Know-your-student (KYS)

https://verify.college

Vincent Olesen Pierre Jézégou Arturo Lidueña
Vaclav Bily Christian Biffi Elliot Verstraelen

May 2024

Contents

Chapter 1

Introduction

Know-your-customer (KYC) is the process by which enterprises and institutions, such as banks and insurance companies, verify the identities of their customers. Enterprise KYC processes often involve verifying a piece of picture ID, such as a driver's license or passport, along with a face liveness check (which extracts a picture of the person being verified). During this process, they confirm that the selfie matches the picture on the ID and that the customer data matches the data on the ID.

In this project, we are addressing the issue of verifying a student's academic status on behalf of businesses. This is particularly useful for webshops or SaaS products that want to offer academic discounts. Specifically, we propose and implement a *student verification service* in the form of a software-as-a-service (SaaS) that validates student status. This service is similar to the KYC processes in enterprises, with the same steps but applied to a student ID, like a student card issued by a university.

While several services, like SheerID, or companies offering student discounts, like Jetbrains and Github, implement student verification, this is traditionally done by verifying that the student possesses an email from an academic institution. We propose emulating the KYC flow as it motivates

- The modeling of a non-trivial workflow (asynchronous, as steps like the face liveness check do not follow a classic request-response pattern and should be auditable by requirement) with cloud technologies.
- The application of machine learning in face extraction, comparison, and text extraction.
- Clear minimum viable product (MVP), which is extensible, allows us to extend the project scope with decoupled features such as verifying an academic email or human-in-the-loop verification.

Furthermore, this project motivates several technical desiderata relevant to cloud computing, such as exercising serverless architectures, infrastructure-as-code (IaC), and security best practices, which we will elaborate on in the following Chapters.

We highly recommend viewing the following demonstrations of our project before reading the detailed report:

- Demonstration of successful verification on desktop
- Demonstration of failed verification on desktop
- Demonstration of successful verification on phone
- Demonstration of failed verification on phone

The code is available in the GitHub repository volesen/kys.

The rest of this report is organized as follows: Chapter 2 presents the non-technical flow to be implemented in student verification. Chapter 3 presents an initial technical solution. Chapter 4 deals with the project's limitations, such as the cloud environment used. Chapter 5 proposes a new technical solution that considers and mitigates these limitations, documents the implementation, and provides a technical report. Chapter 6 describes the project management of the project. In Chapter 7, we provide a discussion and conclusion of the project.

Chapter 2

Functionality and Requirements

In this section, we propose the functionality and requirements of the student verification service from a non-technical point of view. In the following sections, we will elaborate on the technical solutions and the scope of the functionality.

2.1 Verification flow

The main functionality of the project is the verification flow. From the customer's point of view, we propose the following flow:

1. The student fills out form data (e.g., name, email, university, and expiration) in the customer application.
2. The student is redirected to the student verification service.
3. The student uploads a picture of their student card.
4. The student performs a face-liveness check (and a selfie is extracted).
5. The student status is verified or denied, and the student is redirected to the customer application.

We need the flow to function on both mobile and desktop devices. As an additional goal, we want the flow to seamlessly continue from one device to another, for example, switching from a desktop to a mobile device to complete the face liveness check, highlighting the asynchronous nature of the workflow.

2.2 Audit panel and human-in-the-loop verification

We need to review the verification sessions and their status in addition to the main process. Also, we want to allow manual verification of failed sessions. For

example, student cards that do not mention the university name or mention it in an abbreviated form can be manually approved. These features can be implemented in various ways, such as through email notifications, SMS, Slack, or by creating a back office for the the SaaS in general. We defer the choice to the proposed technical solutions.

2.3 Authentication, authorization, and governance

Ignoring the aspect of a business model (who should be charged for the verification service), we should note one aspect of the project that significantly impacts the technical solution, particularly regarding data governance and security/authentication/authorization (cryptographically signed tokens and encryption).

- Should the customer start a verification session and allow the student to verify in the session with the customer's permission? That is, the customer owns the session and data.
- Should the student start a verification session, allowing the customer to verify the student's status with the permission of the user? That is, the student owns the session and data.

We will discuss and handle these questions in the proposed technical solutions, as we decided to handle this in the context of what was less complex (and more elegant) technically.

Chapter 3

Proposed Implementation

3.1 Desiderata

In designing our application, we adhere to the following principles that will form the foundation of our technical solution.

Serverless Our architecture will prioritize serverless solutions to ensure automatic scaling, reduced operational overhead, and cost efficiency. This approach allows the system to elastically handle varying workloads, providing high availability and fault tolerance.

Event-Driven and Asynchronous We will adopt an event-driven architecture. The API will be designed asynchronous by default, as steps in the verification flow are long-running or do not conform to a request-response pattern.

Infrastructure-as-Code (IaC) As a group, we chose to have a single production environment and use our own AWS accounts for local development. Consequently, we have a strong need for reducing prod/dev disparity and a consistent devleopment environment across group memebrs. We will implement Infrastructure-as-Code (IaC) practices to define and manage infrastructure through code. This ensures that our local development environment mirrors the production environment, promoting consistency, reproducibility, and efficient version control of infrastructure configurations.

3.2 Architecture

We initially propose the following technical flow for the mentioned pipeline. The flow is elaborated as a sequence diagram in fig. 3.1.

1. We extract text from the student card to verify against the form data

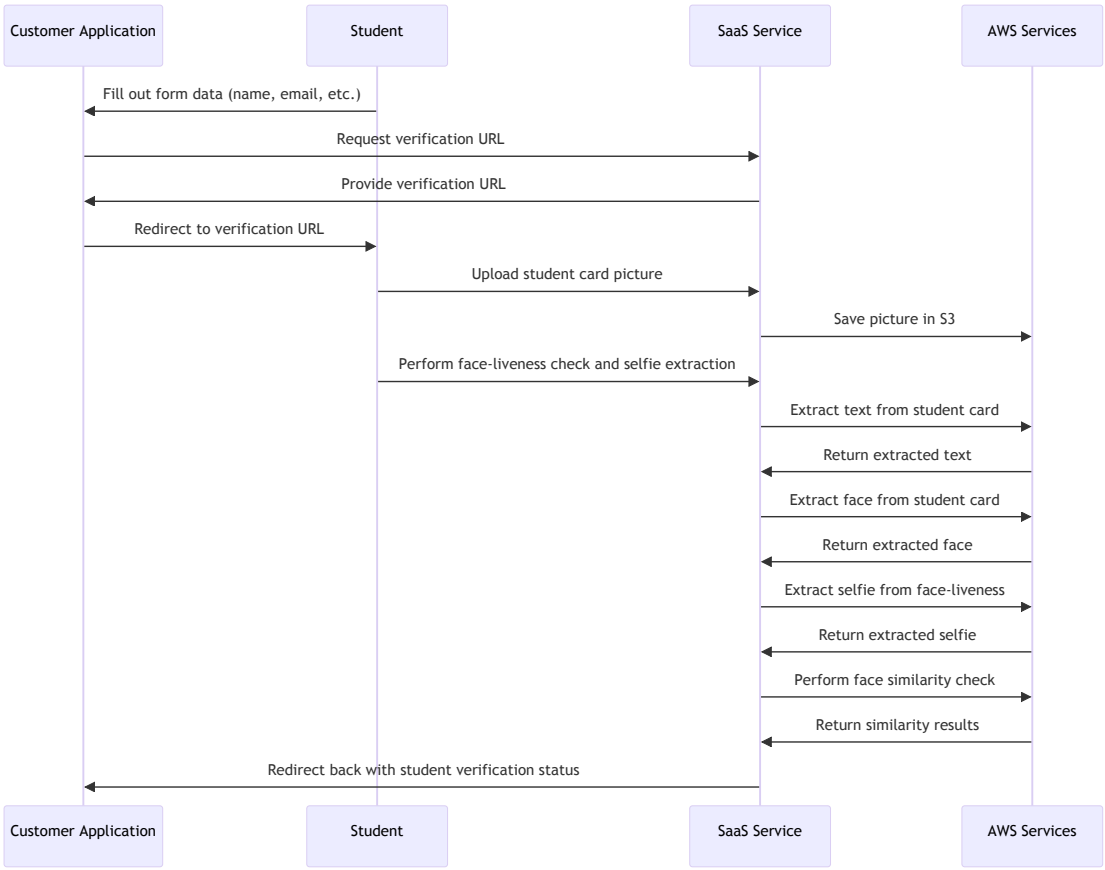


Figure 3.1: Sequence diagram of the pipeline for the minimum viable product (MVP).

2. We extract a picture of the student from the student card
3. We extract a selfie from the face-liveness check
4. We perform face similarity for the extracted faces.

As previously discussed, certain steps in the verification process are asynchronous, long-running, and do not conform to a traditional request-response pattern. To address this, we will model the verification process as a state machine, as illustrated in Figure 3.2. This approach enables callbacks from services, such as the face liveness service, to trigger state transitions, forming the foundation of an event-driven architecture.

Regarding authentication and data ownership, particularly in verifying student status, in this architecture, the company initiating the verification session will be designated as the data owner. The student will be redirected a verification session page, requested by the company. We will later discuss why this strategy was disadvantageous.

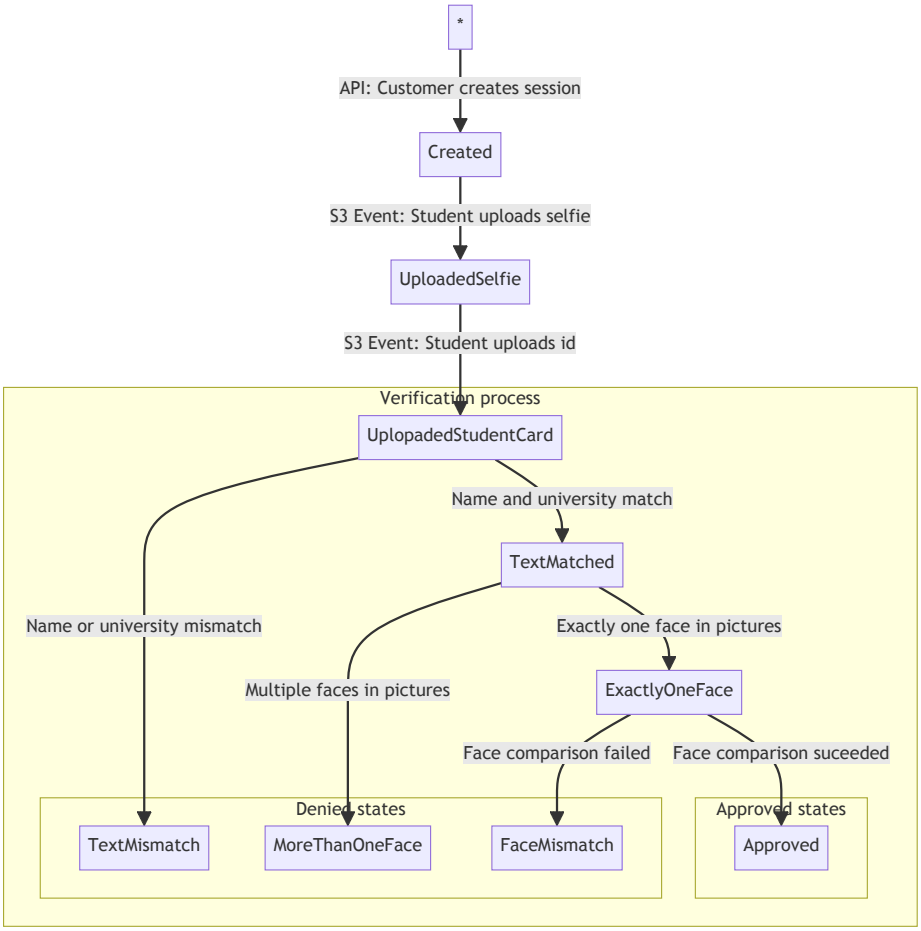


Figure 3.2: State machine representation of a verification session. Nodes represent the state and arrows represent state transitions.

3.3 Components and services

We will now describe the supporting components and services, which are shown in Figure 3.3.

As previously discussed, certain steps in the verification process are asynchronous, long-running, and do not conform to a traditional request-response pattern. To address this, we will model the verification process as a state machine, as illustrated in Figure 3.2. This approach enables callbacks from services, such as the face liveness service, to trigger state transitions, forming the foundation of an event-driven architecture.

Regarding authentication and data ownership, particularly in verifying student status, the student initiating the verification session will be designated as the data owner. The student will request a verification session and receive a secret token, which authorizes them for subsequent endpoints or API calls related to that session. This strategy is advantageous because it allows the SaaS platform

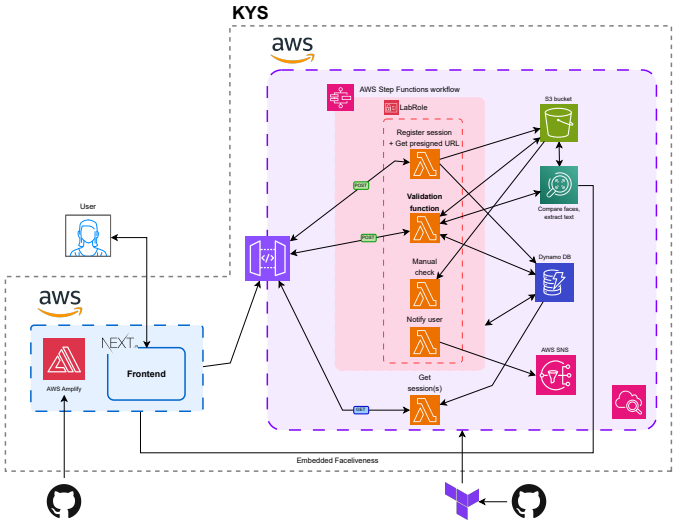


Figure 3.3: Components and services in the proposed architecture.

to issue the student a JSON Web Token (JWT) for the session.

The JWT, a cryptographically signed JSON object, will contain claims about the user's name, university, and student status. This simplifies integration from the perspective of the customer (such as a webshop or online business), as they only need to verify the signature and check the payload against, for example, checkout form data.

We will now describe the supporting components and services.

Amazon Rekognition Amazon Rekognition is a collection of computer vision and other machine learning services. It will be used to perform text extraction from student cards, face-liveness checks, and face-similarity comparisons.

Amazon S3 Amazon S3 is a scalable object storage service. Uploaded student cards and audit pictures extracted from the face-liveness checks will be stored in S3. It is more suitable than alternatives like Amazon Elastic File System (EFS) for storing individual objects such as images and documents.

Amazon DynamoDB Amazon DynamoDB is a fully managed NoSQL database service. Customer-supplied form data will be stored in DynamoDB. A non-relational database is adequate in this context and consequently allows us to exploit the performance and scalability of DynamoDB over, e.g., Amazon RDS.

Amazon Lambda and Step Functions Amazon Lambda is a serverless compute service, and Amazon Step Functions is a service for orchestrat-

ing workflows. Step Functions will be used to model the state machine, allowing state transitions based on events such as S3 uploads and webhook callbacks. Lambda functions will be used to perform tasks like text extraction or face comparison checks in response to these state changes. These services enable a serverless architecture that scales automatically and integrates seamlessly with other AWS services.

Amazon API Gateway Amazon API Gateway is a service for creating, maintaining, and securing APIs. Requests for the API will be routed through the API Gateway. It allows us to decouple traffic management from the underlying application and more easily allows setting up, e.g., support for HTTPS.

Amazon Amplify Amazon Amplify is a development platform for building web applications on top of `React` (for web). The frontend application for the pipeline will be developed and deployed using Amplify. Amplify offers seamless integration with, e.g., Amazon Cognito and frontend components for, e.g., the Amazon Rekognition Face Liveness check.

Amazon Cognito Amazon Cognito is an identity provider that supports OAuth. It will be used for authorization and authentication, particularly for back-office functionalities such as accessing audit pages. Cognito allows us to delegate the complexity of user sign-up, sign-in, and access control.

We should stress that **all the technologies above are serverless technologies**, allowing elasticity and scalability by default.

Furthermore, to version track source code, provision infrastructure, and deploy code (both locally and in production), we will utilize the following components, services, and technologies:

GitHub and GitHub Actions GitHub is a platform for version control and collaboration, and GitHub Actions is a CI/CD service. GitHub will be used for source code version control, and GitHub Actions will be used for CI/CD pipelines to lint code, run tests, and deploy the front end and back end.

Terraform Terraform is an infrastructure provisioning tool. It will define and provide the services mentioned above, ensuring consistent and repeatable infrastructure deployments.

Docker Docker is a platform for developing, shipping, and running containerized applications. Docker Compose will be used for local development to define and run multi-container Docker applications. This setup ensures that our development environment is consistent with the production environment, reducing the "it works on my machine" problem.

Regarding technology choices less relevant to cloud computing, we utilize the `React` framework for the front, written in `TypeScript`, a type-safe JavaScript extension.

3.4 Scope and Objectives

To delineate the scope of our project, we will focus on developing a minimum viable product (MVP) consisting of a frontend application interacting with a backend application to execute the verification flow described previously. This MVP will also include Terraform integration for infrastructure management.

In addition to the MVP, we have identified several features as stretch goals to enhance the system's functionality. These include implementing detailed audit pages for tracking and reviewing activities, adding authentication mechanisms to secure access to these audit pages, human-in-the-loop verification, and configuring a custom domain secured with HTTPS.

We consider the detection of fake student cards and extraction of expiry dates out of scope, as they are not required for the proof-of-concept.

3.5 Twelve-Factor Methodology

Adhering to the Twelve-Factor principles ensures portable, maintainable, and scalable applications. These principles promote a clean separation of concerns between code, configuration, dependencies, and backing services, making deployments on any cloud platform, such as AWS, more straightforward. This methodology supports applications' adaptability and scalability as requirements evolve [3].

This section outlines how our cloud application follows the Twelve-Factor methodology to enhance portability, maintainability, and scalability.

Codebase We use Git for version control, hosted on GitHub, with a monorepository structure for both the Python backend and Next.js frontend. This setup simplifies version control and ensures synchronization during deployments.

Dependencies Backend Dependencies are managed with pip and virtual environments (`venv`), listed in `requirements.txt` for easy installation, and `requirements-dev.txt` for development dependencies.

Frontend Dependencies are managed with `npm`, listed in `package.json`, with installation streamlined by `npm install`.

Configuration Configuration is managed via environment variables, with sensitive information stored outside the codebase. The backend uses Chalice `config.json` and environment variables, while the frontend uses `.env` files and GitHub secrets/variables, enhancing security and portability.

Backing Services We treat backing services as attached resources, exclusively using AWS-managed services like S3, DynamoDB, and API Gateway, ensuring a clear separation between application code and infrastructure.

Build, Release, Run Our application uses AWS Chalice for the backend, with `chalice local` for local development and `chalice deploy` for deployment. The frontend uses Vercel with Next.js, enabling deployment through version control pushes.

Processes We implement a serverless architecture with stateless Lambda functions, which retrieve state information from resources like DynamoDB, ensuring scalability and fault tolerance.

Port Binding Port binding is managed during local development with `chalice local`. In deployment, AWS Lambda functions handle port binding automatically, simplifying development and deployment processes.

Concurrency Our serverless architecture supports horizontal scaling on demand with AWS Lambda functions, eliminating the need for manual server management. Services like DynamoDB and S3, being serverless, enhance scalability and cost-efficiency.

Disposability AWS Lambda functions are ephemeral, spinning up and terminating on demand. This disposability ensures rapid scaling and fault tolerance, as new containers can handle subsequent requests without impacting overall functionality.

Dev/Prod Parity Development and production environments use the same AWS services and configurations, minimizing the risk of issues during deployment and ensuring smooth transitions from development to production.

Logs Logging is centralized using Chalice's middleware and AWS CloudWatch, with AWS X-Ray for tracing. Logs can be exported to third-party services like Grafana via AWS Firehose for advanced visualization and analysis.

Admin Processes Admin processes are managed with Python scripts or Lambda functions, with Infrastructure as Code (IaC) via Terraform, ensuring consistent and auditable infrastructure changes.

Chapter 4

Limitations and Improvements

This section outlines the limitations encountered during the project, the mitigations applied to address these limitations, and the improvements made to the proposed technical solutions during implementation.

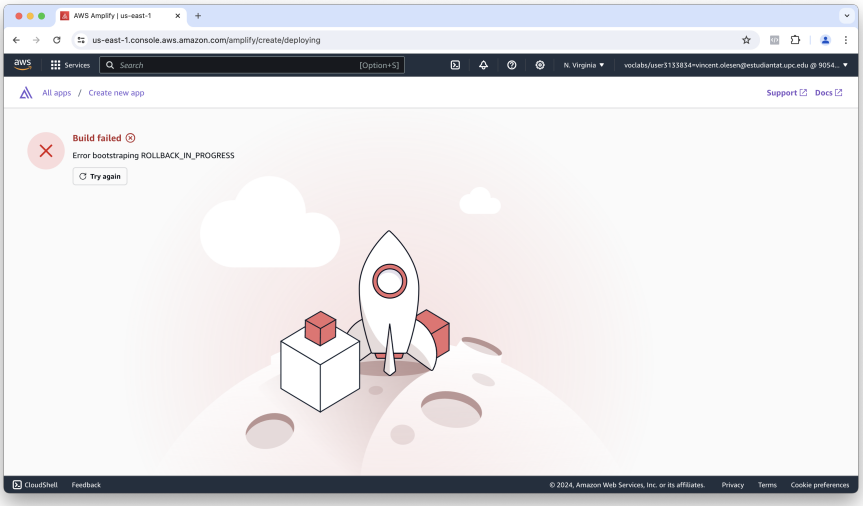


Figure 4.1: Amazon Amplify v2 failure during project setup.

4.1 Limitations and Mitigations

Firstly, the AWS Learner Lab environment presented several limitations:

- The creation of new IAM roles was prohibited. We were restricted to using the `LabRole` IAM role, which had limited permissions. This restriction led

to failures in setting up both Amplify and Step Functions, as illustrated in Figure 4.1.

- The `LabRole` IAM role did not permit programmatic access. Consequently, we had to rely on AWS credentials that rotated every four hours. These credentials needed to be extracted manually from the AWS Learner Lab console, complicating the deployment of a CI/CD pipeline for the backend without human intervention.

To overcome these limitations, we implemented the following strategies:

- Instead of using Amplify, we adopted Vercel, a platform-as-a-service (PaaS) for deploying frontend applications. Vercel abstracts away many complexities, such as CDN configuration, and provides seamless integrations with GitHub for CI/CD, including features like linting and testing on pull requests before merging into the main branch. This choice mitigated the limitations and provided additional functionality with reduced complexity. Vercel offers more capabilities than Amplify by simplifying several aspects that would otherwise require manual configuration. It is noteworthy that Amplify is designed as a counterpart to Google's Firebase and Vercel.
- Instead of using AWS Step Functions, we modeled the workflow ourselves using DynamoDB for session state management and Lambda functions to react to events. Although this approach sacrifices the built-in auditability of Step Functions, this could be mitigated by implementing event sourcing. However, we consider event sourcing out of the scope of this project.
- Instead of using Amazon Rekognition Face Liveness to confirm face liveness and extract a selfie, due to limited permissions, we simply let the user take a selfie, which will then be compared to the student card.

4.2 Improvements

We identified opportunities to deviate from the proposed architecture during the implementation to enhance simplicity and efficiency.

- For backend deployment, we utilized AWS Chalice, a microframework for writing serverless applications in Python. Chalice allowed us to express Lambda functions and event handlers declaratively, significantly reducing boilerplate code, which we showcase in the following section. It also facilitated code reuse between Lambda functions and enabled us to export the Lambda function definitions and packages to Terraform configuration. This streamlined the deployment process and improved maintainability.
- For authentication and data ownership, particularly in verifying student status, we instead consider the student initiating the verification session designated as the data owner. The process is as follows:
  1. The student requests a verification session and receives a secret token.
  2. This token authorizes the student for subsequent endpoints or API calls related to that session.

3. The SaaS platform issues the student a JSON Web Token (JWT) for the session.

The JWT, a cryptographically signed JSON object, includes claims about the user's name, university, and student status. This approach simplifies integration for customers (such as webshops or online businesses), as they only need to verify the JWT's signature and check the payload against, for example, checkout form data. That is, we can be considered an identity provider, rather than a verification service.

- AWS allows the use of pre-signed URLs for file uploads. Pre-signed URLs enable clients to upload files directly to an S3 bucket without passing through our lambda functions, reducing compute and egress costs.

Chapter 5

Final Implementation

This chapter details the final architecture of the project, highlighting deviations from the initial plan and providing an in-depth account of the technical implementation.

The project's codebase, comprising over 10,000 lines across more than 200 commits, is available in the GitHub repository volesen/kys. Due to the extensive nature of the codebase, this chapter will focus on the primary functionality and notable technical implementation details.

5.1 Demonstration

This section demonstrates the core functionalities of the application. Further details on the final project scope is provided in a following subsection.

We have created videos to illustrate the verification process:

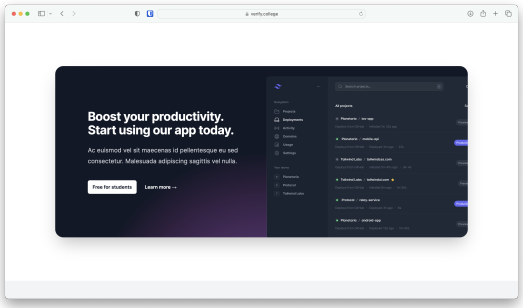
- Successful verification on desktop
- Failed verification on desktop
- Successful verification on phone
- Failed verification on phone

Additionally, Figures 5.1 and 5.7 showcase the main functionalities implemented.

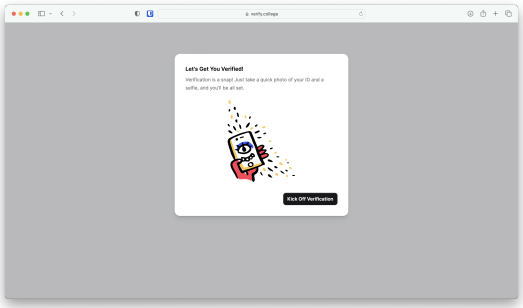
5.2 Architecture

As mentioned in Chapter 4, we consider minor deviations from the initial proposed architecture. The sequence diagram from the initial architecture is updated with the changes and elaborated to reflect the implementation details; see Figure 5.2.

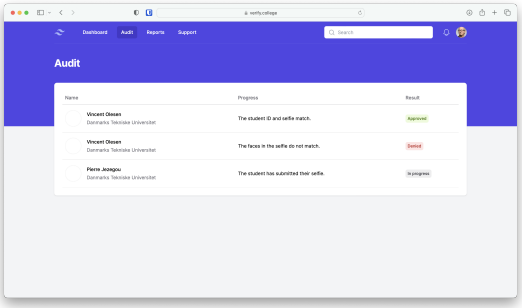
Figure 5.2 illustrates technical details such as handling pre-signed URLs, issuing JWTs, and marking steps that can be performed concurrently.



(a) Mockup of customer integration to offer student discounts



(b) Modal with the first step of the student verification flow



(c) Audit panel

Figure 5.1: Demonstration of the verification flow and audit panel for verification sessions.

5.3 Components and services

As mentioned in Chapter 4, we consider minor deviations from the initial proposed components and services. The architecture diagram from the initial architecture is updated with the changes and elaborated to reflect the implementation details; see Figure 5.3.

We will now elaborate on the technical implications of the changes to the components and services in the project.

In the proposed implementation, we suggest that images, such as the selfie and student card, be uploaded to S3 through a lambda function, utilizing a pre-signed URL. Here is how it works:

- The frontend requests a pre-signed URL to upload a picture from the API
- The API triggers a lambda function
- The lambda function returns a pre-signed URL (with the LabRole permissions - this will need to be modified in our final product)
- The frontend can use this URL to upload the picture via a PUT request (the link is only valid for a predetermined amount of time)

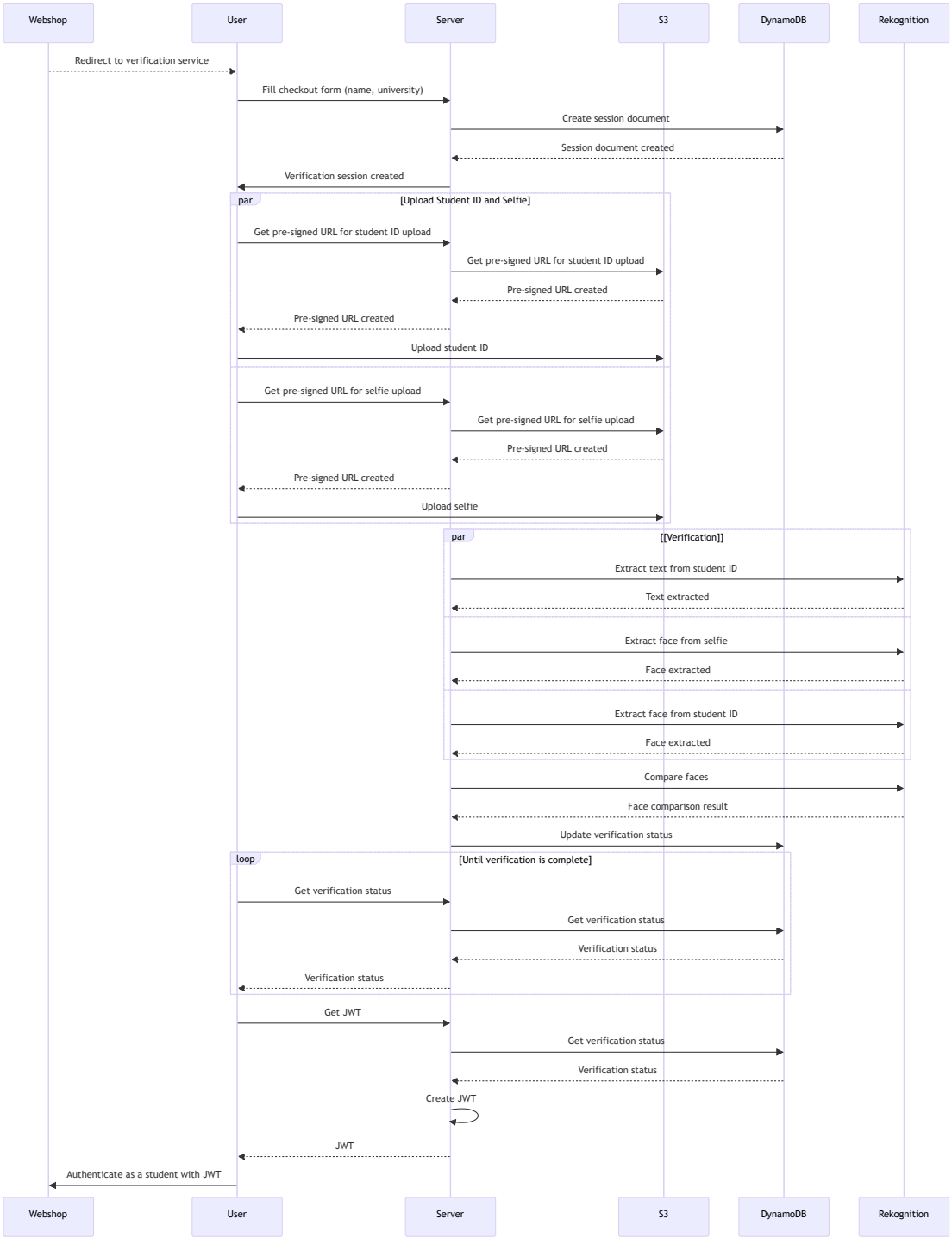


Figure 5.2: Sequence diagram of implementation. The `par` label represents transactions done in parallel. The `loop` label represents recurring transactions.

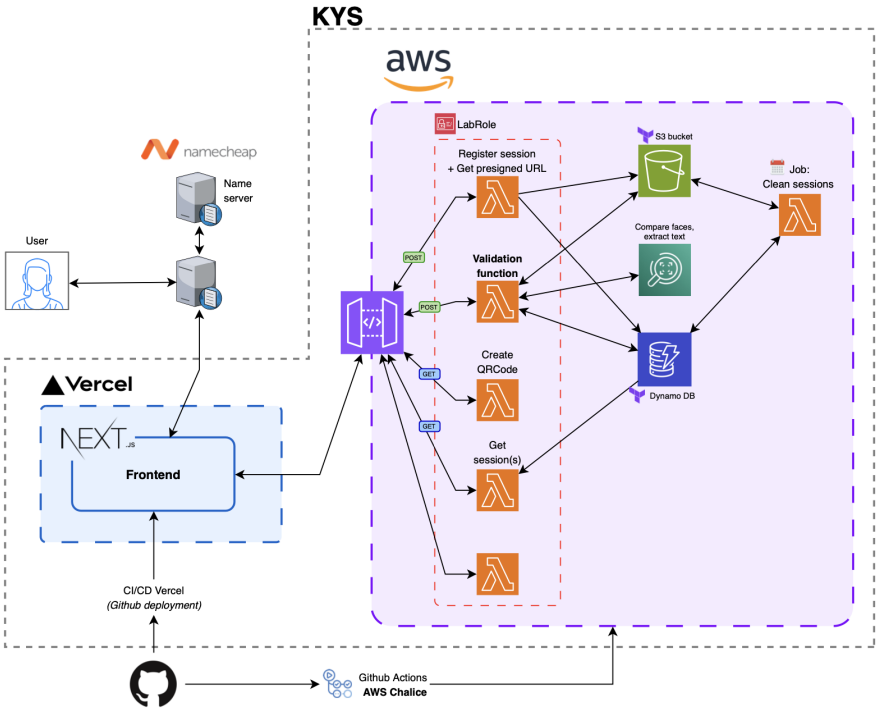


Figure 5.3: Components and services in the final architecture.

You can find all the material here for the API and here for the use of the presigned url in the frontend.

This approach reduces the traffic through the API Gateway and the use of the lambda function, making it more cost-effective.

5.3.1 From step functions to chalice

We had planned to use Amazon Step Functions, but after assessing our specific needs, it proved more appropriate to opt for Chalice. Chalice, which compiles directly into AWS Lambda functions, better suits our Python expertise and the integration required in our project. By using Chalice, we can define all our API routes and functions within a single framework, avoiding the complexity of creating and managing each function and each API route individually via Terraform. This approach dramatically simplifies our development workflow, enabling us to deploy serverless applications more quickly and efficiently while meeting our requirements perfectly (for more details, see 5.7.2).

5.3.2 From Amplify to Vercel

We initially planned to use AWS Amplify as outlined in our proposal, but we ultimately chose Vercel for our deployment needs. This decision was driven by Vercel's ease of deployment and our team's familiarity with the platform, al-

lowing us to ship the product more quickly. Additionally, we created a CI/CD pipeline from GitHub to Vercel, further streamlining our development process by automating deployments and ensuring continuous integration. Vercel's streamlined workflow and intuitive interface enable rapid development cycles, which is crucial for meeting our tight deadlines. While AWS offers extensive capabilities like scalability, load balancers, CloudFront CDN that would benefit a production environment, our current priority is speed and efficiency in getting our product to market. In the future, we could consider migrating our frontend to AWS to leverage these advanced features, but for now, Vercel provides the optimal solution for our immediate needs.

5.4 Twelve factor methodology

We did not make any difference in the final implementation, which affected the reflection and use of the Twelve-Factor methodology, as stated in Chapter 3.

5.5 Scope and Objectives

We successfully implemented the Minimum Viable Product (MVP) and achieved all the stretch goals defined in Chapter 3, which required extending the original scope.

In addition to the MVP and stretch goals we consider the following functionality:

- Taking accents and multi-line names into account on student cards.
- Refactoring the code base several times.

5.6 Project Structure

In this section, we will elaborate on the project structure, with an emphasis on the backend API.

To organize the components of our application, we utilized a monorepo structure, dividing the project into three main folders:

- `terraform`: Contains the Terraform configuration for provisioning the backing services on AWS, such as S3 buckets and DynamoDB instances.
- `web`: Contains the frontend application.
- `api`: Contains the backend application, structured following the Chalice application framework.

We chose the monorepo approach to streamline collaboration and maintain a unified version control system. This structure allowed all team members to access and update different parts of the project simultaneously, reducing integration issues and ensuring consistency across the application. Additionally, a monorepo facilitated easier dependency management, leading to a more cohesive

```
README.md
docs
terraform
web
api
    app.py
    chalicelib
        rekognition.py
        models
            session.py
        qrcode
            __init__.py
        session
            __init__.py
        verification
            __init__.py
```

Listing 1: Directory structure of the project, with an empahizes on the backend API.

development workflow and enhanced overall project maintainability. Synchronizing Terraform with the backend ensured that our infrastructure configurations and code deployments were always in sync, further reducing the risk of discrepancies and deployment issues.

The backend API follows the structure of a Chalice application. The application code is primarily located in the `chalicelib` directory. We adhere to best practices inspired by the Flask microframework, from which Chalice derives much of its structure. Specifically, we:

- **Decoupled services (and endpoints)**: We utilize Chalice Blueprints to organize different services and their endpoints into separate files and directories. This modular approach enables decoupling on a module level and enchanhes code navigation (code used together is located together).
- **Repository Pattern**: We implement the repository pattern in the `models` directory to handle database access, exposing database operations through domain objects. This pattern promotes the separation of concerns and simplifies testing and maintenance.

5.7 Implementation details

5.7.1 Development environment

We also furnish to every collaborator a well defined dockerized development environment (docker-compose here). With this methodology, we can benefit from the advantages of containerized applications such as portability, versatility, and mainly ease of Deployment. This ensures consistency across the team, simplifies setup. Indeed, providing the same developing environment to all the collaborators was one of our priorities to ensure coherence between all.

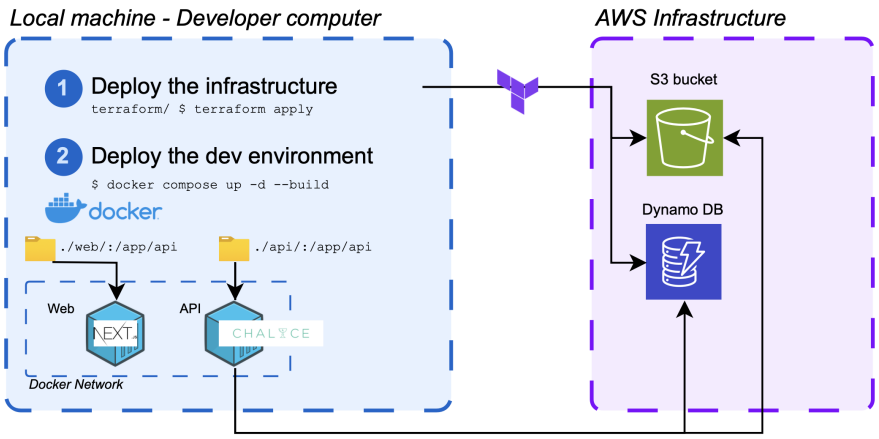


Figure 5.4: Development environment

The team member just has to run the terraform script to create the resources on his AWS learner lab account and then copy the output information as environment variables. Finally, he just has to run the docker-compose stack and can start to develop, debug, and test.

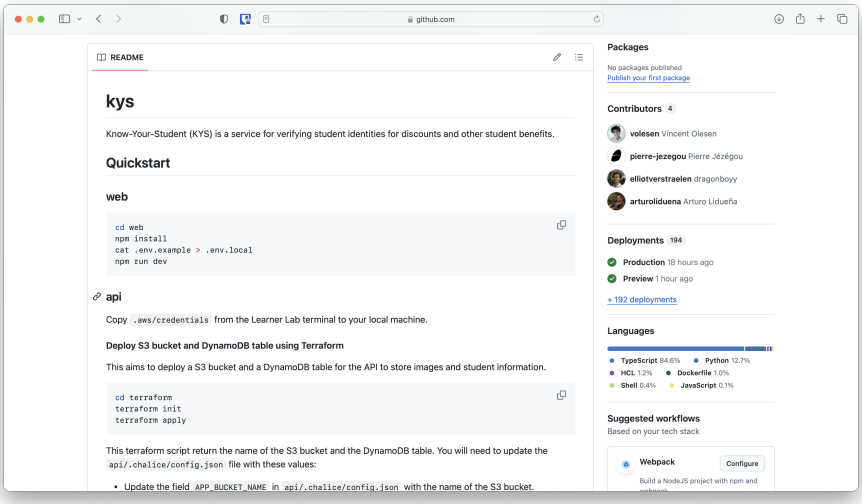


Figure 5.5: A screenshot of the rendered *Quickstart* section in the `README.md` file

We also wrote a quick-start guide to onboard people and allow them to understand how the project is structured and how to deploy it locally if they don't want to use the dockerized development environment.

5.7.2 AWS Chalice

In the development of this project, **AWS Chalice** was employed to facilitate the deployment of the entire logic infrastructure. AWS Chalice, a Python serverless microframework based on Flask, allowed for the efficient creation and deployment of Lambda functions, configuring API Gateway routes, and managing event handlers *from code* to be compiled to e.g. Terraform confiuration. By leveraging Chalice, we got first-class integration with other AWS services and additionally, Chalice's capability to use `chalice local` enabled local development and testing, significantly accelerating the development process and improving debugging efficiency.

```python
@app.route(
    "/session/{id}",
    methods=["GET"],
    content_types=["application/json"],
)
def get_session(id: str):
    session = session_repository.get_session(id)

    if not session:
        raise NotFoundError("Session not found.")

    return session
```

Listing 2: Example route showcasing what will be packaged as a lambda function and an endpoint in the API Gateway.

Likewise, we can decoratively react to S3 events like this:

```python
@app.on_s3_event(
    bucket=APP_BUCKET_NAME,
    events=["s3:ObjectCreated:*"],
)
def handle_s3_event(event):
    ...
```

Listing 3: Example event handler showcasing what will be packaged as a lambda function and a lambda function event trigger.

To further structure the application, we utilized a concept similar to Flask blueprints to separate independent parts of the application. This approach allowed us to modularize the codebase effectively (see listing 1), improving the organization and manageability of the project. By segmenting the application into distinct, reusable components, each handling specific functionalities, we ensured that the development and maintenance processes were more efficient and less prone to errors. This modular structure not only facilitated parallel development among team members but also enhanced the overall clarity and readability of the code.

5.7.3 Audit Panel and Admin Authentication

We implemented an audit panel to provide administrators with comprehensive oversight of the verification process and facilitate necessary interventions. The audit panel includes functionality for monitoring verification sessions. This section outlines the technical implementation of the audit panel and the associated authentication mechanisms for administrative access.

Audit Panel

The audit panel is accessible via a dedicated route and is protected by authentication. Administrators can view a comprehensive table that displays key metrics of recent verification activities.

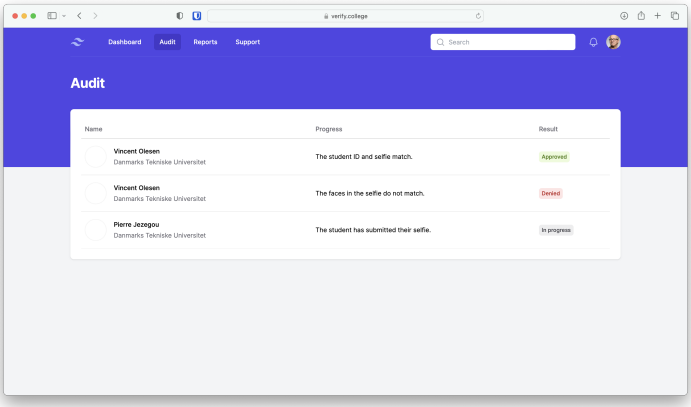


Figure 5.6: Audit panel interface.

Authentication

To ensure that only authorized personnel can access the audit panel and perform administrative actions, we implemented an authentication system using Amazon Cognito user pools.

Infrastructure The authentication system is implemented using Amazon Cognito, which provides user management and authentication functionalities. Below is the Terraform configuration for setting up the Cognito user pool and user pool client.

```
resource "aws_cognito_user_pool" "pool" {
  name = "kys-user-pool"
}

resource "aws_cognito_user_pool_client" "app_client" {
  user_pool_id = aws_cognito_user_pool.pool.id
  name         = "kys-app-client"

  explicit_auth_flows = [
    "ALLOW_REFRESH_TOKEN_AUTH",
    "ALLOW_USER_PASSWORD_AUTH",
    "ALLOW_USER_SRP_AUTH"
  ]
}
```

Listing 4: Terraform configuration for Amazon Cognito user pool and client.

sign-in The following TypeScript code snippet demonstrates how to configure the Cognito client and implement the sign-in functionality, which authenticates users and retrieves JWT tokens.

```
const config = {
  region: process.env.NEXT_PUBLIC_REGION,
  clientId: process.env.NEXT_PUBLIC_USER_POOL_CLIENT_ID,
}

export const cognitoClient = new CognitoIdentityProviderClient({
  region: config.region,
});

export const signIn = async (username: string, password: string) => {
  try {
    const command = new InitiateAuthCommand({
      AuthFlow: "USER_PASSWORD_AUTH",
      ClientId: config.clientId,
      AuthParameters: {
        USERNAME: username,
        PASSWORD: password,
      },
    });
    const { AuthenticationResult } = await cognitoClient.send(command);
    if (AuthenticationResult) {
      sessionStorage.setItem("idToken", AuthenticationResult.IdToken || '');
      sessionStorage.setItem("accessToken", AuthenticationResult.AccessToken || '');
      sessionStorage.setItem("refreshToken", AuthenticationResult.RefreshToken || '');
      return AuthenticationResult;
    }
  } catch (error) {
    console.error("Error signing in: ", error);
    throw error;
  }
};
```

Listing 5: TypeScript code for user sign-in using Amazon Cognito.

The implementation of the audit panel and the authentication mechanisms are crucial components of our project. They ensure that administrators can effectively monitor and manage verification sessions while maintaining strict access

control to sensitive data and functionalities. This approach not only enhances the overall security of the application but also provides administrators with the tools needed to maintain the integrity and reliability of the verification process.

5.7.4 Human-in-the-Loop Verification

As not always the automatic verification process with Amazon Rekognition could correctly verify the identity of the student, we have provided the functionality to perform a manual verification of the data inserted. This feature is implemented using the webhook provided by Slack to post a message on a specific channel. In order to automatically post a message when a verification is required, we have created a lambda function using Chalice that is triggered through a *DynamoDB Stream* activated on the DynamoDB table containing the information about the students whenever a change to the state of a row is modified. The function then proceeds to check if there is an error state and, if so, proceeds to compose the message that needs to be sent to slack, with the information about the student as well as the two images provided during the verification process (using a presigned url). As last, the function makes a *POST* call to the webhook with the message.

Inside the message on Slack we have the possibility to approve or deny the verification request using two buttons that will send a *POST* request to the URL specified in the Slack's application setting. In order to elaborate this request we have created an API endpoint that elaborates the requests from Slack in order to update the record in the databse and sends back a message to Slack to modify its original message with the decision made on the manual verification. See Figure 5.7

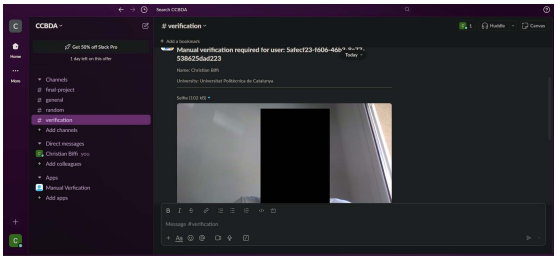
5.7.5 Deletion of Expired Sessions

Keeping the data indefinitely is not compliant with GDPR for multiple reasons. Furthermore, storing unsuccessful verifications wastes resources and increases costs. Therefore, we decided to add a feature to remove denied and non-verified accounts older than 7 days from our database and the pictures linked to these accounts. We set up a job to, every day, remove those elements. We used Chalice to deploy, which converts the code to scheduled AWS lambda. Chalice toolset significantly simplified the implementation (have to tell which element has to be removed):

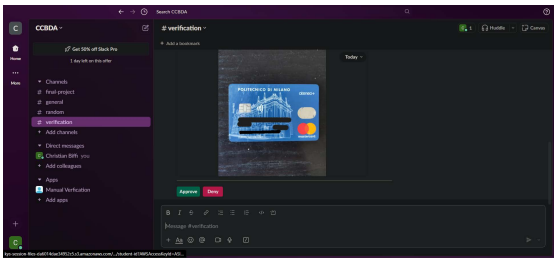
```
@bp.schedule(Rate(7, unit=Rate.DAYS))
def clean_expired_sessions():
    ...
    return deleted_sessions
```

Listing 6: Scheduler with chalice: delete expired sessions

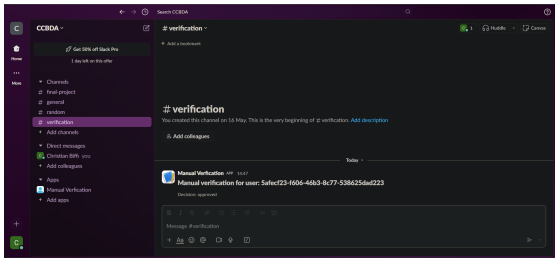
This feature leverages AWS Lambda's event-driven architecture to automate background processes with scheduled tasks. Chalice utilizes a Periodic scheduler to trigger Lambda functions based on a predefined period, enabling flexible



(a) The top of a manual verification as a Slack message, posted by the Slack bot.



(b) The bottom of a manual verification as a Slack message, posted by the Slack bot. The "Approve" and "Deny" buttons can be clicked.



(c) The manual verification Slack message after a verdict.

Figure 5.7: Demonstration of human-in-the-loop verification, using Slack as the platform for the back-office functionality.

scheduling for these automated tasks. This approach offers scalability, cost-effectiveness, and reliability as Lambda functions only run during execution, and AWS manages the underlying infrastructure.

5.7.6 Student Authorization

Our application implements a JSON Web Token (JWT) based approach to facilitate secure user information exchange with authorized third-party applications. This allows us to provide controlled access to user data while maintaining security and privacy. While lowering demand to re-requesting our services again for each verification check by a third party.

Our JWTs prioritize security and clarity by adhering to public/standard claims whenever possible. See in listing 7. This includes essential information like session ID (`sub`), `name`, and `roles`. Additionally, we leverage private claims for data specific to our application, such as `institute` in this example, to provide additional information above the standard claims. This approach ensures a balance between transparency and information control. The token itself includes standard claims like issuer (`iss`), issued-at (`iat`), expiration (`exp`), and a unique token ID (`jti`) for additional security.

```
{
  "sub": "3848a553-c341-4826-974c-6367d7005e0b",
  "name": "John Doe",
  "institute": "Universitat Politècnica de Catalunya",
  "roles": ["student"],
  "iat": 1715961878,
  "iss": "https://verify.college",
  "exp": 1716566678,
  "jti": "9f7c4cb4-aba2-44a7-a943-c654ea161e18"
}
```

Listing 7: Example JSON Web Token (JWT) for a successful verification session.

Using public/standardized claims enables easier integration of our service for third parties because most identity management libraries rely on JWT's public claims such as `roles` claim. See, for example, the simplicity of checking student eligibility using `roles` in the Asp.Net Core Identity system in listing 8.

```
[Authorize(Roles = "student")]
public class StudentDiscountController: BaseController
{
    public bool AllowStudentDiscount() => true;
}
```

Listing 8: Example of `roles` claim integration in Asp.Net Core Identity.

5.7.7 Infrastructure as code and CI/CD pipeline

Use our AWS student accounts

As every team member needs to develop some feature, we must ensure everyone is working with the same infrastructure (containing S3 buckets and DynamoDB tables...). Moreover, creating manually those elements is time-consuming and does not allow the developer to focus on new features or bug fixes. As a consequence, we decided to use infrastructure as code to speed up this process: everyone can have the same working environment (local development in python `venv` and chalice + `npm`, and AWS automated infrastructure).

You can find all the files in the `terraform` folder.

We faced one problem regarding the name of the buckets (which are defined worldwide and are unique). We added a unique ID at the end of the bucket name, generated at its creation.

```
resource "random_id" "bucket_id" {
  byte_length = 8
}

resource "aws_s3_bucket" "session_files" {
  bucket = "kys-session-files-${random_id.bucket_id.hex}"
  force_destroy = true
  tags = {
    Name    = "Session Files Bucket"
    project = "KYS"
  }
}
```

Listing 9: Randomize bucket name with terraform

We also wanted to get, at the end of the deployment process, the bucket name (randomly generated) and the iam role arn used by AWS (LabRole ARN). We used terraform *output* to get those information and use them later (in the `api/.chaluice/chalice.json` for instance).

```
output "session_files_bucket_name" {
  value = aws_s3_bucket.session_files.bucket
}
```

Listing 10: Use terraform outputs to get AWS S3 bucket name

As we can't create IAM roles, we had to use the role `LabRole` provided with our student account. With Terraform, we used `data` keyword to get the resource and use the resources created:

```
data "aws_iam_role" "lab_role" {
  name = "LabRole"
}
...
data "aws_iam_policy_document" "allow_put_from_lab_role" {
  statement {
    principals {
      type        = "AWS"
      identifiers = [data.aws_iam_role.lab_role.arn]
    }
    effect    = "Allow"
    actions   = ["s3:PutObject"]
    resources = ["${aws_s3_bucket.session_files.arn}/*"]
  }
}
```

Listing 11

By the way, we had to find a solution to have permissions for the LabRole to access S3 buckets. As we can not add policies to the role (forbidden by the learner lab), we had to create a policy on the S3 bucket to allow `s3:PutObject`— access to the bucket.

Deploy AWS infrastucture to "production"

We also want to have a functional CI/CD pipeline. Our script (see deploy.sh) works as follows:

- Create a package containing all the chalice workflow (lambda functions, API routes...) with all the dependencies required.
- Inject desired version for python, terraform (by default, the local version is used, but not necessarily supported by lambda)
- Deploy all our infrastructure (bucket, dynamodb table, cognito resources, chalice application...)

The final pipeline is not fully operational, because of the `LabRole` limitations: we need to copy the credentials locally, and run the `deploy.sh` by hand. It could be fully automated and used in a GitHub Action for example, if we could get fixed credentials (token).

Front-end CI/CD

When a push is done on the main branch, it automatically run our CD pipeline to deploy the new version on Vercel. This is managed by a Deployment in GitHub.

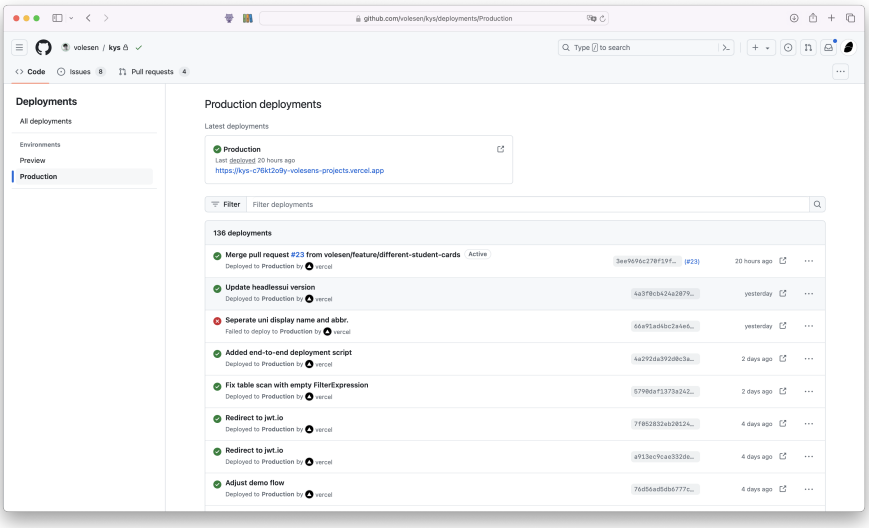


Figure 5.8: Push on main - Deployments

5.7.8 Observability

Observability and monitoring is a first-class citizen of the `chalice` framework, as we got logging and metrics enabled by default for our application. Figure 5.9 and 5.10, showcase logs and metrics used for debugging an issue.

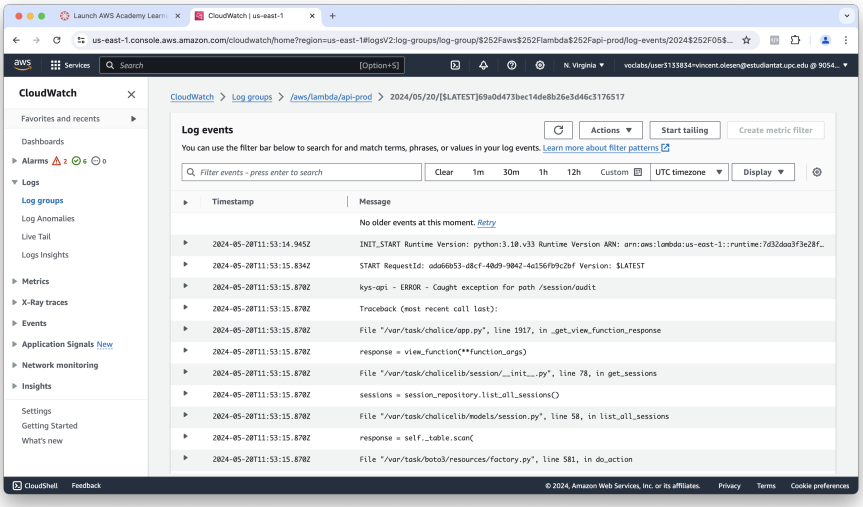


Figure 5.9: Logs from lambda function invocations used to debug a specific issue.

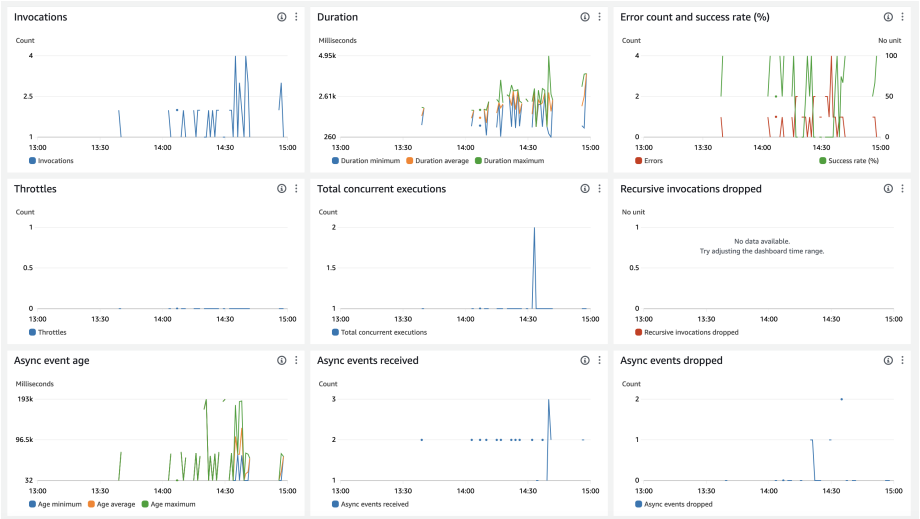


Figure 5.10: Logs from lambda function invocations used to debug a specific issue.

Chapter 6

Project Management

In this chapter, we detail the management strategies and practices employed throughout the project.

6.1 Methodology

Our methodology closely followed principles from *The Lean Startup* [1], focusing on defining a minimal viable product (MVP) and rapidly testing hypotheses, such as the viability of the verification flow. We opted against using Scrum due to the project's size and scope, as we found the associated overhead excessive.

Communication was maintained through weekly meetings on Discord and daily updates via WhatsApp. Task progress and technical issues were managed using GitHub Issues and GitHub Projects, which we will elaborate on in the subsequent sections.

6.2 Task Management

The initial architecture and prototype were proposed by Pierre and Vincent during the HackUPC hackathon, providing a foundation to divide the project into smaller, manageable tasks.

Task management was handled through GitHub Issues, with progress tracked using a Kanban board in GitHub Projects, as illustrated in Figures 6.1 and 6.2. This system integrated seamlessly with the GitHub flow for development, automatically updating tasks and the Kanban board based on pull requests.

The GitHub flow is as follows:

Main Branch The main branch is the primary development line, representing the stable version of our project. It reflects what is currently in production. Changes to this branch are thoroughly tested and reviewed to ensure quality.

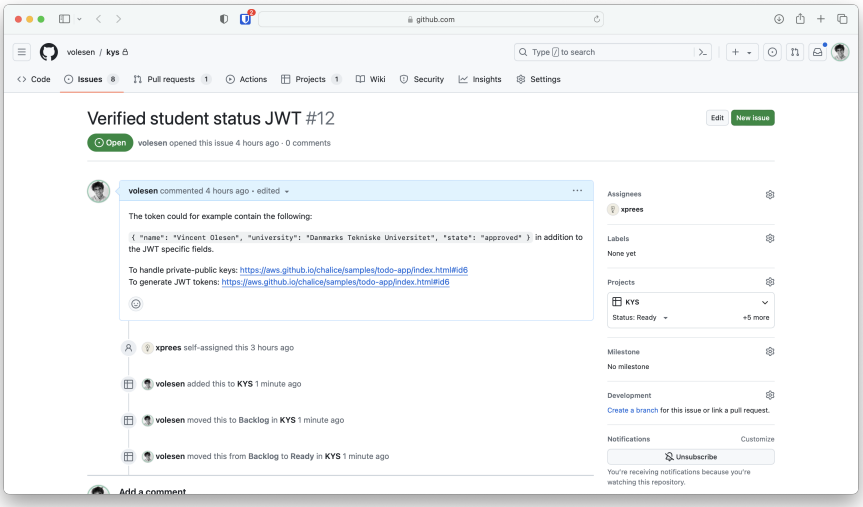


Figure 6.1: Sample functionality expressed as a GitHub issue, with a description and an assignee.

Feature Branches These branches are created from the main branch for developing individual features or tasks. Each feature branch allows independent work on different parts of the project without interference. Completed features are merged back into the main branch through pull requests.

Pull Requests Once work on a feature branch is complete, a pull request is created. This process allows team members to review the code, discuss issues, and ensure the changes meet project standards. Approved changes are merged into the main branch.

Issues Serving as our backlog, issues list all pending tasks, features, bugs, or enhancements. They provide a centralized repository for planned work, aiding in task prioritization, delegation, and scheduling. Tasks are managed using GitHub Issues and organized in a Kanban board format using GitHub Projects, offering clear visibility into progress.

6.3 Documentation

Technical documentation was maintained in the `docs/` directory within the repository. Additional context and documentation were provided in relevant issues and pull requests. The product documentation is included in this document. Our primary focus with documentation was to more easily onboarding team members to the project.

6.4 Time Management

Following the HackUPC hackathon, we determined that the MVP could be implemented well within the project deadline. Consequently, we focused on

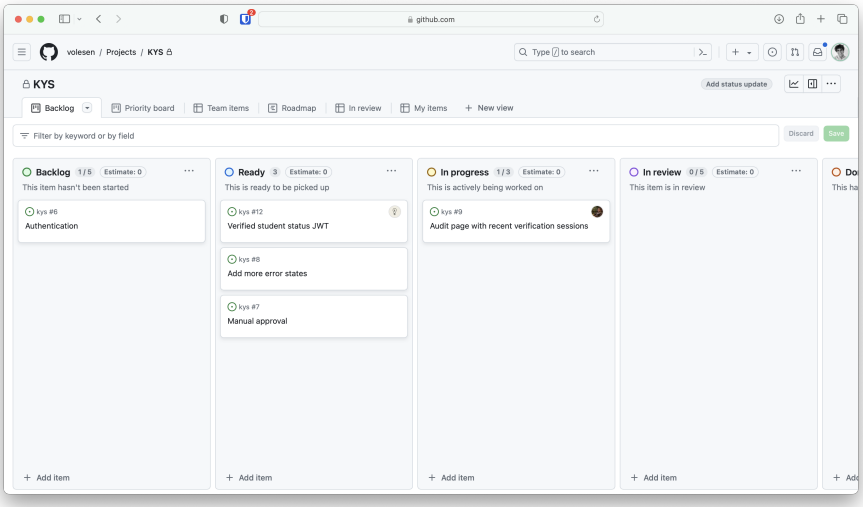


Figure 6.2: Kanban board using GitHub Projects

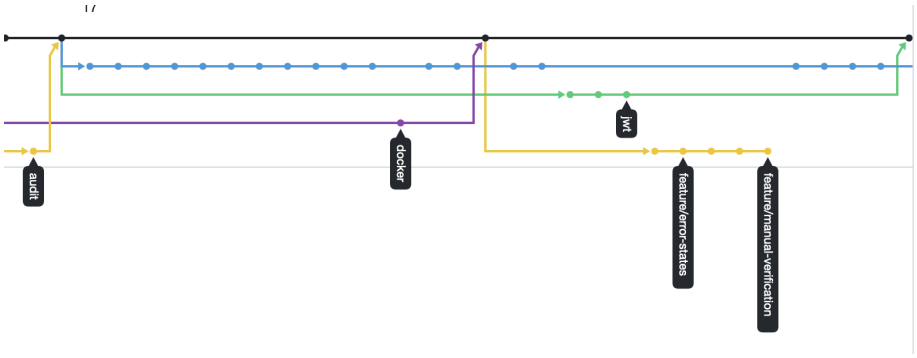


Figure 6.3: Branch Network: Overview of Project Branches

estimating the complexity of issues and delegating them appropriately, rather than estimating the time required for each feature.

6.5 Contributions

The initial planned time by project phase from the project proposal is given in Table 6.1.

The final time contribution by tasks, project phase and team member is given in Table 6.2.

Our initial estimates for the time spent on tasks were quite off because (1) two team members joined the team after the initial project proposal, and (2) we spent significantly less time on architecture and research and spent more time on coding. However, here, we acknowledge that the initial prototype made by

Project Phase	Hours
Project Selection	3
Architectural Design	7
Research	30
Meetings	20
Coding	50
Documentation	30
Final Report	20

Table 6.1: Distribution of hours spent on different project phases.

	Organization	**Architecture**	**Research**	**Coding**	**Documentation**
Vincent					
UI screens	2			12	
UI backend integrations		4		6	
`chalice` migration		4	4	4	4
Vercel migration				2	
CI/CD setup				2	
Code review (and fixes)	2			6	
Report					18
Presentation					4
Subtotal	4	8	4	32	26
Pierre					
Manual AWS implementation		10		5	
Lambda functions & API routes				8	
Terraform	1	1	2	6	1
Development environment	1			2	
DB access refactoring		1		5	
Handle all student id	1			2	
Report					12
Subtotal	3	12	2	28	13
Arturo					
Audit page	1			4	
Cognito integration	1	4	8	8	
Terraform refactor		4		4	
Sign-in page				4	
Report					4
Subtotal	2	8	8	20	6
Václav					
Observability	1		4	3	
Sessions Cleanup	1	1	1	4	
JWT Session export	1	1	3	4	1
Report					8
Subtotal	3	2	8	11	9
Christian					
Manual verification	1	2	2	18	1
Slack integration	1	1	3	3	
Report					2
Subtotal	2	3	5	21	3
Elliot					
Elaborated error states	2			6	
Mount volume for local dev			4	1	
More robust verification	1			8	
Report					
Subtotal	3		4	15	
Total	17	33	31	133	57

Table 6.2: Hours spent, distributed by task and team member.

Vincent and Pierre could be considered research or architecture.

To discuss, we must also acknowledge that research, architecture, and coding come in cycles or iterations. Initially, we had a waterfall approach to the time

plan, which did not reflect how we ended up organizing the project and the associated time spent.

We initiated the project with a sprint to kickstart development quickly, condensing a significant amount of work into the first few weeks. Within this initial period, an MVP (Minimum Viable Product) was developed, allowing thoughtful improvements and refactoring to enhance the codebase, making it cleaner and more reusable. Additionally, documentation was created throughout the project to facilitate onboarding. “‘latex

Chapter 7

Discussion and Conclusion

In this section, we will restate our contributions and address the main takeaways from the project.

Our project yielded a functioning application that met all the project's desiderata, even extending the initial scope.

Several challenges arose during the project, leading to important observations. Key points are discussed below:

Security Practices: Implementing and exercising security best practices was challenging within the Learner Lab environment, as we could not set up custom policies and IAM roles.

Local Development Environment: The inability to fully emulate AWS services such as S3 and DynamoDB locally posed significant challenges. Testing reactions to S3 events required deployment to a development environment in the cloud.

Lambda Function Latency: Deployment constraints, specifically the inability to deploy Lambda functions outside the `us-east-1` region, resulted in latency issues, compounded by cold-start latency. This could have been mitigated by e.g. utilizng the Amazon `Lambda@Edge` service, which would deploy the application at edge infrastructure.

Team Size and Coordination: Our team size exceeded the "two-pizza team" rule advocated by Amazon, making it harder for members to take strong ownership of specific parts of the application [2]. This diluted responsibility. Furthermore, the large team size, it requires a considerable upfront effort to delegate tasks, that are not blocking each other and takes team member schedule into account.

Software Engineering Improvements: Implementing event sourcing could have improved auditability and decoupled state handling from the verification process, leading to a more robust and maintainable system.

The project highlighted the complexity of developing cloud-native applications and emphasized the importance of following best practices in security, development, and team management. Despite the challenges, the project was successful in delivering a robust application that not only met but also exceeded the project requirements.

Bibliography

[1] Eric Ries. *The lean startup: how today's entrepreneurs use continuous innovation to create radically successful businesses*. 1st ed. OCLC: ocn693809631. New York: Crown Business, 2011. ISBN: 9780307887894.

[2] *Two-Pizza Teams - Introduction to DevOps on AWS*. URL: https://docs.aws.amazon.com/whitepapers/latest/introduction-devops-aws/two-pizza-teams.html (visited on 05/22/2024).

[3] Adam Wiggins. *The Twelve-Factor App*. 2017. URL: https://12factor.net/ (visited on 05/20/2024).