Scanalyze

Authors: Anthony David, Jarod Streckeisen, Timothée Van Hove

Scanalyze

1. Introduction

1.1 Purpose

The purpose of this document is to provide comprehensive guidance for the development and deployment of the Scanalyze application. This document aims to outline the user, functional and non-functional requirements that the system needs to fulfill. It serves as both a blueprint for the implementation phase and a standard for system validation.

1.2 Project Scope

Scanalyze is designed to revolutionize the way individuals manage and understand their grocery shopping expenses. Using cutting-edge technologies like Optical Character Recognition (OCR), the app will capture information from grocery receipts, eliminating the need for manual entry. The application is not just limited to tracking; it also offers data analytics and visualization features to provide actionable insights into the user's spending patterns.

This document is intended for a range of readers including but not limited to project managers, developers, testers, and stakeholders. It will serve as the go-to reference for understanding the functionalities that need to be developed, the quality standards that need to be met, and the data flows that need to be established.

1.4 Document Conventions

In this document, `Shall` is used to indicate a required feature. `Should` indicates a desirable but not mandatory feature. All technical terminologies used are explained in the glossary in the appendices.

1.5 Project Goals

- To provide a user-friendly mobile interface for scanning grocery receipts.
- To automate data extraction from receipts for efficient tracking.
- To offer robust data analytics for users to understand their spending habits.
- To ensure secure user authentication and data storage.

By offering these functionalities, Scanalyze aims to become an indispensable tool for smart grocery shopping and personal finance management.

2. System Overview

Scanalyze is designed as a multi-tiered application involving client-side, server-side, and database components. Each of these elements has specialized tasks and together they form a cohesive ecosystem for effective grocery expense tracking. Below is a breakdown of the system's primary components and their interrelationships.

2.1 Android Mobile Application

The mobile application serves as the user interface and is responsible for capturing grocery receipts through the device's camera. It utilizes Optical Character Recognition (OCR) to extract textual data from these receipts. The app also handles user authentication and data visualization through various graphs and charts.

Key Features:

- User registration and login
- Receipt scanning
- Data visualization
- Offline data caching

2.2 Web Server

The web server acts as the intermediary between the mobile application and the database. The server processes the OCR data received from the mobile application and feeds the database. It is the server responsibility to link (index) the abbreviated products on the receipt to the existing product in the database. It uses the database user data to generate statistics, then returns this statistics data to the mobile application. The server is also responsible for handling user authentication tokens.

Key Features:

- Data processing and analysis
- Receipt - database indexation
- API endpoints for client-server communication
- Token-based authentication

2.3 Database

The database stores detailed information about various grocery products, including their descriptions, categories, and indicative prices. This data is queried by the web server whenever a new grocery receipt is processed. The database also securely stores user information such as hashed passwords and expense histories.

Key Features:

- Storage of product descriptions and categories
- User data management
- Scalable data schema

2.4 System Interactions

- **User Authentication**: The user authentication is managed by AWS Cognito which is a user directory, an authentication server, and an authorization service for OAuth 2.0 access tokens and AWS credentials. With Amazon Cognito, the application can authenticate and authorize users with JSON Web Tokens to maintain secure sessions.
- **Data Transfer**: After a user scans a receipt, the mobile app sends the extracted text to the web server for processing.
- **Data Retrieval**: The web server queries the database based on the received text, retrieves matching product descriptions, and returns them to the mobile application for display and analysis.
- **Data Visualization**: The mobile application uses the received data to generate graphs and charts for a visual representation of the user’s expenses.

2.5 Technological Choices

In the development of the Scanalyze application, a thoughtful selection of technologies has been made to ensure efficient and secure functioning across its multi-tiered architecture. These choices have been made in consideration of the application's requirements, performance expectations, and compatibility with the intended functionality.

Android Mobile Application

- Programming Language: Kotlin, because it's the recommended language for new android application
- Development Environment: The application is developed using Android Studio
- Compatibility: The application is designed to be compatible with Android versions 8.0 "Oreo" (API 24) and above.

Optical Character Recognition (OCR)

- OCR Library: The application employs the use of either ML Kit or Tesseract for Android, depending on the performance of each technology

Data Visualization

- Graphing Library: For generating graphs and charts within the mobile application, **MPAndroidChart** is used.

Web Server

- Server-Side Framework: The web server is developed using Node.js and Express.
- Deployment Infrastructure: The web server is hosted on AWS Elastic Beanstalk, utilizing Amazon EC2 instances.

User Authentication and Security

- User Authentication: AWS Cognito is used to manage user authentication and authorization. JWTare used for maintaining secure user sessions.
- Data Transmission Security: All data transmitted between the mobile application and the web server is encrypted using HTTPS

Database

- The database is built using AWS DocumentDB, a fully managed NoSQL database service that is compatible with MongoDB.

3. User Requirements

3.1 User Registration and Account Management

UR1.1: Users shall be able to register with an email and password.

UR1.2: Users shall be able to recover forgotten passwords through email.

3.2 Compatibility

UR2.1: At least, the application shall scan and recognize edible products.

UR2.2: The application shall work with the following grocery stores: Migos, Coop, Aldi, Lidl.

3.3 Receipt Scanning

UR3.1: Users shall be able to capture images of grocery receipts using their phone's camera.

UR3.2: OCR shall be applied to extract readable text from the receipt images.

UR3.3: Receipts data shall be sent to the backend server via a REST API.

UR3.4: The application shall allow users to upload images to be processed.

3.4 Data Management and Visualization

UR4.1: Users should be able to see a numeric version of their receipts in the mobile application.

UR4.2: Users shall see a summary of their monthly and yearly expenditures.

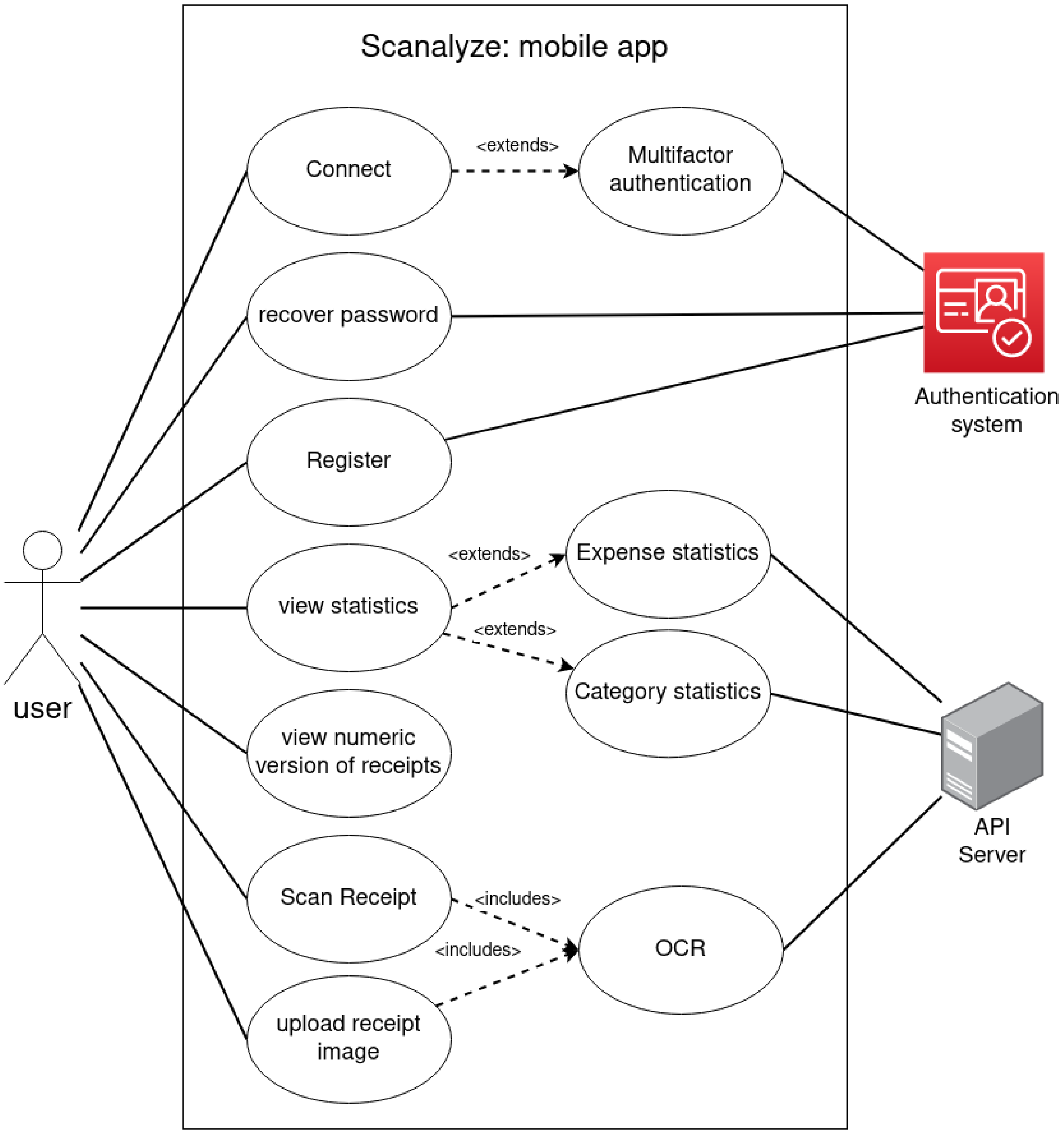
UR4.3: Users shall be able to view graphical representations of their spending by product category.

3.5 Offline Access

UR5.1: Users shall be able to scan and store receipts locally on their device without requiring an internet connection.

UR5.2: Users shall be able to visualize charts and graphs event without an internet connection.

3.6 Use case diagram



4. Functional Requirements

4.1 Mobile Application

4.1.1 Technologies

FR1.1: The app must be developed with Kotlin.

FR1.2: The app must be developed with Android studio.

FR1.3: The app must be compatible with Android 8.0 "Oreo" (API 24) and above.

4.1.2 User Authentication

FR2.1: The app shall allow users to register with a unique, valid email and password.

FR2.2: The app should provide a "Forgot Password" option that will send a reset link to the registered email address.

FR2.3: The app should offer multi-factor authentication for enhanced security.

FR2.4: The app shall use AWS Cognito for the account creation, log-in.

FR2.5: The app shall use a JWT provided by AWS Cognito to interact with the server REST API.

4.1.3 Receipt Scanning

FR3.1: Use ML Kit or Tesseract for android to extract text from the scanned receipts.

FR3.2: Users shall be able to scan grocery receipts using the camera of their mobile device.

FR3.3: OCR functionality shall extract the product names and prices from the scanned image.

FR3.4: When receiving a "correction request" , the app shall prompt the user to select from a range of products, the one that the server didn't recognise on the receipt data.

4.1.4 Data Analysis and Visualization

FR4.1: The application must use MPAndroidChart for mobile app graphs.

FR4.2: The app should display an expense summary in a dashboard format, including metrics like 'Monthly Expenditure,' 'Average Basket Size,' etc.

FR4.3:Users should be able to view their spending trends over different time frames: weekly, monthly, and annually.

FR4.4: The app should provide different types of visualizations, such as pie charts for category-wise spending and line graphs for temporal trends.

FR4.5: The app should provide temporal price visualization for any every product.

4.1.5 Data Caching

FR5.1: The app shall cache scanned receipt data when the app is used offline, or if the server is unreachable.

FR5.2: The app should synchronize with the server when online access is restored.

FR5.3: The app should have the capability to validate cached data against the most recent version in the database.

4.2 Web Server

4.2.1 Technologies

FR6.1: The web server shall be deployed on AWS Elastic Beanstalk, using S3 and EC2 instances as infrastructure.

FR6.2: The webserver must be developed using Node.js and Express.

4.2.2 Data Processing

FR7.1: The server shall parse the text data received from the mobile app and correlate it with stored product descriptions in the database.

FR7.2: The server shall send a "correction request" to the mobile app if some products do not match the database

4.2.3 User Authentication

FR8.1: The server shall validate JWT (JSON Web Token) provided by the mobile application to authenticate the user's identity.

4.2.4 RESTful API

FR9.1: The server shall provide RESTful API endpoints to facilitate communication with the mobile app.

4.3 Database

4.3.1 Technologies

FR10.1: The database is hosted on an AWS DocumentDB cluster

4.3.4 Data Schema and Indexing

FR11.1: The documents in the database shall have the following minimal structure:

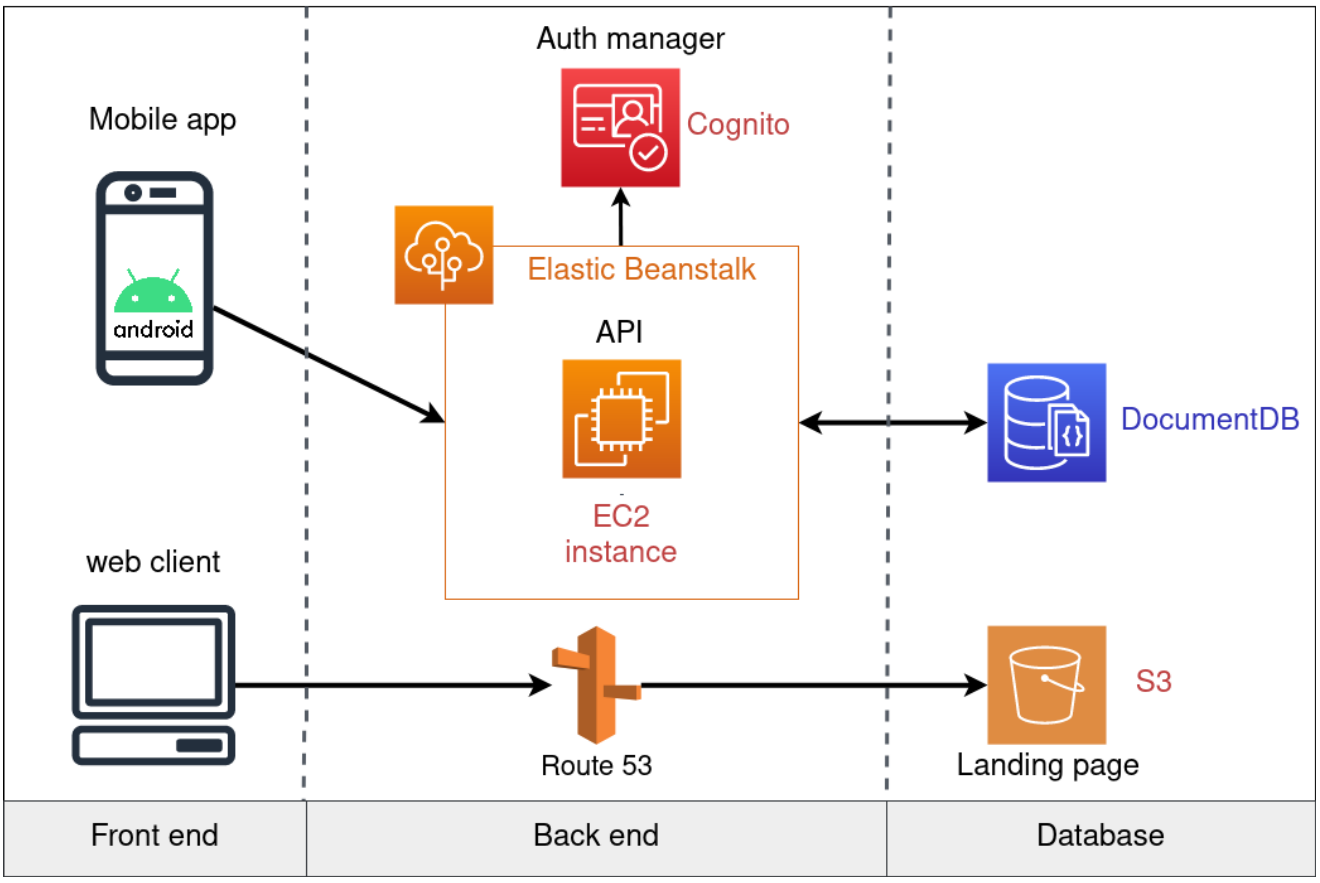
`Tickets`

```
{
    "_id" : ObjectId,
    "user_id" : ObjectId,
    "date" : Date,
    "shop_name" : String,
    "shop_branch" : String,
    "Products" : [
        {
            "product_id" : ObjectId,
            "product_name" : String,
            "quantity" : Double,
            "unit_price" : Double,
            "discount_amount" : Double
        },
        {
           ...
        },
        ...
    ],
    "Total" : Double

}
```

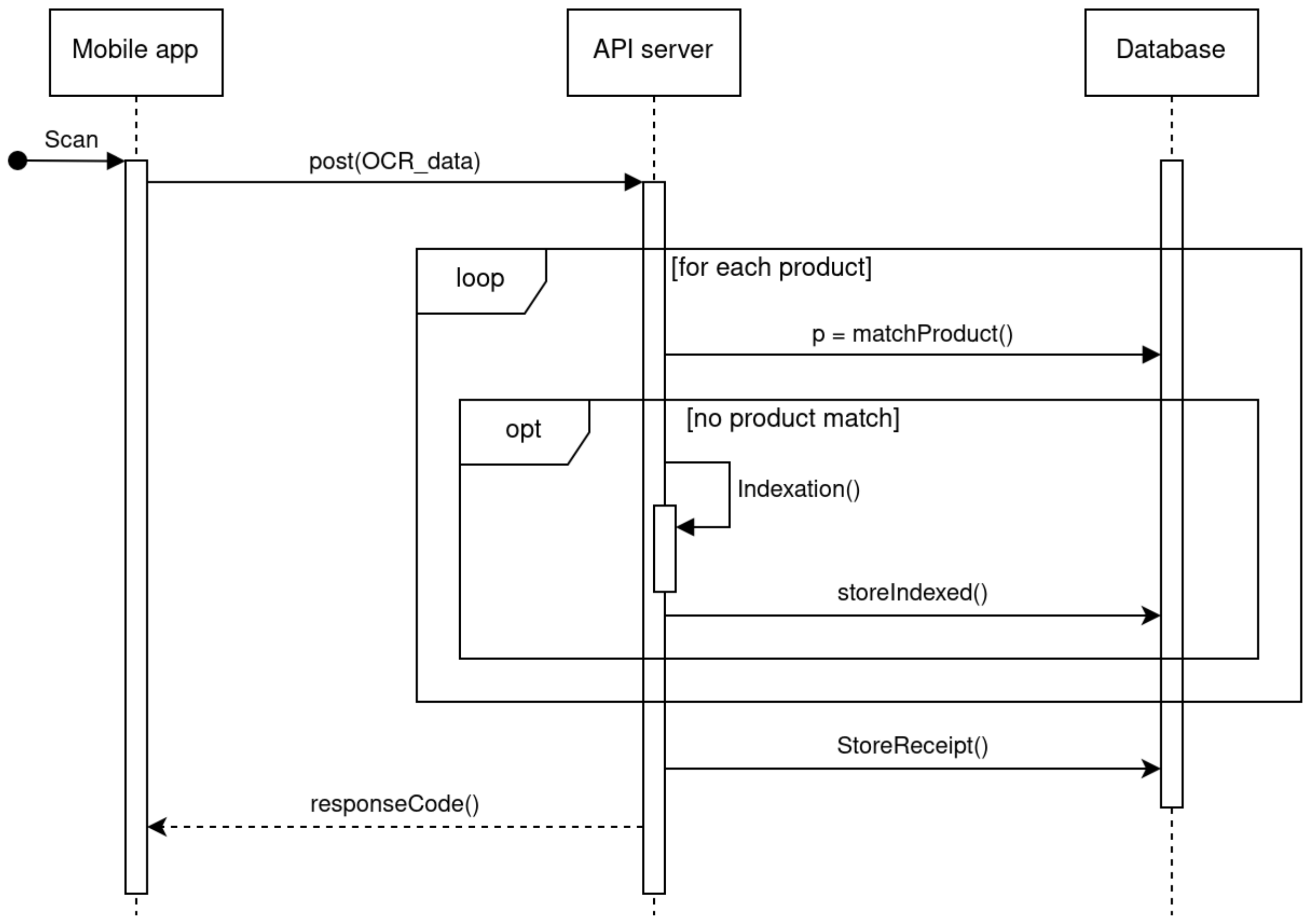
Products

```
    {
        "[shop_name]": [
            {
                "_id" : ObjectId,
                "product_name" : String,
                "abbreviated_name" : String,
                "indicative_price" : Double,
                "category" : String,
            }

        ]
    }
```

4.4 Architecture diagram



4.5 Main scan sequence



5. Non-Functional Requirements

NFR1: The system shall be designed to support up to 10 users

NFR2: All data transmission between the mobile app and the web server shall be encrypted using HTTPS.

NFR3: The initial release shall support English, with the architecture designed to easily add additional languages in the future.

NFR4: receipts shall be scanned in a maximum of 3 seconds

NFR5: The system shall implement mechanisms to verify the integrity of the data, ensuring that it is free of corruption during its life cycle.

6. Development Methodologies

To accommodate our tight 3-week schedule and ensure a high-quality end product, we are adopting a blend of Agile-Scrum and Extreme Programming (XP) methodologies. Our sprints will span one week each, enabling rapid cycles of development, testing, and feedback.

6.1 Team Composition

Anthony David: UX/UI Designer, Database Architect and Product Owner

Responsibilities: Lead on website design, mockups, mobile app design, database creation, product logo, and color themes. As the Product Owner, Anthony will also prioritize the backlog and interact with stakeholders.

Jarod Streckeisen: Full-stack Web Developer, Technical lead

Responsibilities: In charge of the landing page, API server, product-matching algorithms, and potential help with the database. Jarod will be responsible of the CI/CD pipeline of the API web server. Also responsible for API testing and performance testing.

Timothée Van Hove: Mobile Application Developer, Scrum Master

Responsibilities: Responsible for mobile application development and automated unit and integration tests as a part of the development process. As Scrum Master, Timothée will facilitate Scrum ceremonies and remove obstacles that might impede development progress. Timothée will be responsible of the CI/CD pipeline of the mobile application.

6.2 Sprint Plan

Sprint 1 (Week 1)

Deliverables and role attribution:

1. Development Methodology Document (Timothée)
2. Requirement Specification Document (Timothée)
3. Landing Page (Jarod)
4. Mockups for the Android application (Anthony)
5. CI/CD Pipelines (Timothée & Jarod)
6. 5-minute pitch preparation (Anthony)

Sprint 2 (Week 2)

role attribution:

1. Base views of the android app (Timothée)
2. Scanner feature (Timothée)
3. Database infrastructure & integration (Anthony)
4. Scrapping scripts and populate database (Anthony)
5. API routes (Jarod)
6. Matching algorithms (Jarod)
7. Cognito integration (Jarod)

Minimum viable product:

1. Working API
2. Working scanner with data handling
3. Working database with connection to the API server

Sprint 3 (Week 3)

role attribution:

Depending on the backlog at the end of week 2

Minimum viable product

1. An Android application that fulfill all the requirements mentioned above

Deliverables:

1. 15-minute presentation including a video demo

6.3 Development Workflow and Tools

6.3.1 Git branching strategy and conventions

We will use a feature-branch workflow. Each new feature or bug fix will be developed in a separate branch. Each feature branch must be merge into the develop branch. The main branch will remain deployable at all times.

branches: Every feature branch must begin with `ft` followed by the name of the feature. words are separated by hyphens. Example : `ft-my-feature`

6.3.2 CI/CD Pipeline

API Server pipeline

Every time a push is made on the main branch, the API is deployed on AWS Beanstalk. We decided to use AWS CodePipeline instead of Github actions because it is easier to manage within the same AWS infrastructure. CodePipeline is divided in three stages. Sources, Build and Deploy.

Sources : Connect to the repository and listen to code change

Build : Choose the correct folder containing the API

Deploy : Deploy the API to AWS Beanstalk

Because we are using a monorepo, we need to tell AWS which folder need to be deployed. This is configured in the **buildspec.yaml file** located in the root directory of the repository

```yaml
version: 0.2

artifacts:
  base-directory: 'backend'
  files:
    - '**/*'  # Artifact to be used in the next pipeline stage
```

Mobile app pipelines

For the mobile application, we chose to use Github actions. Every time that a push is made on the main branch, the action set up the environment, then executes the unit tests.

```yaml
# .github/workflows/test.yml
name: Android Test

on:
  push:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2
    - name: Grant execute permission for gradlew
      run: chmod +x android_app/gradlew

    - name: Set up JDK 17
      uses: actions/setup-java@v1
      with:
        java-version: 17

    - name: Writing the github secrets to the local.properties file
      run: |
        echo "keyAlias=${{ secrets.RELEASE_KEY_ALIAS }}" >>
android_app/local.properties
        echo "keyPassword=${{ secrets.RELEASE_KEY_PASSWORD }}" >>
android_app/local.properties
        echo "storeFile=${{ secrets.RELEASE_STORE_FILEPATH }}" >>
android_app/local.properties
        echo "storePassword=${{ secrets.RELEASE_STORE_PASSWORD }}" >>
android_app/local.properties

    - name: Run Unit Tests
      run: |
        cd android_app
        ./gradlew test
```

Using this extra GitHub action, when a release is made on GitHub, the APK is built and placed in an S3 bucket. This makes it easy for us to add a direct download link for the APK on our landing page, so users can download the app directly.

```
# .github/workflows/deploy.yml
name: Android Deploy

on:
  release:
    types: [created]

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up JDK 17
      uses: actions/setup-java@v1
      with:
        java-version: 17
    - name: Decrypt keystore file
      run: openssl enc -aes-256-cbc -d -in android_app/scanalyze-release-
key.jks.enc -out android_app/scanalyze-release-key.jks -pass
env:OPENSSL_PASSWORD
      env:
        OPENSSL_PASSWORD: ${{ secrets.RELEASE_ENCRYPTED_KEYSTORE_PASSWORD
}}


    - name: Writing the github secrets to the local.properties file
      run: |
        echo "keyAlias=${{ secrets.RELEASE_KEY_ALIAS }}" >>
android_app/local.properties
        echo "keyPassword=${{ secrets.RELEASE_KEY_PASSWORD }}" >>
android_app/local.properties
        echo "storeFile=${{ secrets.RELEASE_STORE_FILEPATH }}" >>
android_app/local.properties
        echo "storePassword=${{ secrets.RELEASE_STORE_PASSWORD }}" >>
android_app/local.properties

    - name: Build APK
      run: |
        cd android_app
        ./gradlew assembleRelease

    - name: Rename APK
      run: mv android_app/app/build/outputs/apk/release/app-release.apk
android_app/app/build/outputs/apk/release/scanalyze.apk

    - name: Deploy APK to S3
      run: aws s3 cp
./android_app/app/build/outputs/apk/release/scanalyze.apk
s3://${{secrets.AWS_S3_BUCKET}} --region ${{secrets.AWS_REGION}}
      env:
        AWS_S3_BUCKET: ${{secrets.AWS_S3_BUCKET}}
        AWS_ACCESS_KEY_ID: ${{secrets.AWS_ACCESS_KEY_ID}}
        AWS_SECRET_ACCESS_KEY: ${{secrets.AWS_SECRET_ACCESS_KEY}}
```

6.4 Task Decomposition and Allocation

Given the tight schedule, we recognize the potential for risks such as scope creep, development bottlenecks, and unforeseen technical challenges. We'll have a brief daily stand-up to discuss what each member is working on, any roadblocks, and how they can be resolved quickly.

6.5 Agile Methodology

In navigating the tight deadlines and evolving requirements of the Scanalyze project, we have opted for the Agile methodology. This approach encourages iterative development, rapid feedback loops, and stakeholder engagement—all of which are vital given our three-week timeframe and yet-to-be-finalized second sprint. With its focus on incremental deliverables and stakeholder feedback, Agile enables us to manage risks proactively, ensuring that the final product will align with both user needs and academic expectations.

6.6 XP (Extreme Programming) Practices

While Scrum provides us with a framework for project management, Extreme Programming (XP) offers specific technical practices designed to improve software quality and responsiveness to changing requirements. For the Scanalyze project, we've chosen to integrate the following XP practices:

1. **Pair Programming**: Given our team's diverse strengths, pair programming will be instrumental. For instance, when working on the mobile app's integration with the API server, Timothée could pair with Jarod, leveraging Jarod's expertise in full-stack development. This not only ensures high-quality code but also facilitates knowledge transfer within the team.
2. **Test-Driven Development (TDD)**: Quality assurance is paramount. By writing tests before the actual code, TDD ensures that our application meets its requirements from the outset. For example, as Timothée develops the scanning algorithms, tests can be written first to define the expected behavior.
3. **Continuous Integration and Continuous Deployment (CI/CD)**: Given our tight schedule, integrating changes frequently and ensuring they work seamlessly is vital. Our CI/CD pipelines, implemented using GitHub Actions, will automatically test and deploy our code, ensuring consistent product quality and rapid release cycles.
4. **Collective Code Ownership**: To promote flexibility within our team, every team member should feel comfortable working on and improving any part of the codebase. This approach ensures that we're not bottlenecked if a member is occupied with another task and fosters a collaborative code culture.