computing power, sometimes even consuming 33.000 household's (a rough estimate) electrical power a day, just for a single instance of a model. This phenomenon raised some concerns in the environmentalist tech communities, whether the general approach, which is using interpreted languages such as Python, is affordable[10] for the long run, or should we focus our energy and money using a low-level, hardware-optimised way of implementing the computational operations of these per say, highly generalizable algorithms.

Switching to the topic of computer vision: in 2023, the general approach to handle computer vision classification tasks is to use an AlexNet[11]-like Convolutional Neural Network, promising to be able to store and distinguish between 1.000 classes on a single, 8 GB VRAM GPU. However, this approach is only useful if a large volume of data is available at the training time. My original approach only required one sample from an object for training purposes, besides instead of classification I was using clustering, and it was proven to be extremely scalable on handling rapidly

changing databases, like in the case of a typical web application. In some cases when the training data is feature rich, it provides an environmentally viable alternative to the current approach having a reserved computational unit, rather than using only an on-demand provisioned computing resource. My solution, especially after the reworks and extensions, can provide similar results to a CNN system, about and above 98% of accuracy.

1.1.2) System model

The flow of the Recognition Operation, used to model the system:

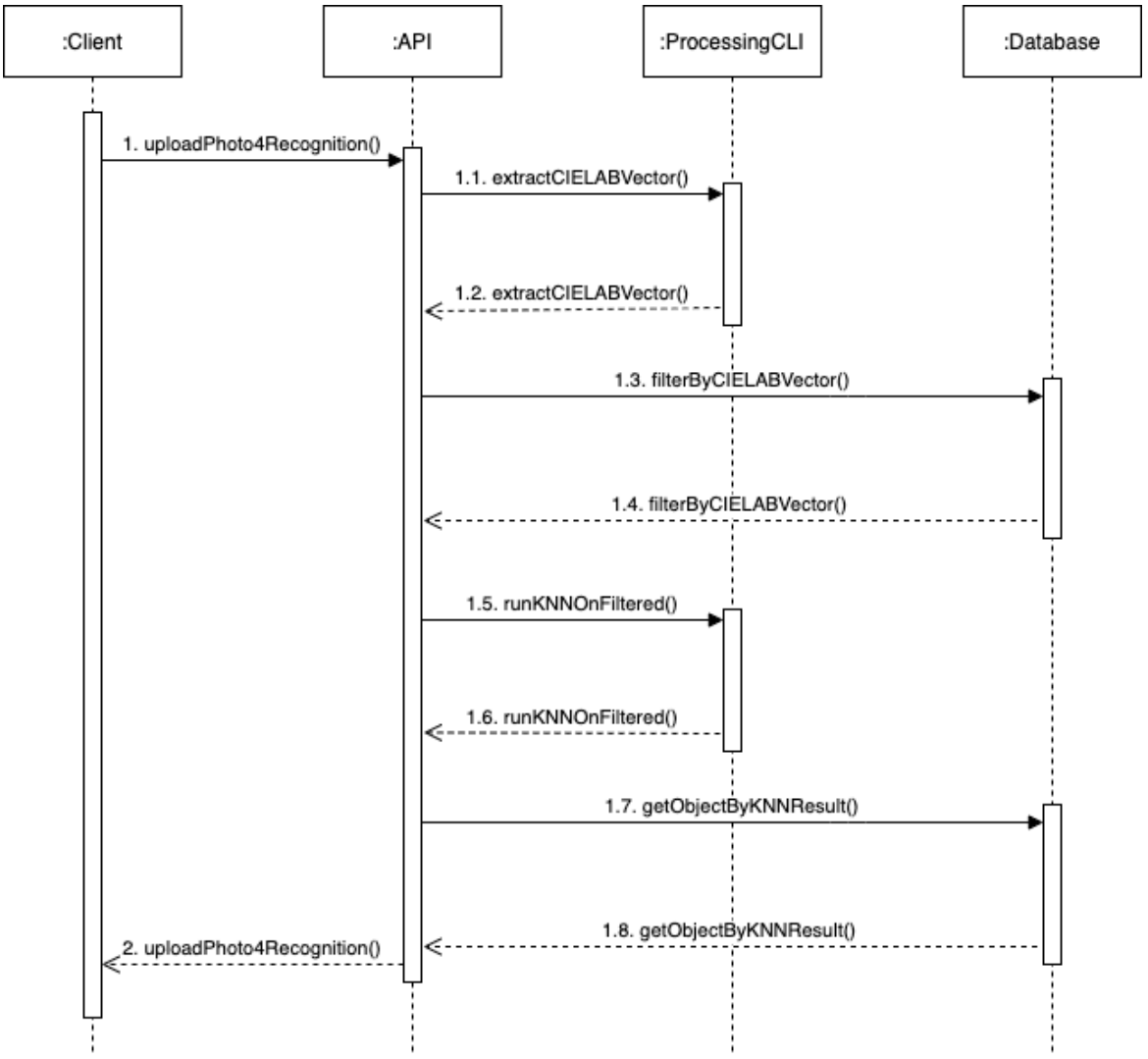

Figure 1.: Recognition Operation flow diagram

Legend for Figure 1:

- Client: any program which can communicate over HTTP protocol
- API: a REST API, written in NodeJS.
- ProcessingCLI: a compiled (to x86) C++ CLI to handle image processing and machine learning workloads.
- Database: an in-memory vector storage

Moving a Three(+) Tier app to Cloud-Native

1.2) Evolution of Deployment models on Public and Private infrastructures, 2009-2023

The evolution of software deployment models on public and private infrastructures from 2009 to 2023 has been marked by significant advancements and shifts in approaches. Below is an overview of the key changes and trends during this period:

1.2.1) Traditional on-premises deployment (Pre-2009):

Before 2009, software deployment primarily revolved around traditional on-premises models where organisations hosted and maintained their software on their own infrastructure.

1.2.2) Early cloud adoption (2009-2013):

The early 2010s saw the rapid rise of cloud computing, with Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP) emerging as major players. Public cloud infrastructure gained traction for software deployment due to benefits like scalability, cost-efficiency, and flexibility. Private cloud deployments within organisations also began to gain momentum, allowing more control and security compared to the public cloud.

1.2.3) Hybrid cloud (2013-2017):

The concept of the hybrid cloud gained popularity, enabling organisations to use a mix of on-premises, private cloud, and public cloud infrastructure for deploying and managing their software. This approach allowed for greater flexibility, workload optimization, and cost efficiency by leveraging both on-premises and cloud resources.

1.2.4) Containerization and Kubernetes (2014-2018):

Containerization technologies, especially Docker, gained prominence, enabling software to be packaged with all its dependencies and run consistently across different environments. Kubernetes emerged as the leading container orchestration tool, providing automation and management capabilities for deploying and scaling applications in various environments, including public and private clouds.

1.2.5) Serverless computing (2016-2019):

Serverless computing, exemplified by AWS Lambda, Azure Functions, and Google Cloud Functions, gained popularity, allowing developers to build and deploy applications without managing server infrastructure. This model offered rapid scaling, cost savings, and a "pay-as-you-go" pricing structure.

1.2.6) Edge computing (2018-2023):

Edge computing gained traction due to the proliferation of IoT devices and the need for low-latency processing closer to data sources. Software deployment models extended to the edge, enabling applications to run on edge devices or edge servers, enhancing performance and real-time decision-making.

1.2.7) Multi-cloud strategy (2019-2023):

Organisations increasingly adopted multi-cloud strategies, leveraging multiple cloud providers to mitigate vendor lock-in, improve redundancy, and optimise costs. Software deployment models evolved to support seamless integration and deployment across various cloud providers.

1.2.8) DevOps and CI/CD (2010s-2023):

The integration of DevOps practices and continuous integration/continuous deployment (CI/CD) pipelines became the standard for software deployment. Automation and collaboration improved the speed and efficiency of deploying software across various infrastructures, both public and private.

1.2.9) Security-first approach (2010s-2023):

Security became a fundamental consideration in software deployment models, with a focus on securing applications and data across public and private infrastructures. Encryption, identity management, and compliance requirements played a critical role in shaping deployment strategies.

1.2.10) AI and Machine Learning integration (2010s-2023):

AI and machine learning applications became more prevalent in software, influencing deployment models to accommodate the unique requirements of these technologies.

1.2.11) Nowadays and the future

Overall, the period from 2009 to 2023 witnessed a transformative journey in software deployment, embracing cloud technologies, containerization, serverless computing, edge computing, and a strong focus on automation, security, and agility to meet the evolving needs of modern applications. I think, for financial and environmental-impact reasons, triggered by the emergence of mass-adopted, large AI models, more and more companies will consider moving away from the classic, x86 platform to the more affordable ARM platform, custom hardware solutions such as FPGAs and NPUs, and the emergence of these platforms will be resulting in large amount of new tools and technologies.

1.3) Introduction to Amazon Web Services and Cloud-Native

Amazon Web Services (AWS[13]) is a leading cloud computing platform offered by Amazon, providing a vast array of services and solutions to individuals, businesses, and organisations. AWS enables users to access computing resources, storage options, databases, machine learning capabilities, and more over the internet. These resources are hosted in data centres distributed globally, allowing for high availability, reliability, and scalability.

Cloud-Native refers to an approach to building and running applications that fully leverage the advantages of cloud computing. Key principles of cloud-native applications include:

- Microservices Architecture:
  Applications are broken down into smaller, loosely coupled services that can be developed, deployed, and scaled independently.

- Containerization:
  Applications and their dependencies are packaged into lightweight, portable containers for consistent deployment across different environments.
- Orchestration:
  Tools like Kubernetes are used to automate the deployment, scaling, and management of containerized applications.
- Continuous Integration/Continuous Deployment (CI/CD):
  Emphasis on automation and frequent releases to accelerate software development and updates.
- Resilience and Scalability:
  Applications are designed to be resilient to failures and scalable to handle varying workloads.

Cloud-native applications, often referred to as "cloud-native workloads," are designed and optimised for cloud environments like AWS, embracing the flexibility, agility, and scalability offered by the cloud to deliver efficient and innovative solutions.

2) System specification

2.1) System components

Originally the system was deployed to an x86 Virtual Private Server, having one vCPU, 1 GB RAM and 20 GB storage. For the database, I implemented a lightweight database abstraction, which persisted into a JSON file after each write. The API was an Express[14] web service, behind an Apache reverse proxy. The CLI was compiled on the server, the dependencies were statically linked. As an extension, the original code included the Tesseract library from Google for Optical Character Recognition operations. This feature was removed in the reworked solution, for numerous reasons, see section 2.5).

2.1.1) System architecture of the prototype

See below the composition of the system components in the original context:

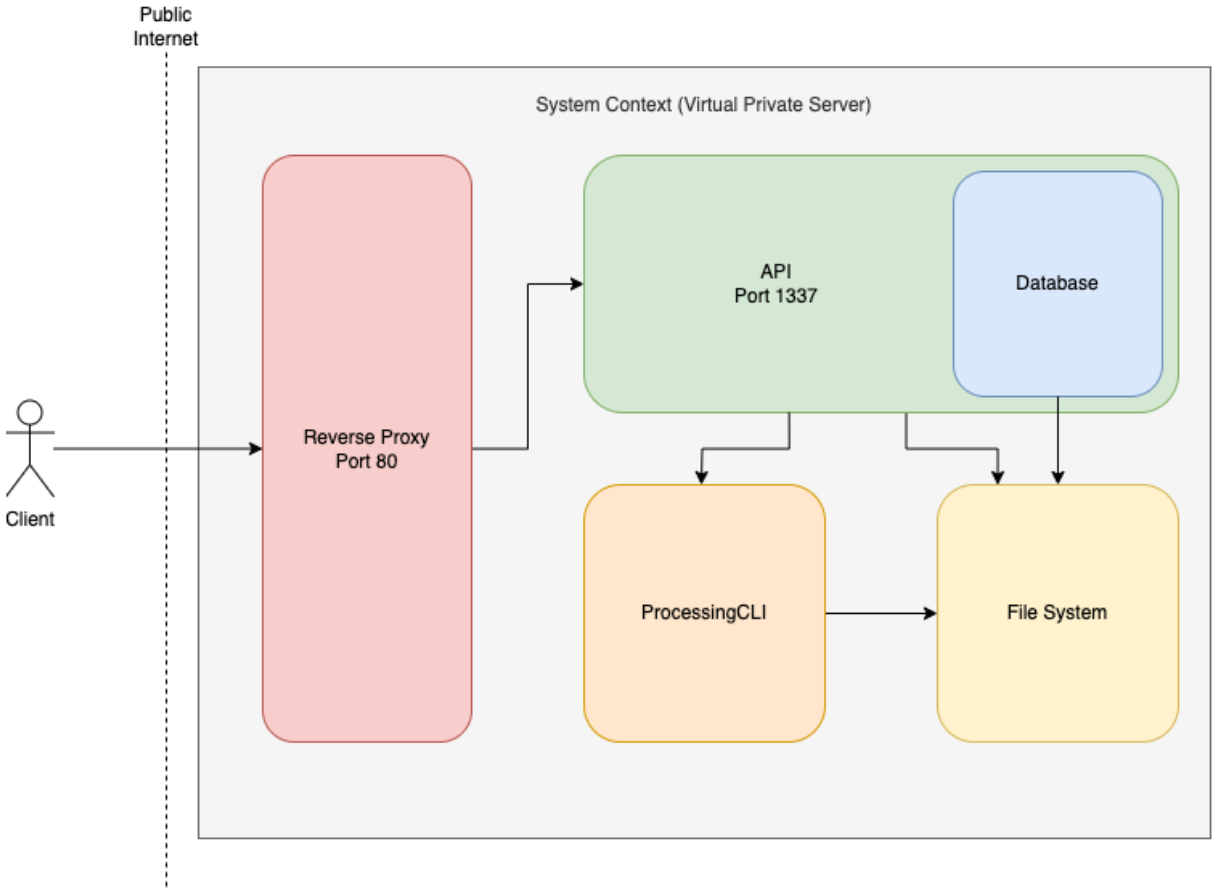

Figure 2.: System architecture of the prototype

2.1.2) REST API

The role of the REST API was to serve data to the client and dispatch computation operations. It contained the Database as well an abstraction to launch the ProcessingCLI as a child process, and parse and process its standard output. The endpoints were:

Method	Endpoint	Purpose
GET	/books	Get all stored books
GET	/books/:id	Get a book by :id
POST	/books	Store a book from the request body
PUT	/books/:id	Update a book by :id from the request body
POST	/books/recognize	Dispatch a Recognition Operation from the image passed in the request body

2.1.3) Database and cache

The Database's purpose was doing Euclidean distance[12] calculations between query vectors and the stored vectors. This happened during every Recognition Operation, right after the extraction of the book cover from its context, between the unwarped cover's query vector and the stored query vectors. For practical reasons I did not use any packaged database engine (we are 5 years before the introduction of Qdrant[15]), because I combined vector search with full-text search to improve the results. For the full-text search I implemented a variant of tf-idf[14] ranking.

2.1.4) Object storage

Apart from metadata such as title, author, release year, tags and the 18 dimensional CIELAB vectors (see section 2.5)), all covers were processed on writes, and their SURF descriptors were stored on the file system in XML format. Typically one book cover consumed 3 MB of storage and was saved under a name derived from the book's Identifier (ID).

2.1.5) A native (C++) CLI as asynchronous worker

The CLI handled the following operations:

Command	Arguments	Returns
preprocess	Image path	18 dimensional CIELAB vector from the extracted cover, Path of the extracted cover in /tmp folder
store	Image path, New book ID	18 dimensional CIELAB vector from the image
recognize	Image path, Candidate IDs, Confidence threshold	Matching ID if the confidence was above the threshold

2.2) Cloud components, the building blocks

AWS has many products. When I designed the system's Cloud Architecture I focused on coming up with the simplest yet flexible enough composite of well-engineered solutions, which could be single-handedly operated, flexible on security configurations, and easy to replicate, if I have to spread the system across multiple locations. In reality, the production system uses 19 components, of which I listed the most important ones below.

2.2.1) Virtual Private Cloud, Route53 and Certificate Manager

VPC is a logically isolated section of the AWS Cloud where you can launch AWS resources in a virtual network that you define. It allows you to have complete control over your network configuration, including selecting your IP address range, subnets, route tables, and network gateways.

Amazon Route 53 is a scalable Domain Name System (DNS) web service designed to provide highly reliable and scalable domain registration, DNS routing, and health checking of resources within your infrastructure.

AWS Certificate Manager (ACM) is a service that lets you easily provision, manage, and deploy public and private Secure Sockets Layer/Transport Layer Security (SSL/TLS) certificates for use with AWS services and your internal connected resources.

When using these services together, you can create a secure and well-organised infrastructure. For instance, you might configure your applications running in a VPC, manage their DNS with Route 53, and secure them with SSL/TLS certificates provisioned and managed by ACM.

2.2.2) Elastic Container Service and AWS Fargate

Amazon Elastic Container Service (ECS) is a fully managed container orchestration service. ECS simplifies the deployment, management, and scaling of containerized applications using Docker containers. It's designed to work seamlessly with other AWS services to enable the building of highly available, scalable, and resilient applications.

AWS Fargate is a serverless compute engine for containers that allows you to run containers without managing the underlying EC2 instances. ECS seamlessly integrates with Fargate to provide a serverless container management experience.

By using Amazon ECS, you can deploy and manage containerized applications with ease, taking advantage of features like scalability, availability, and integration with other AWS services for a comprehensive container orchestration solution.

2.2.3) Aurora and ElastiCache

Amazon Aurora is a high-performance, fully managed relational database engine that's compatible with MySQL and PostgreSQL. It's designed to offer the performance and availability of commercial databases at a fraction of the cost.

Amazon ElastiCache is a managed in-memory caching service that makes it easy to deploy and operate in-memory data stores or caches in the cloud. It helps improve application performance by reducing the load on your databases.

In summary, Aurora is used for reliable and high-performance database storage, while ElastiCache is used for caching frequently accessed data in memory to improve application performance.

2.2.4) Simple Storage Service and Elastic File System

Simple Storage Service or Amazon S3 is a highly scalable, durable, and secure object storage service designed to store and retrieve any amount of data from anywhere on the web. It is commonly used for static web hosting, backup and restore, data archiving, and as a data lake for analytics.

Elastic File System or Amazon EFS is a fully managed, scalable, and elastic file storage service designed to provide scalable file storage to multiple EC2 instances. It's ideal for applications that require shared file storage and a traditional file system interface.

In summary, S3 is ideal for scalable object storage and archival, while EFS is suitable for scalable and shared file storage for applications that require shared access to a file system.

2.2.5) AWS Lambda

Amazon Web Services (AWS) Lambda is a serverless computing service provided by AWS that lets you run your code without provisioning or managing servers. It's an event-driven, serverless computing platform that automatically scales and manages the infrastructure needed to run your applications. Key features and aspects of AWS Lambda include:

- Serverless Computing:
  AWS Lambda allows you to run code in response to events without provisioning or managing servers. It abstracts the infrastructure and allows you to focus on writing and deploying code.
- Event-Driven Execution:
  Functions in AWS Lambda are triggered by events such as changes to data in an Amazon S3 bucket, updates to a DynamoDB table, HTTP requests via API Gateway, and more. Events are the sources of input to your Lambda functions.
- Pay-Per-Use Pricing Model:
  AWS Lambda follows a pay-as-you-go pricing model, where you are billed based on the number of requests for your functions and the duration of their execution.
- Automatic Scaling:
  AWS Lambda automatically scales your applications in response to incoming traffic. It can handle a few requests per day to thousands of requests per second.
- Integrated with AWS Services:
  AWS Lambda can be easily integrated with other AWS services, allowing you to create powerful and flexible applications. Common integrations include Amazon S3, DynamoDB, Amazon Kinesis, Amazon SQS, API Gateway, and more.
- Programming Languages:
  AWS Lambda supports multiple programming languages, including Node.js, Python, Java, Go, Ruby, .NET Core, and custom runtimes through the AWS Lambda Runtime API.

- Stateless Execution:
  Each Lambda function is designed to be stateless, meaning that it should not rely on the state from previous invocations. However, you can use other AWS services to manage state if needed.
- Versioning and Aliases:
  AWS Lambda allows you to publish multiple versions of your function and use aliases to route traffic between different versions. This facilitates smooth deployments and rollback capabilities.
- Error Handling and Monitoring:
  AWS Lambda provides features for error handling, logging, and monitoring using AWS CloudWatch, making it easy to track function invocations, duration, and error rates.
- Security and Permissions:
  Lambda functions can be configured with fine-grained access controls using AWS Identity and Access Management (IAM) to ensure secure execution and interaction with other AWS resources.

AWS Lambda is widely used for a variety of use cases, including web applications, microservices, data processing, real-time file processing, chatbots, IoT applications, and more, where event-driven, serverless, and scalable execution is required.

2.3) Simplified network topology and access configuration

In Amazon Web Services (AWS), the security architecture is built using several key building blocks to ensure a robust and comprehensive security posture. The building blocks I used include Identity and Access Management (IAM), VPC, Security Groups, Bucket Policies and Access Control Lists.

Security Groups are useful for controlling network IP range- and port accesses within the VPC, from direction ranges to destination ranges. Think of it like an inter-component firewall configuration, typically attached to IaaS resources, such as EC2 and Fargate.

Bucket Policies are useful for granulary configuring S3 access rights to buckets and folders. Some services should be able to read and write but not delete, some only write to a destination (think of intermittent files).

The Access Control List configuration associated with IAM users controls which user can access which service, typically their AWS API through one of the official AWS SDKs. The configuration of the system as follows:

IAM user	ElastiCache	Aurora	S3	Lambda	EFS
Developer			X	X	
Fargate runner	X	X	X	X	
Lambda runner			X		X

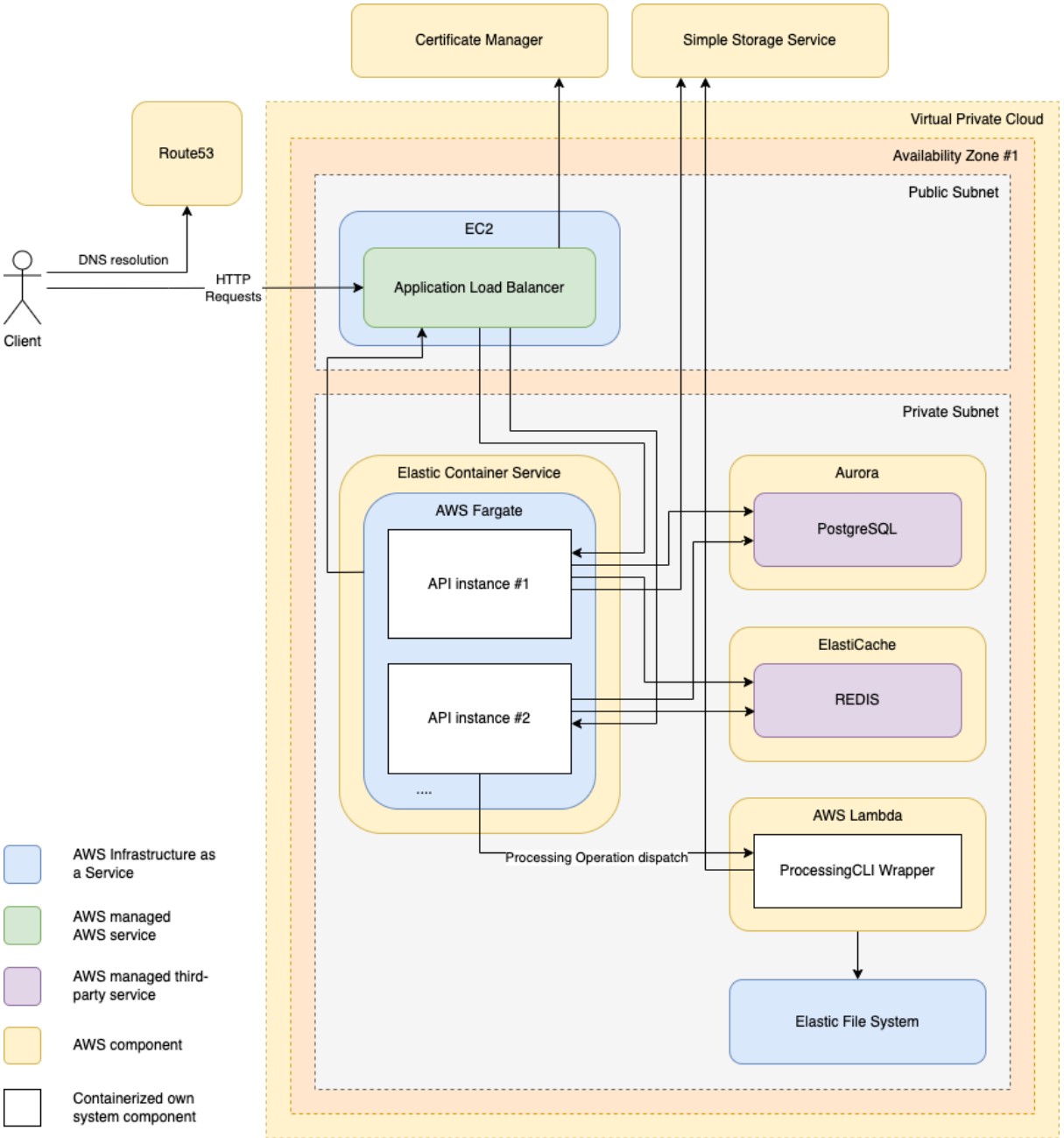

Figure 3.: Network topology on AWS

2.4) Continuous- Integration and Delivery from developer's angle

Continuous Integration and Continuous Delivery are pivotal to Agile software development. It ensures short development to production cycles, while promoting automated builds and deployments of software components. CI systems typically contain a scheduler and a task runner component, builds consist of dependency installation, code compilation, test execution, integrity checks, nowadays static and dynamic code scans, image generation, publishing the image to an object storage and promoting it for release. CD systems are very similar from an architectural standpoint, these systems are

responsible to deploy images from object storages to bare metal or Virtual Private Servers, configuring network visibility within the cluster and doing post-deployment checks such as liveness and readiness probes. Kubernetes, the most popular orchestration engine, provides a comprehensive solution for all of these tasks, and AWS EKS on top of Elastic Container Service is a viable alternative to organisations that want to leverage the advantages of managed infrastructure.

2.4.1) Containerization and Elastic Container Registry

In my system I used Docker and the Elastic Container Registry, AWS's private object storage for images. Unlike Virtual Machines, containers do not have hardware level virtualization and an underlying hypervisor, or reserved computing resources. They technically wrap the host's operating systems by the docker engine. This means, Docker images are easy to distribute and its resource consumption can change dynamically as long as the host machine is capable of providing that.

2.4.2) Version control in Gitlab and usage of GitlabCI

To store the codes I used Gitlab's managed private code repositories. Gitlab comes with a Dockerized Continuous Integration solution, which uses a Domain Specific Language in .yml format to let developers describe tasks, commands, environments by specifying base images. It is supporting environment variables, from repo or group level and triggers on branch content changes and events.

2.5) Reimplementation and changes for Version 1.0, briefly

First task I had to solve was to reimplement the API by following the old code as guidance, but using a more recent version of NodeJS (v16). While Express technically is just a router (like PAT in Go-lang), I wrote my own MVC abstraction on top of it, using singletons and dependency injection.

The API got an authentication endpoint using Application ID and Application Secret pairs, returning a Bearer token. Each application credential pair has associated permissions such as read, write and recognize, and the Bearer token is then used to authorise the calls.

The schema of the new API got a more generalised `/object` prefix. The metadata is now a free-form JSON object, and the API creates thumbnails from object images, which are then uploaded to S3.

The database has changed to PostgreSQL 13.3, and I reimplemented the Euclidean distance algorithm as a stored procedure. I also added REDIS to the stack, now used only for storing sessions, encrypted as JSON web token.

I managed to revive the ProcessingCLI to use newer dependency versions, built in a multi-stage, multi-repo pipeline, separately from the compilation step. Since GitlabCI and Lambda both use x86 architecture, with a lightweight, Debian based Docker base image I managed to create a compiling CLI.

An issue I bumped into early on is that while VMs have dedicated memory, a so-called random memory allocation strategy, built into many libraries, in order to provide better performance for the programs has an issue in Docker. The Docker engine allocates memory as a blob for each process, and when programs trying to address memory by the random allocation strategy, it immediatelly crash the Docker process. The issue arose when I tried to incorporate the latest Tesseract library, which was

heavily built on this strategy, probably targeting VMs or embedded system, so I decided to remove it from the final code.

The CLI dependencies are now built one by one, and the compilation now uses dynamic linking strategy. That means the final image contains all the dependencies, and with a Yolo-based object labelling extension the final image is about 1.6 GB, and the build takes about 55 minutes to complete, from source to image. For the Lambda runtime I introduced a small NodeJS wrapper, which technically dispatches commands by launching the CLI as a child process.

Both the API and the ProcessingCLI images are stored on Elastic Container Registry, and pulled from it during deployment.

3) Deploying the system to AWS Public Cloud

Before jumping into some further details, let me outline the sizing of the environments. By experimenting with a relatively large dataset of 15.000+ book covers, consuming about 60 GB of disk on Elastic File System, I empirically came to the conclusion that the optimal resource allocation for the API is about 0.5 vCPU and 1 GB of memory. While the API does not perform any computation-intensive task, this amount of memory ensures, that about 100 processing command can be dispatched from a single instance in Fargate.

Experiments showed, that in case of the Lambda image, the platform allocates CPU resource linked to RAM on a 1:2 ratio, and while having 1 core with 2 GB memory was the most optimal configuration for the single threaded CLI, AWS billed double the amount of the used computing time. The Yolo extension, having 80 distinct object classes in a binary neural net descriptor can operate just below 1 GB of RAM, while my model is consuming about 400 MB on a batch size of 42 classes. Because of this, the current configuration is 0.5 cores and 1 GB RAM, which implies about 1.5x performance degradation for half the cost. In the future, many of the current bottlenecks will be cleansed out, by reworking the entire preprocessing pipeline.

3.1) Security considerations and vulnerability remediation strategy

As I wrote in section 2.3) I applied many security practices during the configuration and cloud setup, like least-priviliged principle. To add to the already mentioned list, I built in many control points for advanced logging and monitoring, error-supressing exception handling and strict network-isolation configuration.

For development purposes I am using Docker Compose, a Docker extension to simulate multi-component environments locally. While the API is able to connect to a provisioned Lambda instance through the AWS JavaScript SDK, I found it would be not a wise decision letting local instances of the ProcessingCLI Wrapper connect to the production EFS storage. EFS storage works like a networked USB storage, and unfortunately there is no option to restrict clients mistakenly erase data from it. Because of this this, the Wrapper uses S3 to mirror the EFS descriptor storage on each command, and if EFS is not mounted to the Docker process, meaning its not running in the AWS Lambda platform, it downloads the necessary files from the S3 mirror before proceeding with the operation. The good thing, is that through the AWS JavaScript SDK, this integration was also relatively easy, and I could use a Dev IAM token to authenticate my local instance.

Further strengthening the system: I am using a set of salts to encrypt sensitive data in the system. Practical example is the already mentioned session information storing in ElastiCache or the hashing of Application Secrets. I built in an application-specific rate limiting solution using ElastiCache again, and a strict request origin check, configurable on application level. Sessions have an expiration of 12 hours, then a reauthentication is required from the client. The salts are rotated quarterly.

For the CI builds I use restricted Gitlab accounts and 2048-bit RSA keys for moving code from one repo to another. I also use separate IAM users for publishing images to or pulling images from the Elastic Container Registry. I erase old, vulnerable or somewhat broken images from the registry frequently to keep the platform safe from operator errors.

About vulnerabilities, strategies I apply:

- Always using an actively updated and patched Docker base image (like node:16)
- Dependency versions are hardcoded into the package.json to avoid unwanted vulnerabilities to be introduced by unscanned dependency patch releases
- I use the `npm audit` feature frequently and execute it during build to actively monitor known vulnerabilities, and if found, I am remediating them by hand, while checking the integrity and compatibility of newer dependency versions to the existing code, in a timely manner
- The builds use the `npm ci --only=production` command to omit development packages

There are plenty more strategies, like secrets detection, static and dynamic code scans, many which are already offered by Gitlab, for enterprise accounts. Maybe at some time in the future I will introduce these to the builds.

3.2) Zero-time deployment, load balancing and autoscaling in ECS

Elastic Container Service applications using AWS Fargate target by default are deployed as standalone services, with a restriction that these could be only provisioned to a public subnet, in order to have a public IP address allocated for them. However by provisioning an Application Load Balancer, the customer can leverage Route53's hosted zones and custom subdomain configuration options along with issued SSL/TLS certificates by the Certificate Manager. Upon expiration, the Certificate Manager automatically reissues these certificates and updates the load balancer's configuration.

When we talk about Zero-time deployment, we mean the notion of automatically registering and deregistering services in the load balancer's routing table upon the service's readiness or shut down. For this, each Fargate service needs to implement a liveliness and readiness probe, and the platform itself automates the configuration change in the load balancer after a successful service provisioning operation.

Autoscaling is a feature of Fargate. It is specified by scale-up and scale-down policies, where customers can specify which metric and what target level they want to have the platform adjust to the demand from clients. My system has 2 instances by default, which can scale up to 16 instances. I tested it when imported thousands of books from a NodeJS script having 8-fold parallelism, within minutes the platform scaled up to 12 instances. This feature can be extremely useful in data-intensive applications and in cases, when rapid change in traffic can occur.

3.3) Parallelizing asynchronous worker calls on Lambda

By default, Lambda comes with a reserved parallelism of 10 instances. Cold start happens, when the Lambda platform does not have a running instance of a function, and the platform needs to provision a new instance to handle the client's event. This causes, depending on the runtime and image size, few hundred milliseconds of delay, before running the function and responding to the event. Lambda platform keeps these functions alive for a non-public, probably minutes period, meaning repeating or subsequently occurring events can be served much quicker. And if a function is busy executing, a new function gets provisioned.

In my case, when we talk about thousands, tens of thousands of distinct objects, a query can require hundreds or more concurrent functions to run at the same time. Lambda platform allowes customers to reserve parallelism, from 10 up until 1000 instances. Due to the platform's underlying infrastructure calling a function 100 times concurrently, even if these functions are cold started, the response would be served by the time the slovest function finishes the execution. Ideal for parallelizable workloads such as stateless vector comparisons.

3.4) The dilemma on deploying the components to multiple availability zones

Deploying components to multiple availability zones is a critical aspect of ensuring high availability, fault tolerance, and resilience in cloud computing. An availability zone is a distinct physical location within a region that is designed to be isolated from failures in other availability zones, providing redundancy and minimizing downtime. However, deploying to multiple availability zones presents a dilemma in terms of cost, complexity, and trade-offs.

Deploying components across multiple availability zones improves availability by reducing the impact of a single zone failure. However, this redundancy can significantly increase operational costs due to the need for additional resources and data replication. It also introduces complexity in managing and synchronizing data across zones, and can lead to challenges in application design, monitoring, and troubleshooting.

Allocating resources across multiple availability zones requires careful load balancing to ensure optimal performance. Striking the right balance in resource allocation and load distribution is a dilemma, as improper allocation can lead to underutilization or overutilization.

In my case, the cost was an extremely important aspect when I deployed the system to Cloud-Native. While Cloud-Native is considered the best and most versatile solution for leveraging all the benefits of managed infrastructure, its costs are still way above the basic approach, for example, a large EC2 instance containing all the components. You will see in a latter section the financial implications of hosting the platform using the tools I already described, after the initial release and after the upload of a typical customer dataset, hosted in a single availability zone.

4) Evaluating the results

In this section I am outlining some key aspects I took into account when I evaluated my solution. You will see that hosting on Cloud-Native has a clear advantage: the costs are calculated based on resource utilization and time spent on the computation-intensive operations, aligned with the inbound traffic of the API.

While matching feature vectors in large databases are very computation intensive, with the achieved parallelism, the API able to respond in a satisfactory time. I compared three datasets, two of them relatively small, were used to fine-tune the preprocessing parameters, and a large one, used to fine-tune the optimal batch size and achievable parallelism.

4.1) Financial implications

The chart below shows the average cost of the Version 1 configuration between 1st of January, 2023 and 31th of May, 2023.

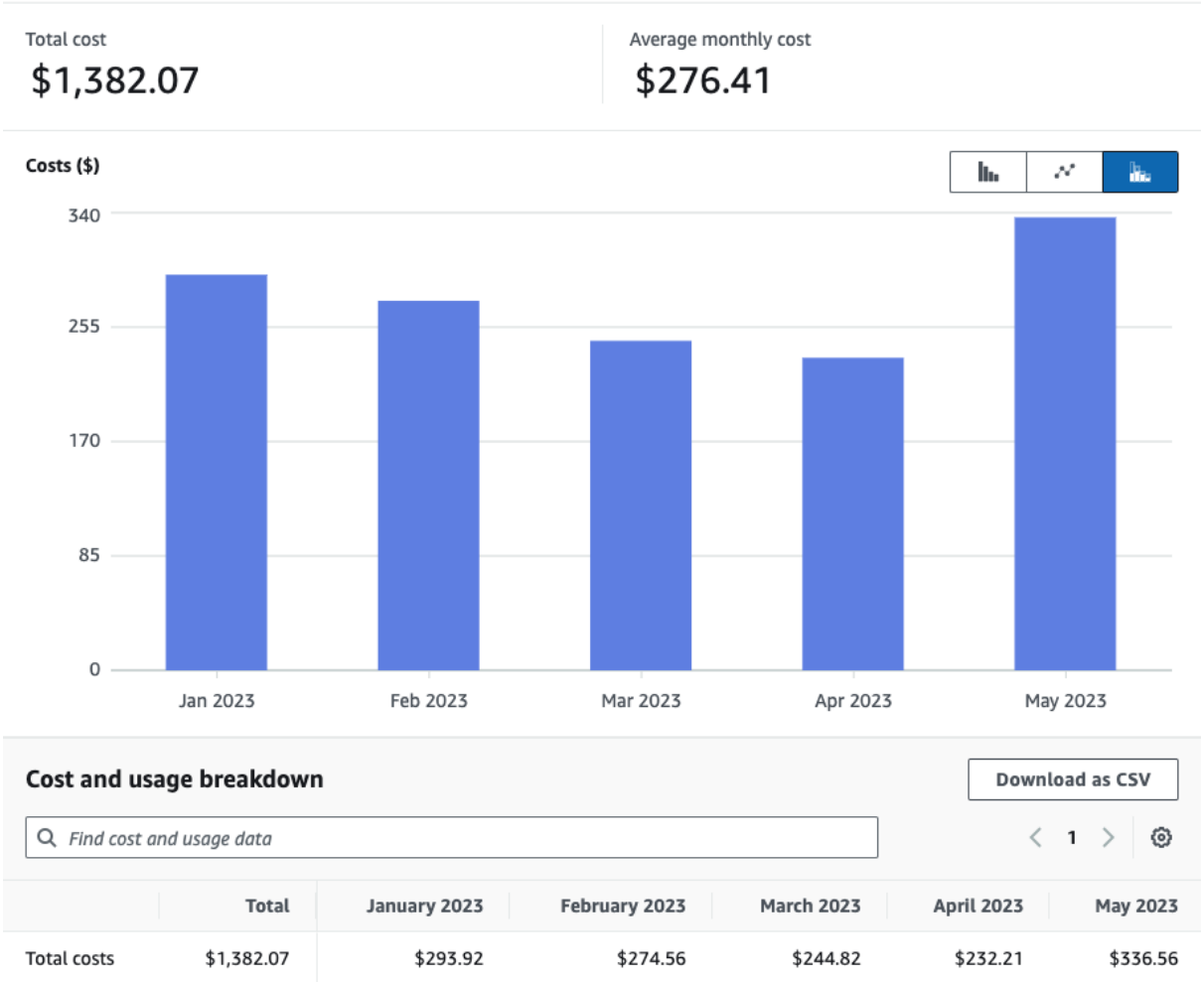

Figure 4.: Monthly costs between 1st of January, 2023 and 31th of May, 2023

Moving a Three(+) Tier app to Cloud-Native

As you can see, the cost decreased in the first 4 months of year as I was introducing more and more optimizations to the system. The jump of about $100 in May was after I migrated 15000 books into the system, consuming about 60 GB of space.

Figure 5.: Drilling down on May, 2023

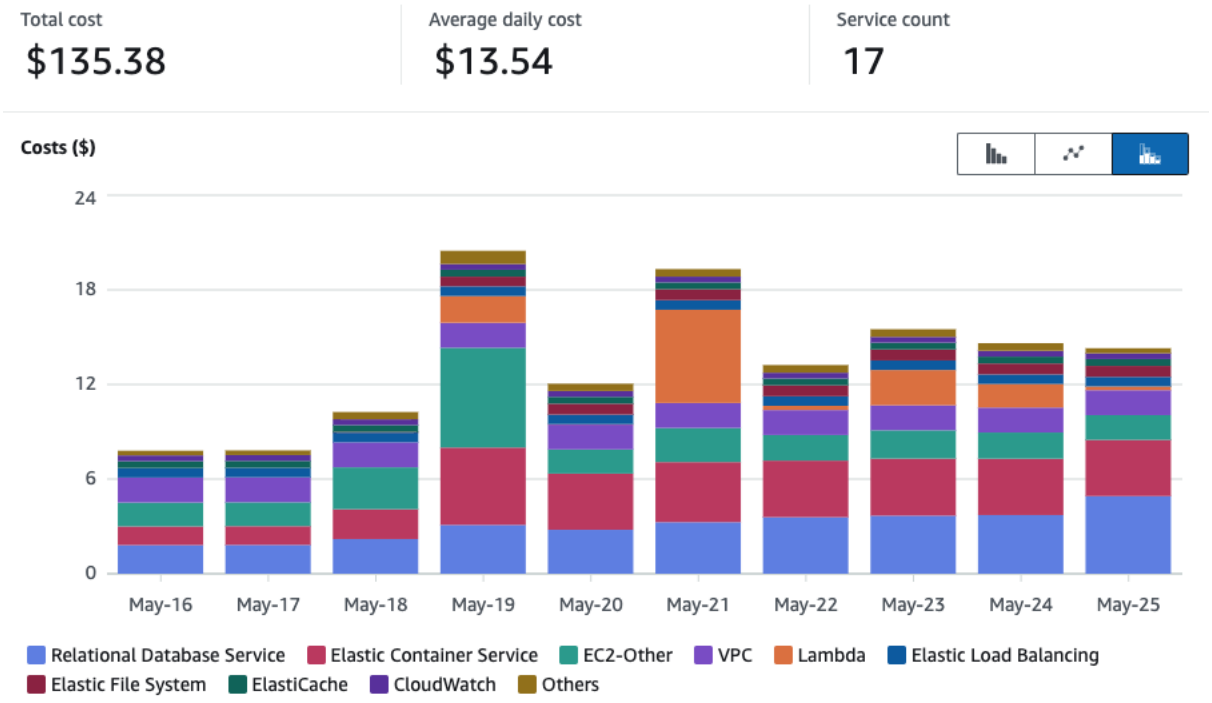

Figure 6.: Drilling down the period between 16th of May and 25th of May

Moving a Three(+) Tier app to Cloud-Native

To see, where the hosting's real cost arise, which are Lambda, EFS, Aurora and S3, we can make a pricing model estimation:

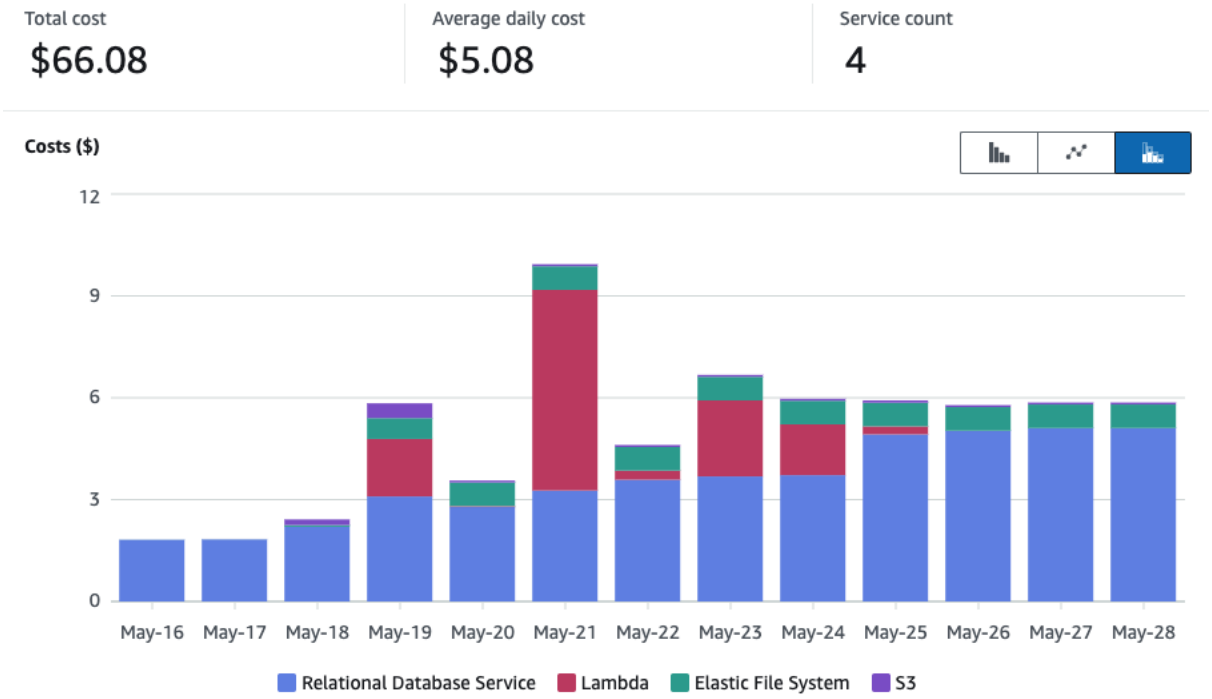

Figure 7.: Extended range until the consolidation on the 28th of May

Component	Usage	Daily cost increase	Unit costs
AWS Lambda	15.000 requests	$6	$0.0004 / store OP
Elastic File System	60.000 MB storage	$0.4	$0.000006 per MB
Aurora	15000 records	$3	$0.0002 per record
S3	75000 thumbnails	insignificant	insignificant

Dimension	Calculation	Result
Cost to store an object (per request)	$0.0004 + 4 * $0.000006 + $0.0002	$0.000624 USD
Cost to store 15.000 objects (per day)	(4 * $0.000006 + $0.0002) * 15000	$3.36 USD

4.2) System response times on various datasets

The following experiments were performed agains three datasets. The first two were relatively small, were used to fine tune the preprocessing operation's parameters, the last one is the previously mentioned large dataset, used to fine tune the optimal batch size and parallelism. Without comments.

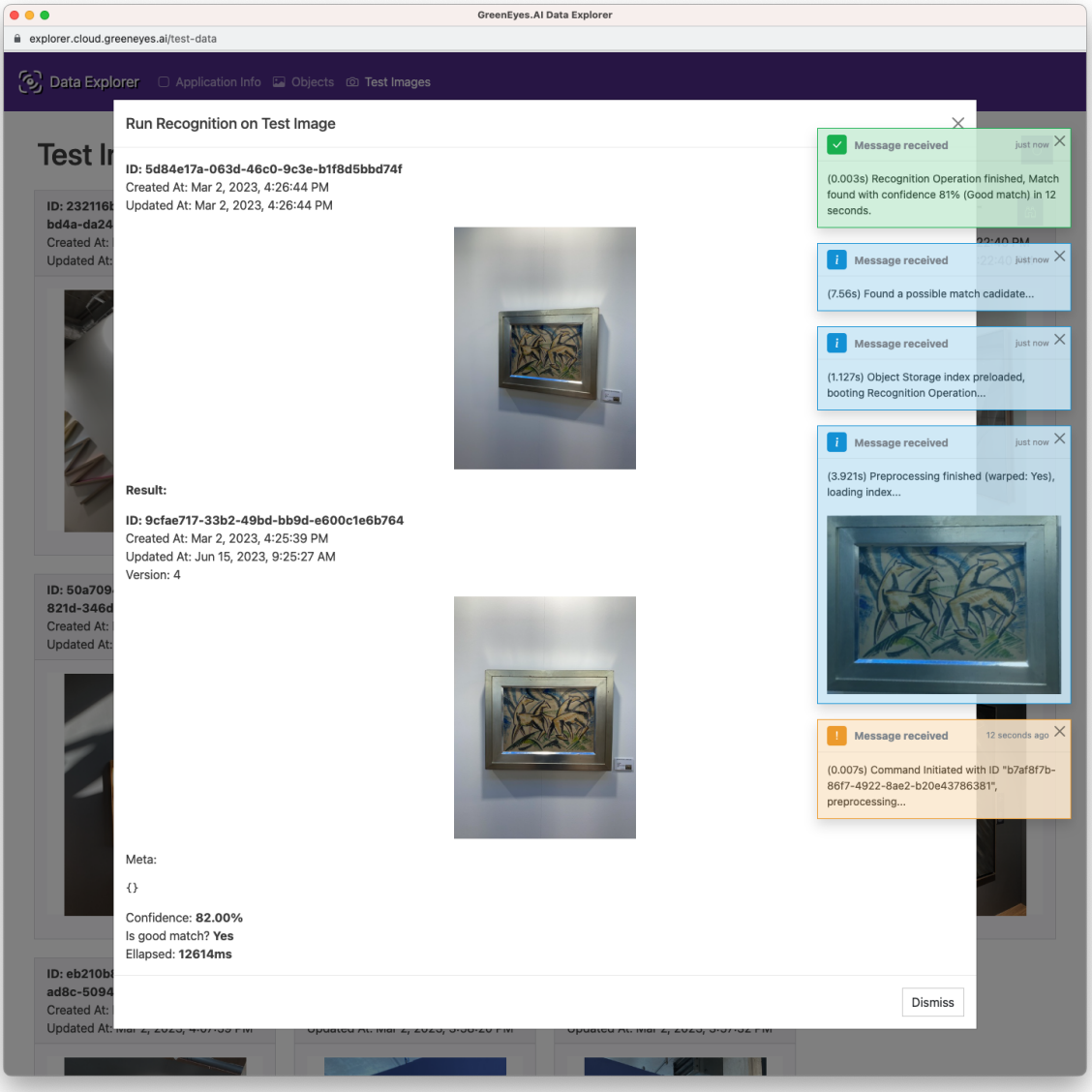

Figure 8.: Paintings from Art and Antique Budapest demo dataset result (21 objects)

Moving a Three(+) Tier app to Cloud-Native

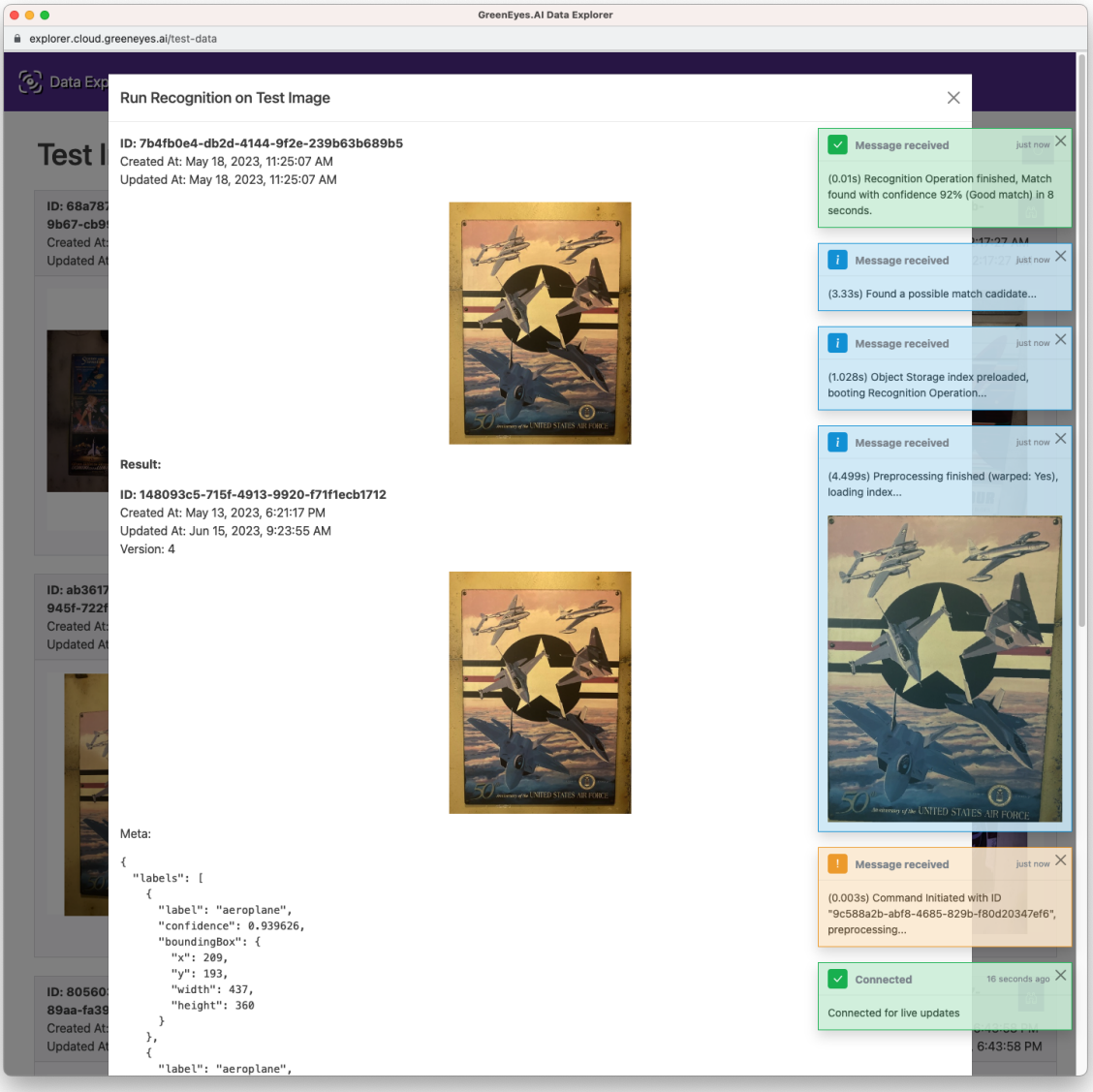

Figure 9.: Printed and mounted posters demo database result (52 objects)

Moving a Three(+) Tier app to Cloud-Native

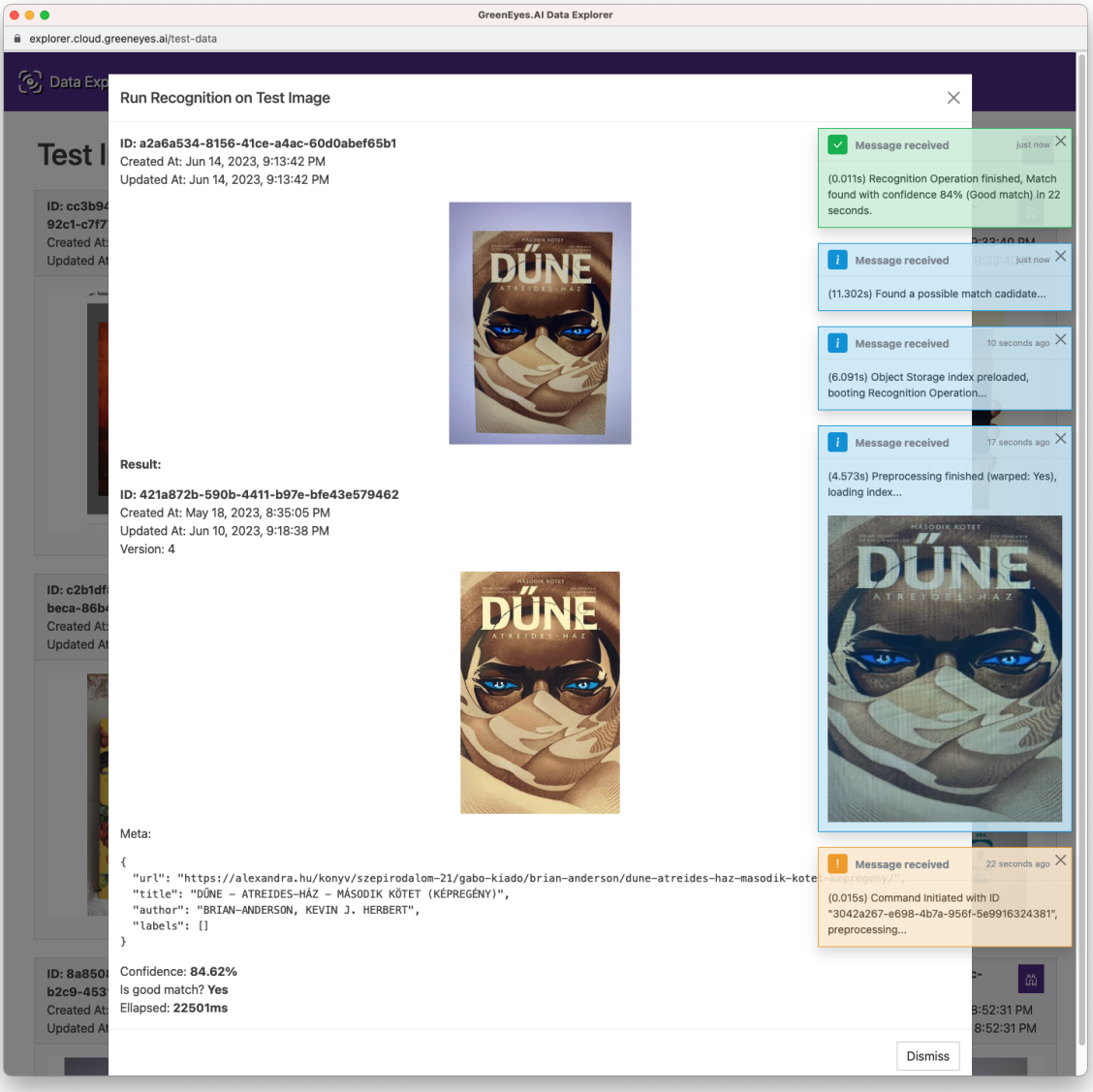

Figure 10.: Book Store Showcase with 15k+ Objects demo dataset (15811 objects)

5) Summary and closing

From the prototype to Version 1 was a long journey. 5 months, endless nights and weekends, researching, struggling with AWS configurations and the C++ CLI build pipeline, but it has worthed. In the meantime, we built a complete Web Suite and a TypeScript toolkit to facilitate API integration efforts of our testers, both from the browser and server side.

Hope you find this paper comprehensive and entertaining. I enjoyed every hour of writing it, since this project is my own mind's manifestation, and me and my team putting a lot of effort into turning the business into a prosperous and prestigious global enterprise.

If you would like to learn more about the project, visit our website at https://www.greeneyes.ai.

6) References

[1] What is three-tier architecture? (https://www.ibm.com/topics/three-tier-architecture), last visit: 12th of October, 2023

[2] Á. Kiss, "Modern Book Catalog", Bachelor thesis, Óbuda University John von Neumann Faculty of Informatics, 2014 (https://www.arpi.im/thesis.pdf), last visit: 12th of October, 2023

[3] CIELAB color space (https://en.wikipedia.org/wiki/CIELAB_color_space), last visit: 12th of October, 2023

[4] Speeded up robust features (https://en.wikipedia.org/wiki/Speeded_up_robust_features), last visit: 12th of October, 2023

[5] k-nearest neighbors algorithm (https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm), last visit: 12th of October, 2023

[7] Boost library (https://www.boost.org/), last visit: 12th of October, 2023

[8] OpenCV (https://opencv.org/), last visit: 12th of October, 2023

[9] EU AI Act: first regulation on artificial intelligence (https://www.europarl.europa.eu/news/en/headlines/society/20230601STO93804/eu-ai-act-first-regulation-on-artificial-intelligence), last visit: 14th of October, 2023

[10] Green Programming: Reducing your carbon emissions when coding (https://datascience.aero/green-programming-reducing-your-carbon-emissions-when-coding/), last visit: 12th of October, 2023

[11] AlexNet (https://en.wikipedia.org/wiki/AlexNet), last visit: 12th of October, 2023

[12] Euclidean distance (https://en.wikipedia.org/wiki/Euclidean_distance), last visit: 14th of October, 2023

[13] Amazon Web Services (https://aws.amazon.com), last visit: 14th of October

[14] Express (https://npmjs.com/package/express), last visit: 14th of October

[15] Qdrant (https://qdrant.tech/), last visit: 14th of October

[16] tf-idf (https://en.wikipedia.org/wiki/Tf%E2%80%93idf), last visit: 14th of October