Tank Hunter – Mobile Application for Armoured Fighting Vehicle Detection

Final Project Report

DT282 / TU858
BSc in Computer Science International

Igor Alekhnovych

D18130122

Sean O'Leary

School of Computer Science

Technological University Dublin

31.03.2023

Abstract

The purpose of this project is to create a cross-platform mobile application to allow loyal citizens to assist the Armed Forces, the Main Directorate of Intelligence and the Security Service of Ukraine in gathering information regarding the movements of enemy military columns and supply convoys, while enhancing the quality of data obtained, enforcing security, and improving user experience in comparison to some of the existing solutions.

Tank Hunter is an application that will be developed as an outcome of this project. It will provide users with all the tools necessary to take an image of enemy armoured fighting vehicles and upload alongside geolocation and other useful information. Prior to being uploaded, the image will be validated in the background using deep learning algorithms for presence of military vehicles. As a result, having some preliminary information regarding armoured fighting vehicles detected will also facilitate workload for military analysts.

Declaration

I hereby declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed:

Student Name:

Igor Alekhnovych

Date:

31.03.2023

Acknowledgements

I would like to express a special thanks to each and every person who helped me and supported me throughout the development of this project, including many of my friends, especially Anna-Mariia and Alec. A special thanks to my parents, Oleksandr and Maryna, and my brother, Vadym, who supported me from the very beginning of my studies at TU Dublin.

I want to say a massive thank you to all the current and former service men and women of the Armed Forces of Ukraine for protecting my country from the invaders and for giving me the opportunity to return home after I graduate. I want to acknowledge the heroism of millions of ordinary Ukrainians that inspired the world to unite against tyranny and barbarism.

I wish to thank Dr. Tuomo Hiippala of University of Helsinki for sharing some thoughts regarding this project and providing a dataset of armoured fighting vehicles that proved to be very useful for the research and development sections of this project.

I wish to acknowledge my supervisor Sean O'Leary for his guidance and support from the proposal stages of the project and throughout. I also wish to thank Dr. Susan McKeever for her support in the Machine Learning area, her advice, interest in my work and encouragement has been invaluable. Finally, big thanks to Jonathan McCarthy for his tireless work coordinating our final year project module.

Table of Contents

Table of Figures

1. Introduction

1.1. Project Background

The Russo-Ukrainian war has been ongoing between Russia and Ukraine since 2014 with a major escalation on the 24th of February 2022 when Russia launched a full-scale invasion into Ukraine. Ukrainian government considered only 7% of Ukraine's territory to be under occupation in 2019, whereas by September 2022 this number increased to 18%. Even though Armed Forces of Ukraine have reclaimed 54% of land during several counter-offensives, many people still reside in the occupied villages and cities (Scott Reinhard 2022). The purpose of this project will be to allow loyal citizens to assist the Armed Forces, the Main Directorate of Intelligence and the Security Service of Ukraine in gathering information regarding the movements of enemy military columns and supply convoys.

As a response to the invasion, many IT specialists from Ukraine and around the world assisted the Ministry of Digital Transformation of Ukraine to develop various applications and tools. Among those is e*Enemy* - a chatbot in Telegram for citizens to submit images or videos of Russian troop movements (Den Prystai 2022). Even though this solution proved to be useful, it has drawbacks: validation of the information received has to be done manually and thus the time between obtaining the data and processing it in the corresponding department. This is the problem the project aims to resolve: enhancing the quality of data obtained, enforcing security, and improving user experience.

1.2. Project Description

This project is going to be a cross-platform mobile application built with Flutter that will be available both for iOS and Android devices to allow for as large user base as possible. Image validation is a crucial feature in this project and it will be based on usage of artificial intelligence (AI) trained on a custom dataset. This project will be using the Convolutional Neural Network (CNN) model to perform image classification: recognize armoured fighting vehicles (AFV) within the image.

To report enemy vehicles, users only need to take a photo within the app, answer a couple of short questions and write an optional comment. The application validates the image and uploads it alongside its associated meta data to the cloud database in Firebase. In case there is no Internet at the moment of report creation, the report is saved locally allowing users to upload it later. There is an

award system for reports that were manually verified by humans. Also, users are able to see the numbers of the Russian Armed Forces and the so-called DPR/LPR Militia losses within the application in case they are cut off from the news outlets and media in Ukraine.

1.3. Project Objectives

To deliver the final product described above, there will be the following milestones along the way:

- Implementation of Core Features and Firebase Setup.
- Preparation of a dataset of Armoured Fighting Vehicles.
- Training and Deployment of Image Classification Models.
- Implementation of Secondary Features.
- Testing and Deployment.

1.4. Project Scope

The scope of this project is limited to developing a mobile application only. This project does not provide a platform for military and intelligence personnel to view and process reports, because developing such a platform would probably require the same amount of time and resources as mobile application. Although this project comes with Firebase Console, it is intended to be used for data, user management and application support purposes only.

This project is not intended to be used as a complete replacement to human personnel verifying the data since it is quite difficult to achieve the best performance on distinguishing very similar, but still different models of the same type of vehicle with a relatively small dataset publicly available, but it should significantly reduce their workload.

Moreover, image classification is to be performed only after the photo is taken, currently there are no plans to identify military vehicles from videos or live feed over performance concerns as many devices may not have enough computational power for that.

Furthermore, this project is not intended to be used to report air targets such as planes, helicopters, drones and ballistic missiles because it requires a different approach due to their fast moving nature.

1.5. Thesis Roadmap

The rest of the paper is structured as follows. Section 2 discusses similar existing solutions to the problem that the project aims to resolve, available technologies for the project implementation, published works on the field of recognition of military equipment, related past final year projects. Section 3 provides an overview of several modern software development methodologies, explains the scientific approach to the research part of the project, describes system operation accompanied by various diagrams and user interface concepts that were created during designing stages of the project. Section 4 provides a detailed overview of the development process and describes various technical aspects of a system. Section 5 outlines the testing and assessment strategy for the project. Section 6 draws a conclusion and outlines the future work to be potentially carried out on the project.

2. Research

2.1. Introduction

This chapter covers all the background research carried out for this project, comparison of the technologies and tools available for the implementation of this project and an overview of the existing or very similar solutions to the problem the project is aiming to resolve.

2.2. Existing Solutions

Many application were created to support the Armed Forces of Ukraine during the first months of the Russo-Ukrainian war, some of them being aimed at reporting various military equipment, both on ground and in the air. Additionally, owing to technological advancements in the AI area, militaries around the world became interested in incorporating AI in their vehicles and equipment. Reviewing existing solutions will give an understanding of what features need to be incorporated in the project and what should be avoided in order to create a successful wartime application.

2.2.1 Bachu

In March 2022 Security Service of Ukraine launched an application called *Bachu* for reporting the movement of enemy vehicles, equipment, soldiers, sabotage and reconnaissance groups. The application allows to send an image alongside other important information, such as current geolocation and a comment. In case there is no internet connection, the information is temporarily stored locally until the connection is available again ("Bachu" 2022). However, this application does not offer any client-side validation of information or user authentication, meaning that even enemy actors can use this application to upload false information.

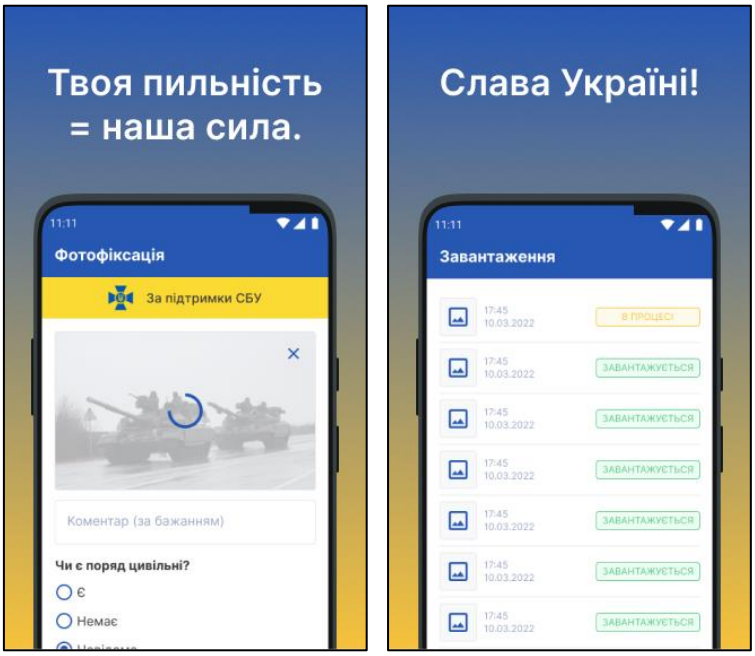


Figure 1: Screenshots of Bachu

2.2.2 iePPO

In October 2022 another war effort application called *iePPO* was released as a response to an increasing amount of air raids on Ukrainian cities and strategic infrastructure. It was developed in cooperation with Ukraine's Armed Forces. People can use this application to instantly report the location and the direction of incoming enemy missiles or other objects. The login is possible only for the Ukrainian citizens via their ID and a digital document application *Diia* ("IePPO" 2022).

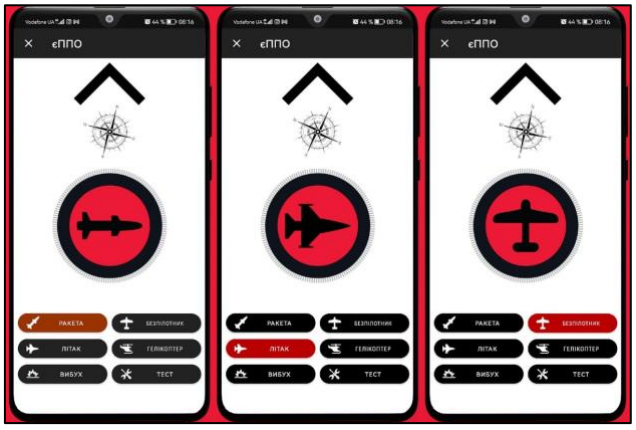


Figure 2: Screenshots of iePPO

2.2.3 ATLAS

In 2019 the US Army has started the Advanced Targeting and Lethality Automated System (ATLAS) program. It is designed to improve military technology through OCR, AI and ML. Its target recognition system will use machine learning algorithms trained on large datasets provided by the US Army and multiple sensors to help vehicle crews with identification of potential targets. It is also supposed to defeat camouflage and spoofing. ATLAS has not been brought into active military service yet, but the first demonstration will be on a 50 mm XM913 autocannon which is planned to be installed in a future Optionally Manned Fighting Vehicle (OMFV) (Sydney J. Freedberg Jr. 2019).

2.3. Technologies Researched & Selection

Several possible technologies may be considered for the development of the application. These include cross-platform frameworks, databases and deep learning libraries and models. A selection of technologies will be made based on the requirements of the application.

2.3.1 Cross-Platform Frameworks

It is essential that a wartime application is made available for as many people as possible, hence developing a native application for this project was not an option due to the need to maintain two different codebases, so native options, like Swift for iOS and Kotlin for Android, will not be discussed.

2.3.1.1 Flutter

Flutter is a framework developed by Google in 2017 and since then it has significantly gained in popularity. According to a 2021 global developer survey, around one third of mobile developers use cross-platform technologies, of which 42% use Flutter (Lionel Sujay Vailshery 2022). It is using Dart (optionally typed programming language) and provides SDK for developing cross-platform mobile applications. By using Flutter it is possible for the developers to take advantage of the native features of a mobile device, such as camera, file storage, GPS, Bluetooth, and various sensors, but Flutter

applications can also incorporate some native code snippets if necessary. In Flutter, graphics and animations are drawn on the screen in the real time using the Skia rendering engine and its compiled directly to C, which eliminates any performance bugs of the interpretation process. Another advantage of this framework is a hot reload feature that allows any changes to be observed at any time, which in turn makes development iterations faster, saving time and money. Currently there are more than 500 000 applications built with Flutter and due to the outstanding support from Google and the development community, new packages are constantly released (Michał Kochmański 2022).

2.3.1.2 React Native

React Native is a framework developed by Facebook and it based off React which is one of the most popular web development frameworks. According to a 2021 global developer survey, around 38% of mobile cross-platform developers use React Native (Lionel Sujay Vailshery 2022). It is implemented with JavaScript (dynamically typed programming language), so it managed to gain an extensive support of the development community with many libraries available. However, React Native relies on JavaScript Bridges for communication with the native code, negatively impacting the performance of the application. Recent versions of React Native addressed this with JavaScript Interface (JSI) allowing developers to directly invoke the native module without using bridges. Moreover, just like Flutter, React Native supports hot reload, making development more efficient (Andrew Baisden 2022).

2.3.1.3 Xamarin

Xamarin is a framework developed by Microsoft in 2011 with UI implemented using XAML and C#. This framework is using native UI elements, so a single shared codebase can look different across Android and iOS devices. This can be viewed both as an advantage, allowing for an application to look more native, and as a downside, because for many applications it is vitally important to keep the consistent look across all platforms, so it is necessary for developers to create platform-specific custom renderers that increase production time and defeat the purpose of an application being cross-platform.

In 2020 Microsoft addressed this issue with .NET Cross-Platform Application Interface (MAUI), and evolution of Xamarin, but many developers have already moved away from this framework to other technologies (Michał Kochmański 2022). Moreover, even though it is in a General Availability status since May 2022, MAUI is still a rather new technology with many developers encountering plenty of

bugs, crashes, issues and unstable performance and claiming that it is not really suitable for production (David Ramel 2022). According to a 2021 global developer survey, just 11% of mobile cross-platform developers use Xamarin (Lionel Sujay Vailshery 2022). Hence, it is highly likely that even smaller number of developers use MAUI. It is worth noting that, in view of the above, Xamarin is not very suitable for implementation of this project.

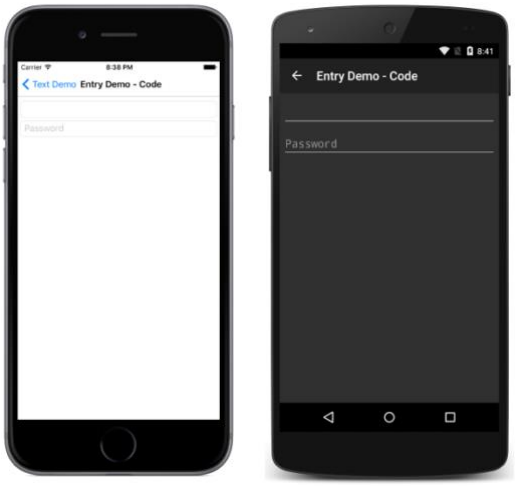


Figure 3: Xamarin UI – iOS and Android

2.3.2 App Development Platforms

2.3.2.1 Firebase

Firebase is a backend as a service platform that was released in 2012 and initially was only a real-time database. In 2014 it was acquired by Google and now Firebase offers many other products, such as cloud storage, authentication, machine learning, crashlytics and performance monitoring. Currently, there are more than 3 million applications that actively use Firebase and recently (Tyler Crowe 2022). Its core features are divided into three parts: Build, Release & Monitor, and Engage, perfectly suiting the needs of those that want a single platform for the entire development process. Firebase is designed to scale well and many of its functionalities are free. Moreover, due to the NoSQL nature of Firebase, database management is very flexible and convenient. Even though Firebase imposes certain limitations, it only matters for enterprise-level projects, for example, when there is a need to perform complex querying, or to integrate with microservices (Bogdan Guriev 2021).

2.3.2.2 Supabase

Supabase is a very similar alternative to Firebase. It is free, because it is an open-source product. Supabase offers a database and authentication service, storage and serverless functions for back-end logic. As for the database, it is using a PostgreSQL relational database and it provides instant APIs to handle CRUD operations in a relatively straight-forward way to avoid writing repetitive code. Also, it is quite easy to migrate data from another platform to Supabase. Unfortunately, Supabase is at a very stage so far and the community support is still rather limited (Colin Contreary 2022).

2.3.2.3 Heroku

Heroku is a platform and a service that enables developers to deploy and manage applications in the cloud. Heroku offers CI/CD, app metrics, code and data rollback, smart containers, data clips and Heroku Redis. Just like Supabase, Heroku is using PostgreSQL database. Heroku is known for very simple pipelines from development to production, it is only necessary to link the repository from GitHub to Heroku application and configure Heroku to properly run the application. However, Heroku is more suitable for web applications rather than mobile applications. Furthermore, it is more suited for medium to large enterprise customers due to relatively high costs (Krunal Lathiya 2022).

2.3.3 Object Detection Algorithms

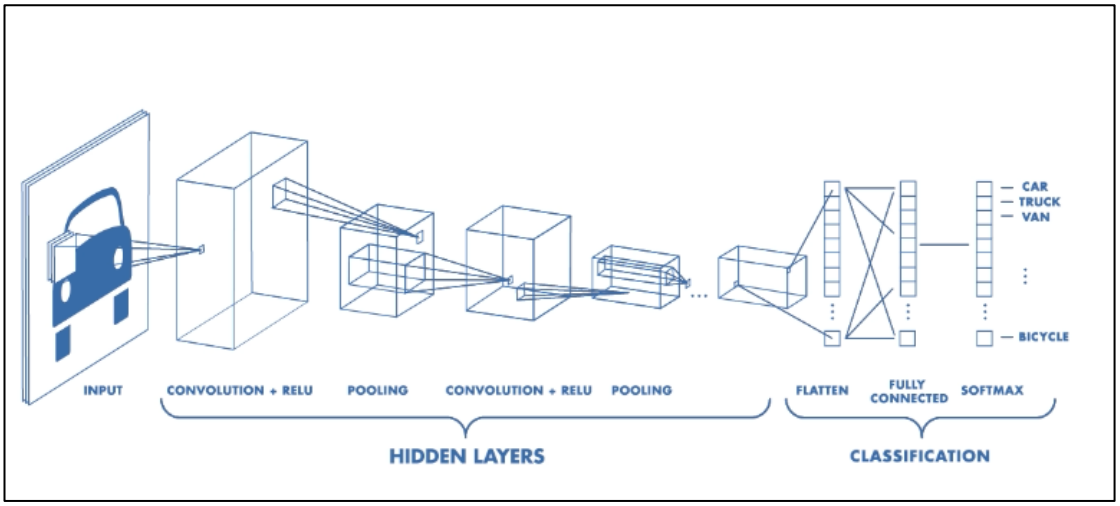


Figure 4: Classic CNN Architecture (Matthew Stewart 2019)

Currently, convolutional neural networks (CNN) is an industry-standard approach to object detection and object classification. CNN is using filters to analyse the influence of nearby pixels and moves them across the image from top left to bottom right. After the filters have passed over the image, a feature map is generated for each filter. These are then taken through an activation function, which decides whether a certain feature is present at a given location in the image. CNN consists of many layers, usually the first layers learn basic feature detection filters (edges, corners), the middle layers learn filters that detect parts of objects and the last layers recognise full objects (Matthew Stewart 2019).

To implement the object detection system, three popular convolutional neural network models are considered: YOLO (You Only Look Twice) and SSD (Single Shot Detector) and Mask R-CNN. The latter is using Region Proposal Network (RPN) to propose regions of interest from features and is used for instance segmentation, whereas YOLO and SSD are using detection without proposals, making them fast enough to even perform live detection, but their accuracy is worse (Harshit Kumar 2019). Due to the fact that military vehicles are painted in such a way to blend with the environment and that certain types of vehicles might look too similar, higher accuracy is prioritised over speed, making Mask R-CNN the best option. However, the final choice will be made during the development phase.

2.3.4 Deep Learning Frameworks

An open-source framework TensorFlow is one of the most commonly used frameworks and, like Flutter, developed by Google. It provides extensive, flexible features, an exhaustive library for programming, classifications, regression models, neural networks, including a suite to write algorithms for software. TensorFlow allows to train and deploy models almost on any platform. However, this framework is known to have a rather high learning curve. It is also possible to use Keras, which is a library that runs on top of TensorFlow and provides only very simple and consistent high-level interfaces, whereas TensorFlow provides both high-level and low-level interfaces. Keras also provides clear feedback for user errors (Project Pro 2022).

However, there is also another approach to deep learning that enables developers with limited machine learning expertise to train high-quality models, also offered by Google: AutoML Vision, which takes care of many things, including preparing the training dataset, monitoring and tuning the training process and, at the output, provides an object detection system with a very good performance (Harshit Dwivedi 2020). It can be trained online and, either deployed locally as a TensorFlow Lite model for free, or online as an API for a certain charge (Firebase 2022).

2.3.5 Technology Selection

Overall, Flutter and React Native seem like the best options for implementing this project due to their large community support, performance and quality of user interface in comparison to Xamarin. However, Flutter is slightly more preferable due to even better performance than React Native and this project author's personal preference of an optionally typed over dynamically typed programming languages that allow detecting errors earlier and producing more rigid code. Another benefit of choosing Flutter over other frameworks is that coming from a Google product line-up, it has very good compatibility with Firebase that can provide hosted backend services such as real-time database, cloud storage and machine learning. Upon creating custom deep learning models in Keras and AutoML, it would be possible to deploy them on a device using TensorFlow Lite. In case the model is updated, there would be no need to download a new version of the application, but instead get a latest model from Firebase, thus preserving cellular data, which might be very limited in a warzone.

2.4. Studies Relating to Project Domain

The topic of recognising military vehicles seems to be quite under-researched, at least publicly. The only papers relevant to the topic of the project come from Finland and were published between the years of 2017 and 2022. All of them are using deep learning to recognise and classify military vehicles.

2.4.1 Recognizing Military Vehicles in Social Media Images Using Deep Learning

In their conference paper Hiippala makes extensive use of openly available libraries, models and data, keeping in mind organisations with fairly limited resources where manual labour for monitoring and analysing social media for open-source intelligence is not a viable option. Deep convolutional neural networks (CNN) are used for a wide range of computer vision tasks, including object recognition because they learn the necessary features automatically and have better performance than manually created human-engineered features, which were previously used for describing the content of images.

However, the task of recognising military vehicles is quite challenging owing to the fact that publicly available training data of military vehicles is rather limited and, in contrast to civilian vehicles that have plenty of individual characteristics and the differences are quite often clearly visible, different

military vehicle types tend to be very similar to each other. To overcome this challenge, two different approaches to the architecture have been explored: training a network from scratch using data augmentation to increase the volume of data, and training a network without data augmentation, but opting to use transfer learning, essentially using the features learned from some other dataset to describe the images in the current dataset.

It was necessary to create a dataset of the combat vehicles encountered in Ukraine and most used by both sides of the conflict: two classes of 1457 images of T-72 main battle tank and 1088 images of BMP infantry fighting vehicles, and a negative class of 1477 images of other objects to enable the AI to learn about other kinds of photographic material posted from the conflict zone.

After conducting the experiments Hiippala discovered that residual neural networks significantly outperform convolutional neural networks at all levels of accuracy achieving an average accuracy of 95.18%, while also generalise much better when using transfer learning (Tuomo Hiippala 2017).

2.4.2 Military Vehicle Recognition with Different Image Machine Learning Techniques

Legendre and Vankka continued studying differences between transfer learning and self-design convolutional neural networks approaches to classification of military vehicles.

They obtained training data from previous works and from social media featuring images of infantry fighting vehicles, armoured personnel carriers, main battle tanks, including smoke screen covered vehicles, foliage camouflaged vehicles, tanks firing, and other images from modern military conflicts.

For transfer learning, two different architectures were selected: ResNet50 and Xception. ResNet50 performance achieved validation accuracy between 81%-95%, whereas Xception achieved validation accuracy between 27%-64%. Image augmentation did not cause any significant changes on performance of both transfer learning algorithms.

Validation accuracy of architectures trained from scratch is between 62%-87% and it was noted that different architectures adapt in a more optimal manner for the different classification cases. Image augmentation increased validation accuracy for all cases by ≈10%.

Researchers also proposed to use 3D renderings of military vehicles for image augmentation purposes to potentially improve quality and variability of the data (Daniel Legendre and Jouko Vankka 2020).

2.4.3 Instance Segmentation and Classification of Armoured Fighting Vehicles

Rissanen et al. had a different approach from the above-mentioned papers: they created a two-phase deep learning prototype for detection and classification of military vehicles and extensively used synthetic images in the training data.

Researchers used a web scraper to automatically collect the real images of each of 18 models of AFV and check for duplicates, then manually removed false positive images and leftover duplicates, obtaining over 5500 unique images with vehicles in various combat and noncombat situations and environments. Due to scarcity of real images available for some models, Rissanen et al. created synthetic images for BMP-1, Leopard 2A4, T-14 Armata, T-90M using 3D modelling software Blender that also allowed them to make such parameters as ambient light, camouflage pattern, background environment, translation and rotation of a vehicle.

The first model created is the CenterMask2 instance segmentation model for detecting military vehicles in the images. The second model, based on EfficientNet-B7, is an image classification model that differentiates between the models of military vehicles. Its input are the images from the CenterMask2 network which are detected as AFV which the model then classifies.

Instance segmentation model was capable of detecting armoured fighting vehicles in plain sight with high precision and partly obstructed or camouflaged tanks with good to high precision. Introduction of synthetic images to the training dataset increased overall average precision by about 10%. Image classification model reached 84.5% accuracy with 0.983 validation loss. Rissanen et al. stated that this result is mediocre in comparison to other EfficientNet-B7 models and explained it by the fact that partly obstructed or camouflaged tanks are difficult images (Ville Rissanen et al. 2022).

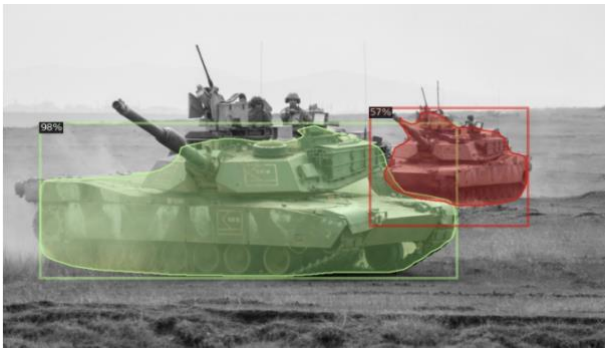


Figure 5: Detection of M1 Abrams (Ville Rissanen et al. 2022)

2.5. Existing Final Year Projects

2.5.1 Euro Coin Classification Using Image Processing & Machine Learning

This research-oriented project tackled the problem of euro coins classification using image processing and machine learning, while investigating different approaches of building models for object recognition. The project was implemented using OpenCV library with Python. Various computer vision techniques were used, such as Normalization, Thresholding, Gaussian Blur, Edge Detection and Hough Transform to aid the recognition process. As for the machine learning aspect of the project, Normal Bayesian and k-NN (k-Nearest Neighbours) methods were implemented, with former achieving 28.93% better performance than the latter. These are very simple and fast machine learning algorithms because they do not require any training time, but they also have some drawbacks: it is necessary to store the training data to produce new predictions. Also, they cannot learn the relationship between features because they assume that all features are independent or unrelated. Due to the research-oriented nature of the project, no proper deployment of the system was done, but only a very simple application using Cordova for demonstration purposes.

2.5.2 One Hour Only Machine Learning Smart App

This project is an Android mobile application that offers students the option to redeem deals offered by businesses within the food and beverages industry. The mobile application itself was developed using Java for Android and its web version is built with PHP and Angular. This application also incorporates machine learning to recommend specific deals to specific users taking into consideration their current location, more specifically, similarity-based learning algorithms. Just like the Euro Coin Classification, this project is using k-NN method of classification. To display deals based on the user's interaction with the system the Jaccard Similarity algorithm is used to looks at the user's latest purchased deal against other system users purchase trends within the system. However, due to the lack of users, author of the project found it quite difficult to carry out testing. Another issue was that there was not enough deal data to perform mining analyses.

2.6. Conclusions

During the research phase of the project, similar existing solutions were studied, including wartime mobile applications developed for users in Ukraine, with various functions being noted. It is critically important to make sure that bad actors would not be able to use the application to upload false data.

Furthermore, after researching available technologies, it was decided to go for the Google technology stack for this project (Flutter, Firebase, TensorFlow) to avoid any incompatibility issues and to make use of extensive documentation and community support.

Some findings were made after reading the relevant scientific papers, more specifically:

- In case of AFV recognition, data augmentation did not make any significant impact when performing transfer learning, unless training the convolutional neural network from scratch.
- Creating synthetic data is a rather effective way to increase accuracy. Within this project, it is potentially be a good idea to use screenshots from an online tank combat simulator *War Thunder* ("War Thunder" 2011), where most tanks currently deployed by Russia in Ukraine are very realistically depicted, as synthetic data.
- Usage of a two-phase system (CenterMask2 and EfficientNet-B7) for detection and classification provides very good accuracy. Just like CenterMask2, Mask R-CNN is also an instance segmentation model, but it also has to perform the classification, so it was be interesting to compare the performance of the two architectures.

Figure 6: Russian T-72 in War Thunder ("War Thunder" 2011)

3. System Design

3.1. Introduction

This chapter covers the description of modern software development and scientific methodologies implemented within this project as well as early designing stages of the project, including various diagrams and prototypes of the user interface.

3.2. Software Methodology

The methodology to be used throughout this project's lifecycle should make it easier to keep tasks on track, finish them by the deadline, and guarantee that deliverables are supplied for each milestone in a controlled and effective manner. For the sake of this project, Scrum, Kanban, and Feature Driven Development were among methodologies that were considered for implementation. All of them are based on an Agile approach, which divides project milestones into smaller tasks, thus allowing to continuously improve the project while preserving quality (Phuong Anh Nguyen 2021).

3.2.1 Scrum and Kanban

Scrum is a popular framework for organising work towards the development of complex products. Its principle is breaking down the work into 2-4 weeks long independent sprints with the team divided into three main roles: Product Owner, Scrum Master and Development Team. It incorporates a product backlog (product goal), sprint backlog (sprint goal) and tasks to be carried out during sprints. Scrum enforces communication and interaction among team members and makes daily meetings essential to track progress and identify problems during the development process. Planning meetings take place at the beginning and end of each sprint to discuss the overall progress and details of the tasks expected to be completed in the next sprint. Although Scrum is quite easy to use and speeds up the development process and generally very effective, it is not really suitable for this project as it usually works best for project with medium sized teams and relatively long duration months.

Kanban can be considered a light-weight version of Scrum, because this methodology only has backlog, so there is no need to organize work into sprints. Kanban has no time limit or repetitious procedure, it is flexible when it comes to implementing tasks, and relies on continuous delivery. The

Kanban board shows how tasks progress through different stages. This board can be divided into three categories: To Do, In Progress, and Done. This makes it simple to keep track of how the project and certain tasks are going. In case there are multiple people working on a project, tasks can be assigned to specific individuals, which saves team members from becoming overwhelmed with work. Kanban is a very good fit for smaller development teams or even a single individual because of its event-driven nature, making this approach a very good candidate for implementation in this project.

3.2.2 Feature Driven Development

Feature Driven Development (FDD) is a client-centric approach that emphasizes iterative and incremental development with the aim of delivering software that functions. It focuses on creating feature-specific systems by breaking up the development of system functions into a number of short phases. The team is divided into six primary roles: Project Manager, Chief Architect, Development Manager, Chief Programmer, Class Owners and Domain Expert. The methodology itself is divided into five basic phases: develop an overall model, build a features list, plan by feature, design by feature, build by feature. Breaking down big activities considerably lowers the risk involved with developing large-scale software projects and promotes efficiency. Even though FDD requires less meetings than Scrum, more documentation is necessary. However, just like Scrum, this is a relatively heavy-weight approach more suitable for large, scalable and complex projects, rather than small teams and individual developers, making this approach not suitable for this project (Mike Bentley 2021).

3.2.3 Methodology Selection

It was decided to use a hybrid of Kanban and Feature Driven Development methodologies for this project. Thus, such phases of Feature Driven Development as building an overall model and building a features list will be incorporated during the design process, whereas Kanban principles will be used during the development process due to the fact that it is flexible, very easy to use and does not impose any time constraints.

Figure 7 is a screenshot from a web-based tool called *Trello* that was used to visualize work. This Kanban Board has five columns: Planned, In Progress, Done, Future Plans and Scrapped. Since this screenshot was taken during the final stages of the project development, most cards are either in Done or Future Plans columns. Also, all cards have labels that depict which part of the project the task

relates to and coloured according to the their priorities, from red (top priority features, core functionality) to green (low priority features, secondary functionality). Kanban Board proved to be useful in weekly project supervisor meetings, allowing for clear understanding of progress up to date and easier planning of work for future.

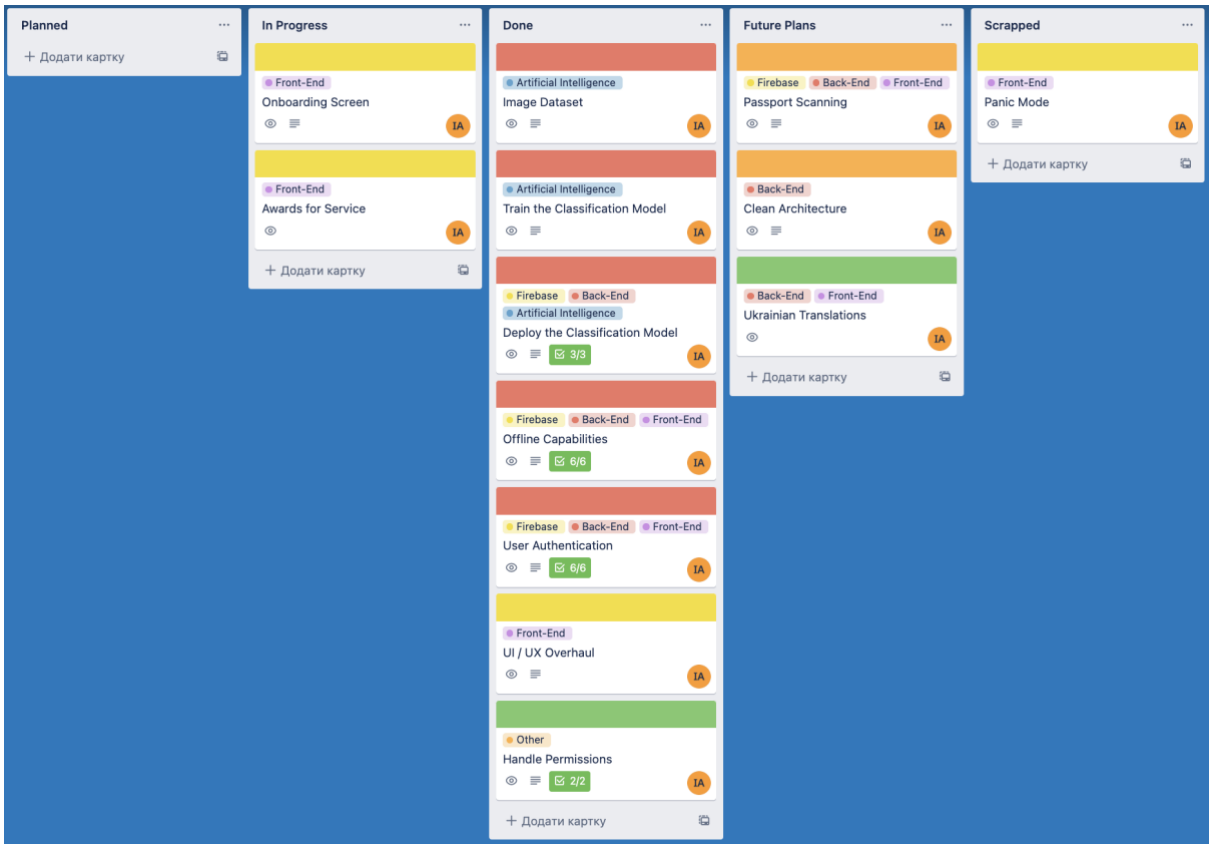


Figure 7: Kanban Board in Trello

3.3. Scientific Methodology

This project is partially research-oriented, so it is essential to discuss the steps involved in a scientific method. The project is utilising the Cross Industry Standard Process for Data Mining (CRISP-DM) which includes six sequential phases as demonstrated in Figure 8 (Nick Hotz 2022). The milestone that the research was attempting to achieve is a deep learning network capable of distinguishing between different models of armoured vehicles.

To train a network, a dataset is required. Initially, it consisted of images of captured and damaged military vehicles (Stijn Mitzer and Jakub Janovsky 2022), an existing dataset of images of T-72 and BMP

(Tuomo Hiippala 2017) and images obtained through web-scraping. Next, the dataset was expanded with synthetic data, screenshots from tank combat simulator *War Thunder* ("War Thunder" 2011).

After curating the dataset, the experiments took place: firstly, a model based on AutoML was trained on the dataset without synthetic data, then it was retrained using synthetic data and their results were compared. Even though originally it was also planned to train an instance segmentation model based on Mask R-CNN architecture, decision was made not to because of certain limitations that will be discussed in Section 4 and to allocate more resources to the app development part of this project.

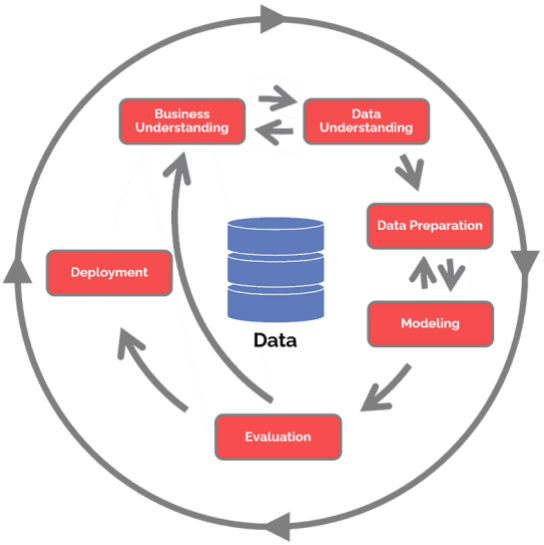


Figure 8: CRISP-DM Diagram (Nick Hotz 2022)

3.4. Overview of System

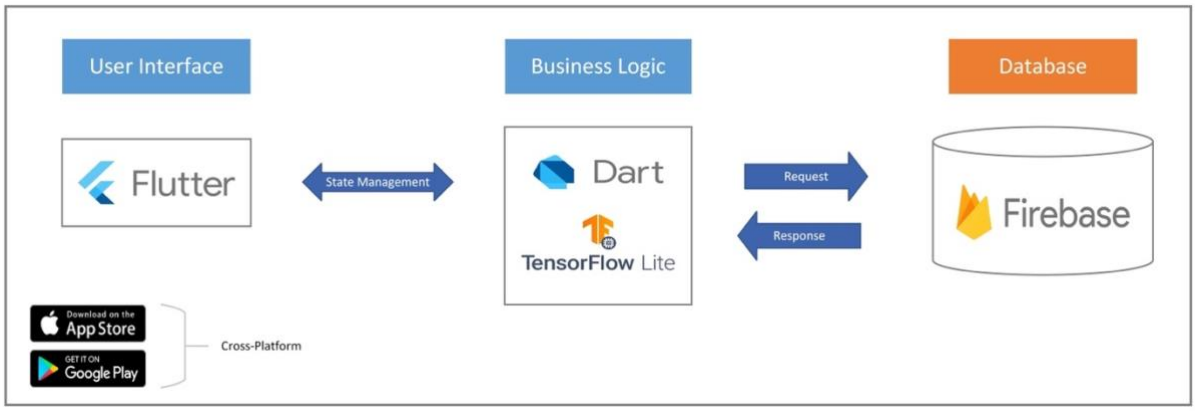


Figure 9: System Architecture Diagram

The cross-platform mobile application follows the system architecture shown in Figure 9 above. It provides quick and reliable way to report enemy military vehicles. Since the application is supposed to be used relatively close to the battlefield where there is next to no internet connection, there are no dedicated client-side (front-end) and server-side (back-end) parts in this project. All processing is happening on the device, internet connection is required only for user authentication, file uploads and downloading some non-mission-critical files and information from Firebase.

One of the aims for this project was trying to adhere to principles of Clean Architecture, as much as possible. And according to those principles, Business Logic should be decoupled from UI (Anton Klimenko 2020). Also, there are two NoSQL databases: Firebase and Hive. Firebase is the primary database, but it requires internet connection, so Hive is used to store reports locally and to cache API.

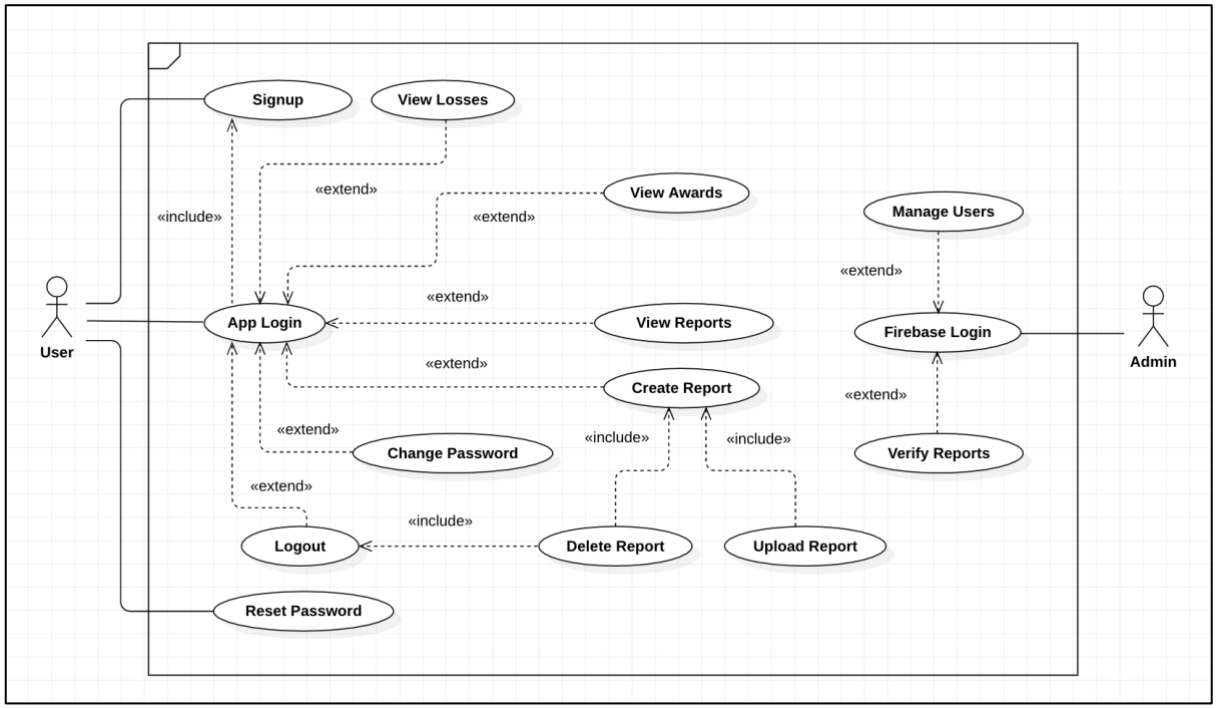


Figure 10: Use Case Diagram

The use case diagram above in Figure 10 describes the functionality interactions that the user can have with the application. It also illustrates how the admin has exclusive access to all user details and reports uploaded to the database.

When user opens the application for the first time, they are required to register. They need to fill in their details, such as first name, last name, passport number, email address and password. Upon registration, user profile is created in Firebase.

Originally, it was planned that user only needs a passport number and a password to login. However, Firebase Authentication requires either a phone number or an email address. An advantage of this approach is that in case user forgot their password and wanted to reset it, the password reset link will be sent to their registered email and they can change their password there.

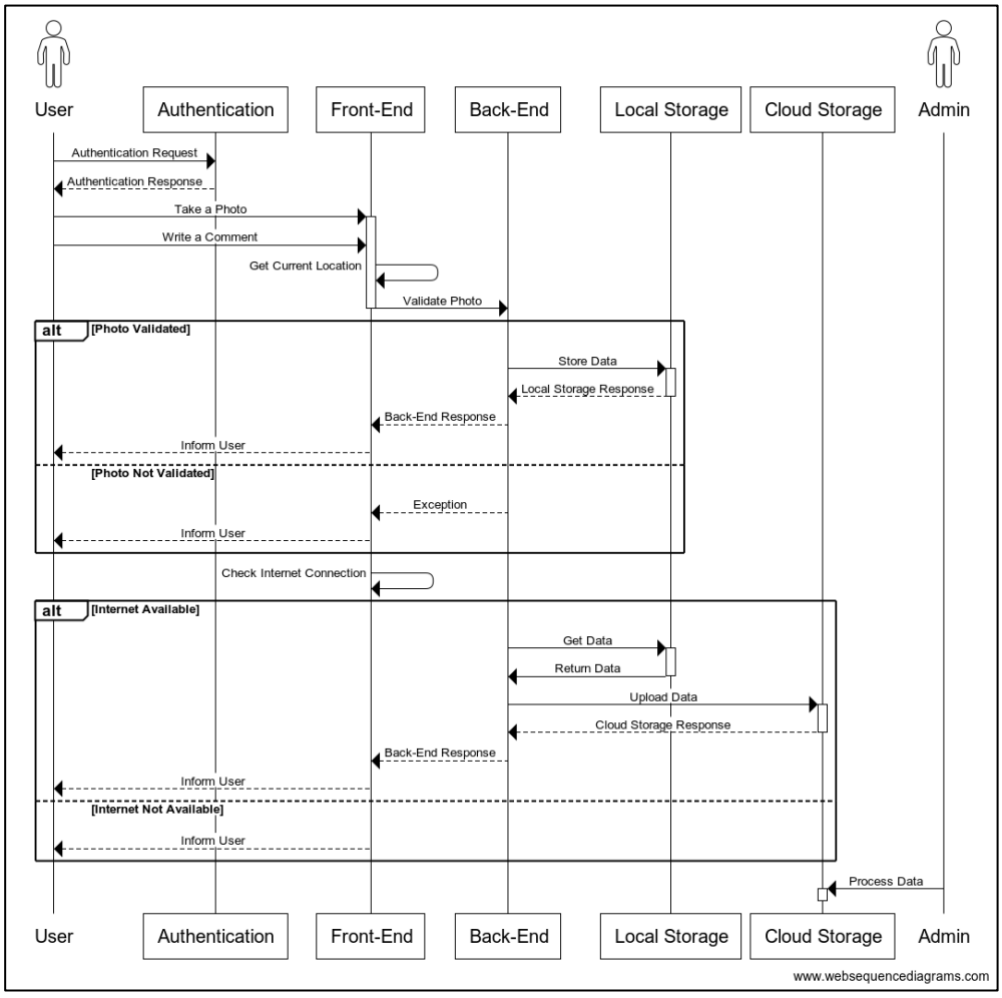


Figure 11: Sequence Diagram

The sequence diagram above in Figure 11 describes the process of creating and uploading the report the Firebase. Upon taking a photo and writing an optional comment, the application saves the current location and passes the photo to the image classification model. If at least one type of AFV is present

in the image, the report is ready to be uploaded. If internet is not available, the image is saved in local storage and the path to the image along with report details, including image classification model output is saved in Hive to allow user to upload the report later when they have reliable internet connection. Reports are stored for no more than 12 hours, because the situation on the frontline is changing rapidly and there is not much use from the old intelligence reports. If internet is available, image and report details are uploaded to the Firebase. User can view the list of pending reports on a separate page and, if internet is available, upload all pending reports. User can also delete a report from pending reports, for example, if they decided that there are similar reports from the same area.

There is an achievements system in a form of virtual medals to keep user's morale high: the more verified reports come from the user, the more prestigious medals and orders they get. This information can, for example, be used to issue real awards after the war is over. User can view their awards on a separate page within the application.

User can see the numbers of the Russian Armed Forces and the so-called DPR/LPR Militia losses within the application in case they are cut off from news outlets and media in Ukraine and it can also act as a morale boost. The app updates statistics through an API ("Russian Warship" 2022). The data from the API is cached, so that information remains accessible for some time after the last successful API request even when there is no internet connection.

In case user is forced to hand over their device for a check by the enemy soldiers, it is recommended that they delete the application from their device, or at the very least, logout from the application. When they logout, all pending reports are deleted, removing traces of any spy activity.

Firebase Admins are able to view to user profiles, their reports and awards through Firebase dashboard. Incoming reports must be manually verified and, if a report turned out to be false or there is simply not enough information provided, it will be deleted. If a user is constantly uploading false information, it is possible to disable their account.

The class diagram below in Figure 12 illustrates the connections between the various classes in the application. The creation of a class diagram gives developers an excellent overview of the needed system and helps them get ready for the work that will be involved when implementing such system.

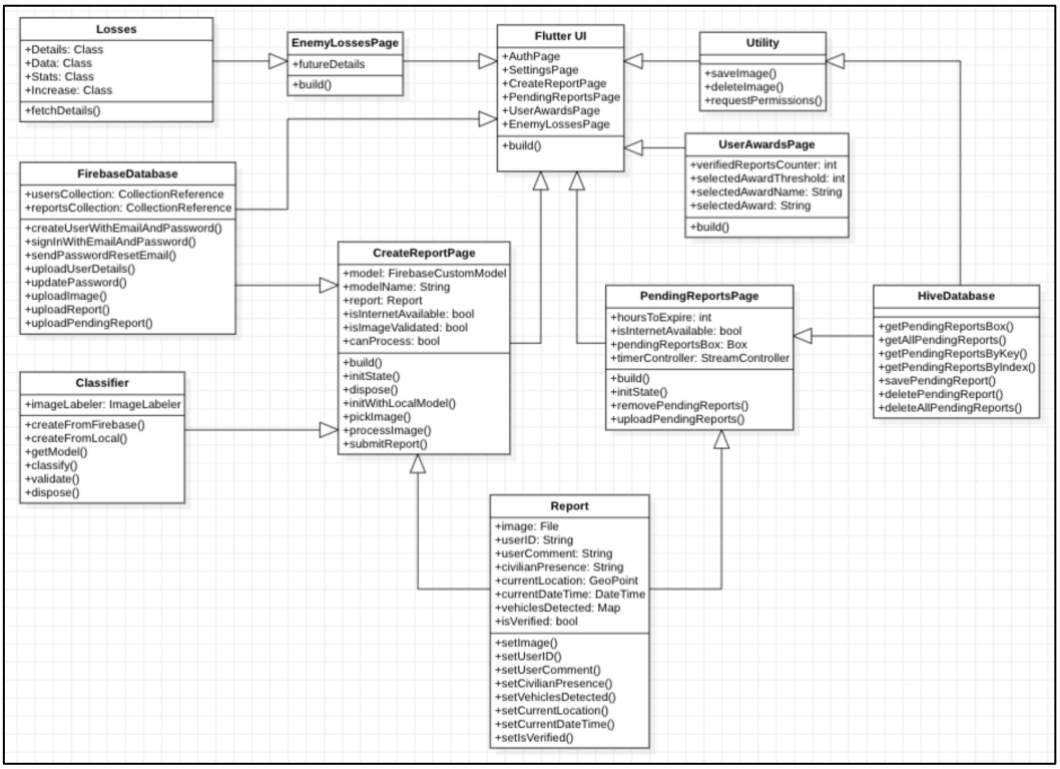


Figure 12: Class Diagram

The database diagram below in Figure 13 shows a structure of user and report documents that will be stored in a Firebase Database. Because the application essentially fulfils one single purpose, there is no need to store anything else in the cloud.

User Details	Report
{ "_id": id, "firstName": string, "lastName": string, "verifiedReports": int, "passport_number": string }	{ "_id": id, "userID": id, "image": string, "comment": string, "civilians": string, "location": point, "time": timestamp, "isVerified": boolean, "vehicles": [{ "model": string, "confidence": double }] }

Figure 13: Database Diagram

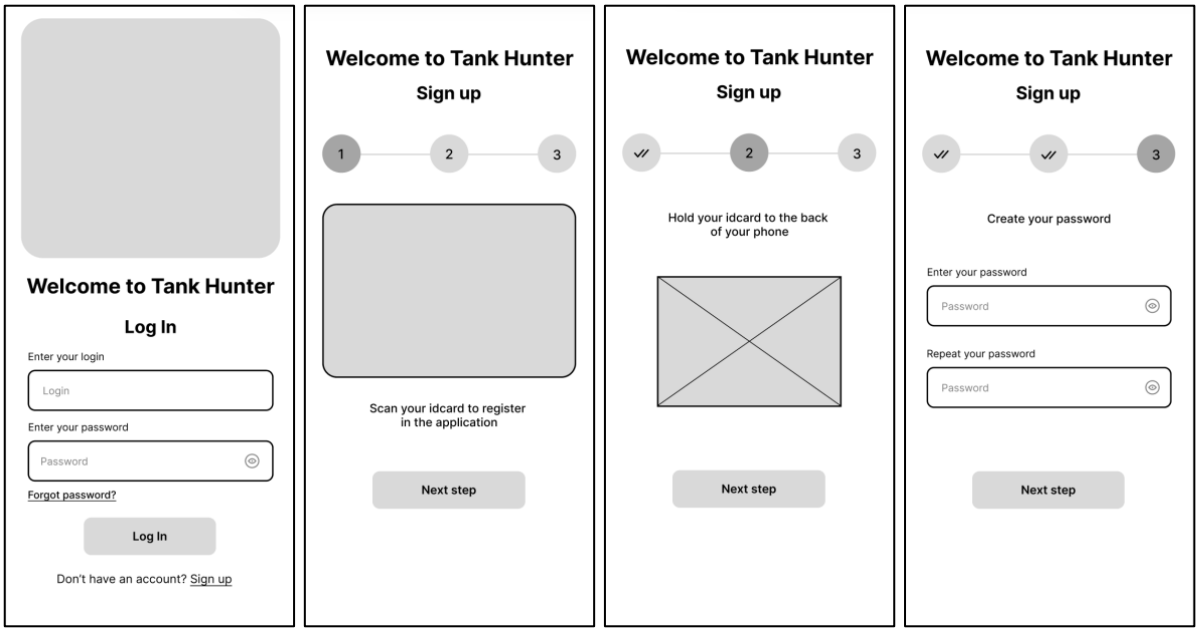
3.5. UI/UX Design

During the prototyping stage, some low-fidelity wireframes were created to help visualise the use cases and how certain features might behave. Moreover, it was possible to show wireframes to stakeholders even before work on prototypes is started. As a result, it was possible to gather some valuable feedback and suggestions at a very early stage of the project.

Wireframes in Figure 15 illustrate login screen and registration process, whereas Figure 16 shows the rest of the features of that were planned to be implemented in the application. Wireframes are similar to blueprints in architecture, they are intentionally avoiding any styling to focus viewer's attention on structure, order, and organisation of content.

After showing low-fidelity wireframes and an early prototype of *Tank Hunter* to the stakeholders, some initial feedback was collected. Decision was made to make the final design of the application similar to *Diia*, a mobile application that allows Ukrainian citizens to use digital documents and access over 50 governmental services ("Diia" 2023). Since many Ukrainian users are already used to this application, it would be easier for them to use *Tank Hunter* if it looked and felt the same way. Moreover, if the Ukrainian government eventually becomes interested in *Tank Hunter*, minimal changes would be required to make it fit in with the rest of e-government applications. Also, some UI elements are using yellow colour - this is a throwback to the "Yellow Ribbon", a resistance movement in the occupied territories of Ukraine, created in April 2022.

Figure 14: Screenshots of Diia



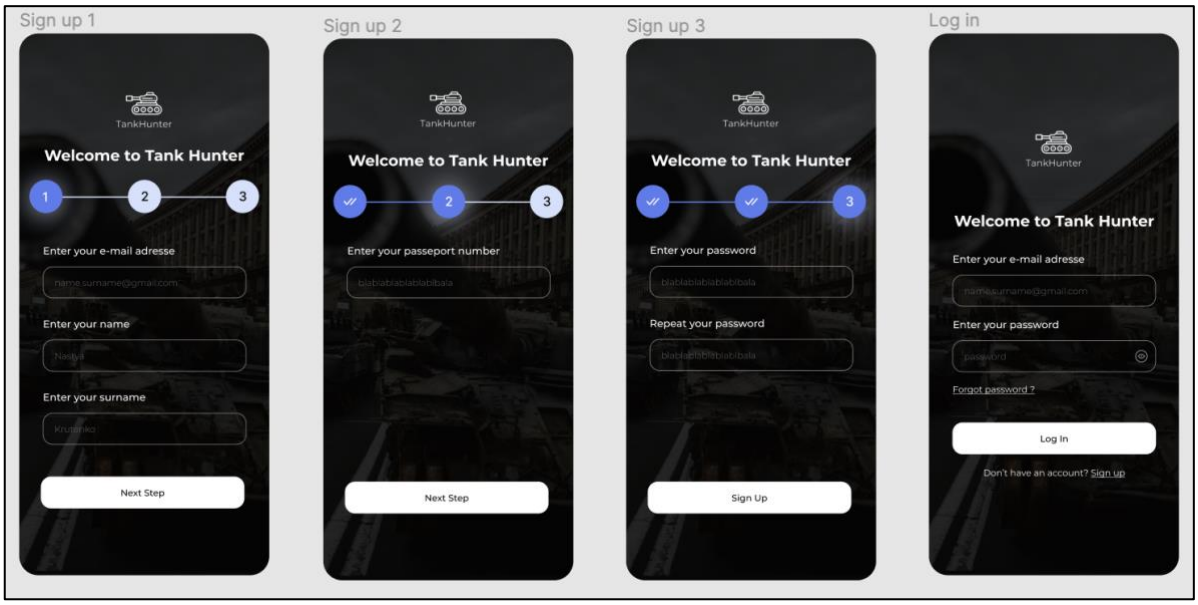


Figure 15: Authentication Wireframes

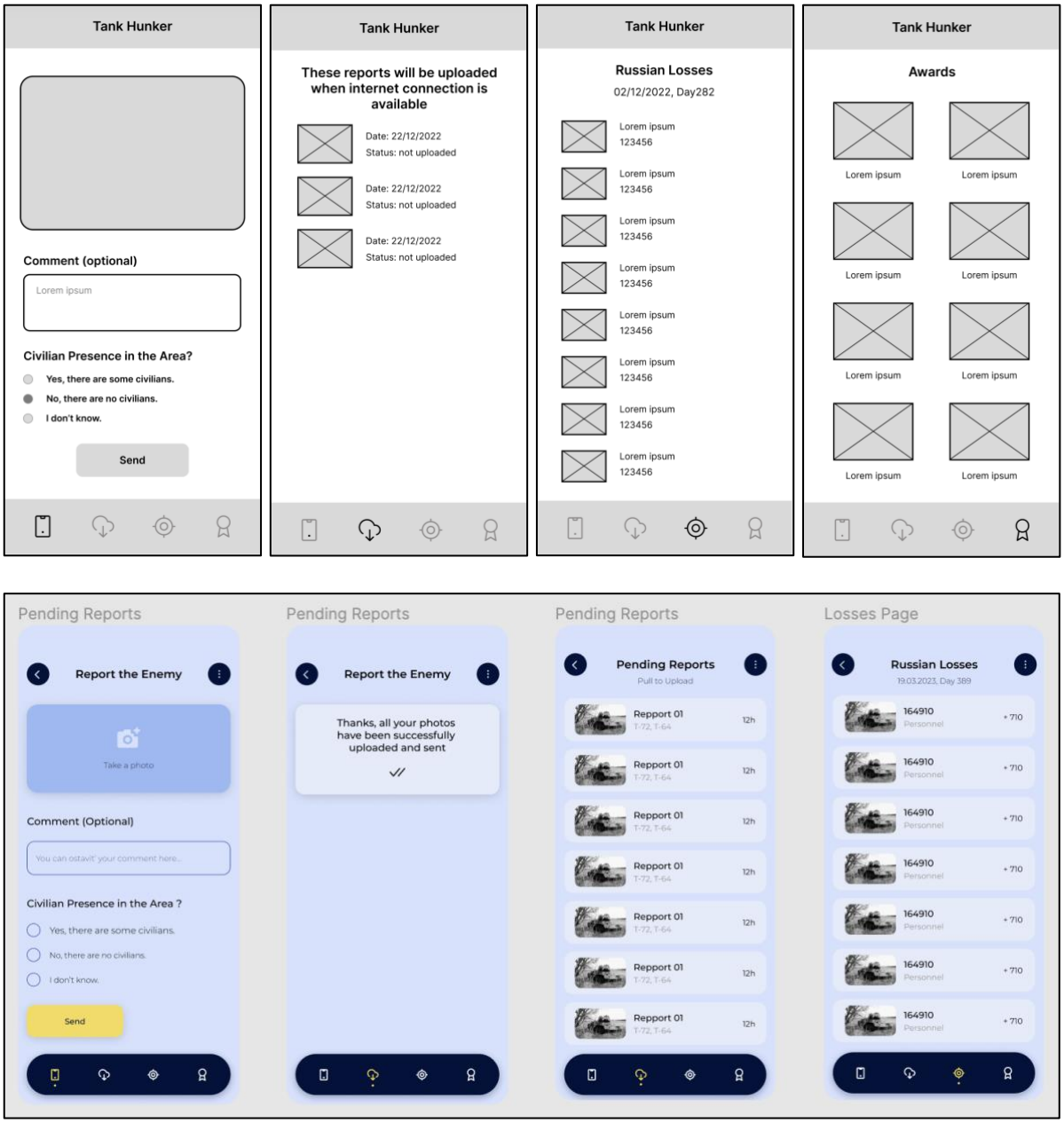


Figure 16: Core Features Wireframes

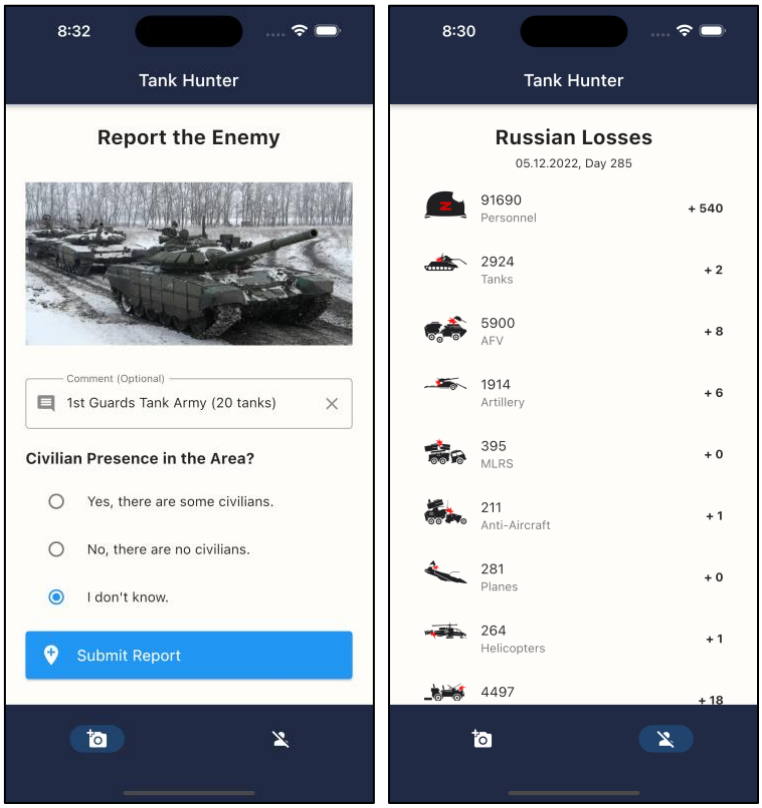


Figure 17: Tank Hunter Prototype

3.6. Conclusions

In conclusion, this chapter has successfully established a comprehensive methodology for developing our system. We have provided a clear overview of the system, including its architecture diagram, use case diagram, sequence diagram, class diagram and database diagram. Additionally, we have created both low-fidelity and high-fidelity wireframes to ensure the user interface is intuitive and easy to use. Our design choices and attention to detail will ensure the successful implementation of our system.

4. Application Development

4.1. Introduction

This chapter covers the description of the early development process of this application that took place after a substantial research carried out in Section 2 and according to the design documentation outlined in Section 3 of this report. The main focus of this stage of the project involved further implementation of core functionality, training and deployment of an image classification model deployment, introduction of secondary features and UI/UX overhaul.

4.2. Application Development

The application was developed with Flutter 3.9 and Dart 3.0.0 using Android Studio IDE. During the development of the application, state management was a crucial consideration. One approach to state management in Flutter is BLoC (Business Logic Components) pattern, which is a powerful architectural pattern that separates the presentation layer from the business logic and state management.

The BLoC pattern provides benefits such as improved code organisation, testability, and scalability, making it a popular choice for large and complex applications. Compared to the setState method, which is commonly used in smaller and simpler applications, the BLoC pattern offers a more structured and modular approach to state management.

However, after careful consideration of the size and complexity of *Tank Hunter*, it was determined that using BLoC pattern was not necessary. While we recognised the benefits of the BLoC pattern, implementing it would have added unnecessary complexity to our application. In addition, given our level of experience with Flutter at the time, it would have been challenging to implement the BLoC pattern effectively, if at all, given a rather limited timeframe.

Instead, we opted to use the setState method for managing state in our application. While this approach is less powerful than the BLoC pattern, it was sufficient for our needs. We also made an effort to keep the application architecture organised by separating the domain, data and presentation layers. *Presentation Layer* consists of the UI and the event handlers of the UI; *Domain Layer* contains the code for business logic, such as entities and use cases; *Data Layer* is responsible for data retrieval from API and databases. While we did not strictly follow the Clean Architecture approach, we believe that this separation of concerns will make the application more maintainable in the long run.

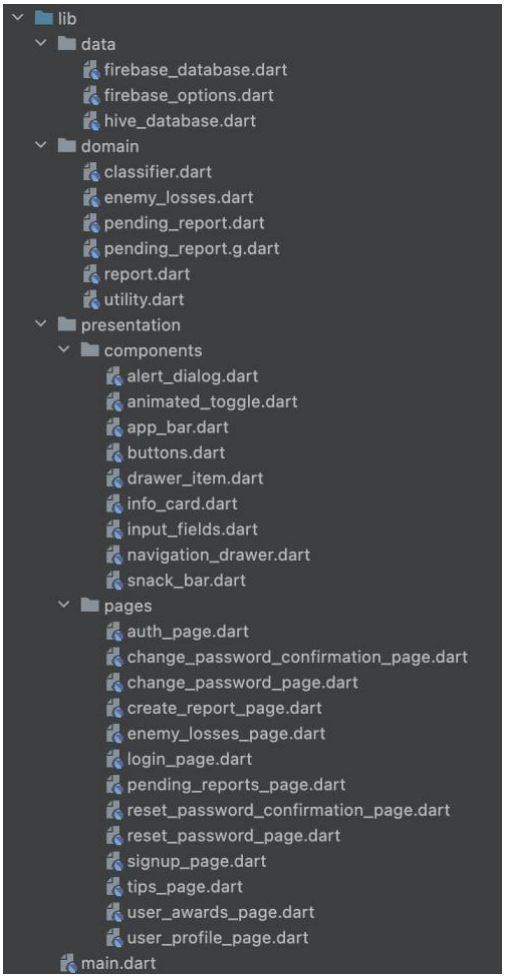


Figure 18: Project Structure

4.2.1 Presentation Layer

As the User Authentication pages and some other pages in *Tank Hunter* may not possess intricate functionalities or distinctive features, it may not be necessary to provide an exhaustive description of these components in this report (see Appendix for screenshots). Instead, the focus will be directed towards the pages that are responsible for the primary functionality of the app, which will be discussed in greater depth. This approach will enable the highlighting of the application's unique attributes.

Currently the application has four main pages, a settings page, a page with some useful tips and a few authentication-related pages. Pages in Flutter are built with widgets, similarly to components in React.

Widgets can display something, they can define design, and they can also handle interactions. They describe what their view should look like given their current configuration and state. When a state of

the widget changes, it rebuilds its description, which Flutter compares against the previous description to determine the minimal changes needed in the underlying render tree to update the widget state.

To start with, we will briefly discuss *main.dart* file, the entry point of the application. The code snippet in Figure 18 defines a main function that initializes Firebase and Hive and runs the runApp function to launch a *TankHunter* widget. The *TankHunter* widget builds a *MaterialApp* that contains a *StreamBuilder* widget that listens to changes in the user's authentication state using Firebase. If the user is authenticated, it displays the *RootPage* widget. Otherwise, it displays the *AuthPage* widget.

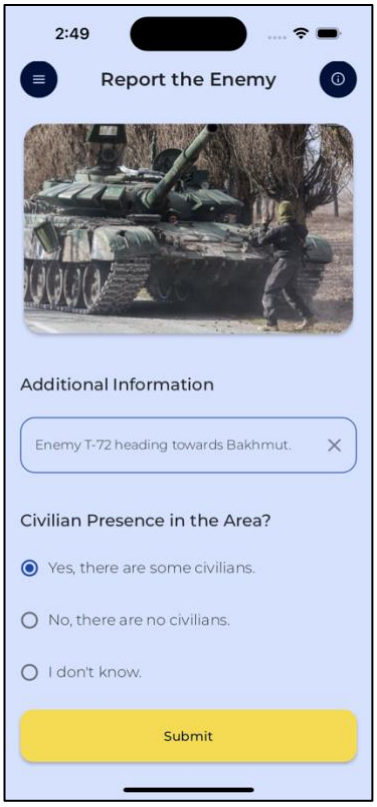
```dart
Future<void> main() async {
  WidgetsFlutterBinding.ensureInitialized();
  SystemChrome.setPreferredOrientations([DeviceOrientation.portraitUp]);
  await Firebase.initializeApp();
  await Hive.initFlutter();
  Hive.registerAdapter(PendingReportAdapter());
  await Hive.openBox<PendingReport>('pending_reports');
  runApp(TankHunter());
}

class TankHunter extends StatelessWidget {
  TankHunter({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    final textTheme = Theme.of(context).textTheme;

    return MaterialApp(
      debugShowCheckedModeBanner: false,
      theme: ThemeData(...),  // ThemeData
      home: StreamBuilder<User?>(
        stream: FirebaseAuth.instance.authStateChanges(),
        builder: (context, snapshot) {
          if (snapshot.hasError) {
            debugPrint('You Have an Error! ${snapshot.error.toString()}');
            return const Text('Something Went Wrong!');
          } else if (snapshot.hasData) {
            return const RootPage();
          } else {
            return const AuthPage();
          }
        },
      ),  // StreamBuilder
    );  // MaterialApp
  }
}
```

Figure 19: Code Snippet of main.dart

Report the Enemy page represents the most important part of this project and allows users to report enemy vehicles. The page has a button to take an image from the camera. After picking an image, the application checks if it has AFV in it. If there are no military vehicles, an error is displayed, and if there are military vehicles, the application records the user's comment, the location, and the detected vehicles. The report is then uploaded to a Firebase database, or if there is no internet connection, it is saved as a pending report in Hive local database.

The *initState* method is called when the widget is first created, and, on Android it initializes the local TensorFlow Lite model, whereas on iOS it downloads the Firebase model. This was a last moment fix due to the fact that development and testing of the project happened primarily on iOS devices and this bug was not detected until the last moment. The *dispose* method is called when the widget is removed from the tree, and it disposes of the instances of *Classifier* and input controllers.

There also several private methods on this page, including *_initWithLocalModel,* which initializes the classification model, *_pickImage,* which prompts the user to pick an image and validates if has military vehicles, and *_submitReport* which uploads the report to Firebase or saves it locally in Hive.




```
Future _submitReport() async {
  if (report.image == null) {
    ScaffoldMessenger.of(context).showSnackBar(buildSnackBar(messageText: 'Error: Upload an Image', isError: true));
    return;
  } else if (!isImageValidated) {
    ScaffoldMessenger.of(context)
        .showSnackBar(buildSnackBar(messageText: 'Error: Military Vehicles not Found', isError: true));
    return;
  }

  showDialog(
      barrierDismissible: false,
      context: context,
      builder: (context) {
        return Center(child: LoadingAnimationWidget.hexagonDots(color: const Color(0xff0037C3), size: 50));
      });

  report.setUserComment(commentController.text);
  report.setCivilianPresence(civilianPresence);

  String resultMessage = 'Thank You for Your Service!';

  if (isInternetAvailable) {
    await FirebaseDatabase.uploadReport(report).timeout(const Duration(seconds: 10), onTimeout: () async {
      resultMessage = 'Report added to Pending Reports';
      File image = await Utility.saveImage(report.image!);
      HiveDatabase.savePendingReport(report, image);
    });
  } else {
    resultMessage = 'Report added to Pending Reports';
    File image = await Utility.saveImage(report.image!);
    HiveDatabase.savePendingReport(report, image);
  }

  if (context.mounted) {
    Navigator.of(context).pop();
    ScaffoldMessenger.of(context).showSnackBar(buildSnackBar(messageText: resultMessage, isError: false));

    commentController.clear(); // remove any text from the comment input
    civilianPresence = CivilianPresence.unknown; // reset the civilian presence radio buttons

    report = Report();
    setState(() {});
  }
}
```

Figure 20: Report the Enemy Page

```dart
Future _initWithLocalModel() async {
  if (Platform.isAndroid) {
    modelName = 'assets/ml/vertex.tflite';
    Classifier.createFromLocal(assetPath: modelName, confidenceThreshold: 0.20, maxCount: 3);
  } else {
    model = await FirebaseModelDownloader.instance
        .getModel(modelName, FirebaseModelDownloadType.localModelUpdateInBackground);
    Classifier.createFromFirebase(assetPath: model?.file.path, confidenceThreshold: 0.20, maxCount: 3);
  }
  canProcess = true;
  setState(() {});
}

Future _pickImage(ImageSource source) async {
  bool isPermitGranted = await Utility.requestPermissions();
  bool isServiceEnabled = await Geolocator.isLocationServiceEnabled();

  if (isPermitGranted && isServiceEnabled) {
    if (canProcess) {
      await _processImage(source);
    } else {
      if (context.mounted) {
        ScaffoldMessenger.of(context).showSnackBar(buildSnackBar(messageText: 'Error: Unknown Error', isError: true));
      }
    }
  }
}
```

```dart
Future _processImage(ImageSource source) async {
  await report.setImage(source); // pick image from camera or gallery and save it in report

  if (report.image != null) {
    setState(() {}); // update UI, set a preview image
    vehiclesDetected = await Classifier.classify(report.image!); // check for AFV presence

    // if no AFV detected
    if (vehiclesDetected!.isEmpty) {
      isImageValidated = false;
      if (context.mounted) {
        ScaffoldMessenger.of(context)
            .showSnackBar(buildSnackBar(messageText: 'Error: Military Vehicles not Found', isError: true));
      }
      // if at least a single AFV was detected
    } else {
      isImageValidated = true;
      debugPrint(vehiclesDetected.toString());
      report.setVehiclesDetected(vehiclesDetected!); // save the map with AFV detected
      await report.setCurrentLocation(); // set current location
      await report.setCurrentDateTime(); // set current time
    }
  } else {
    debugPrint('Image Not Selected');
    return;
  }
}
```

Figure 21: Report the Enemy Page

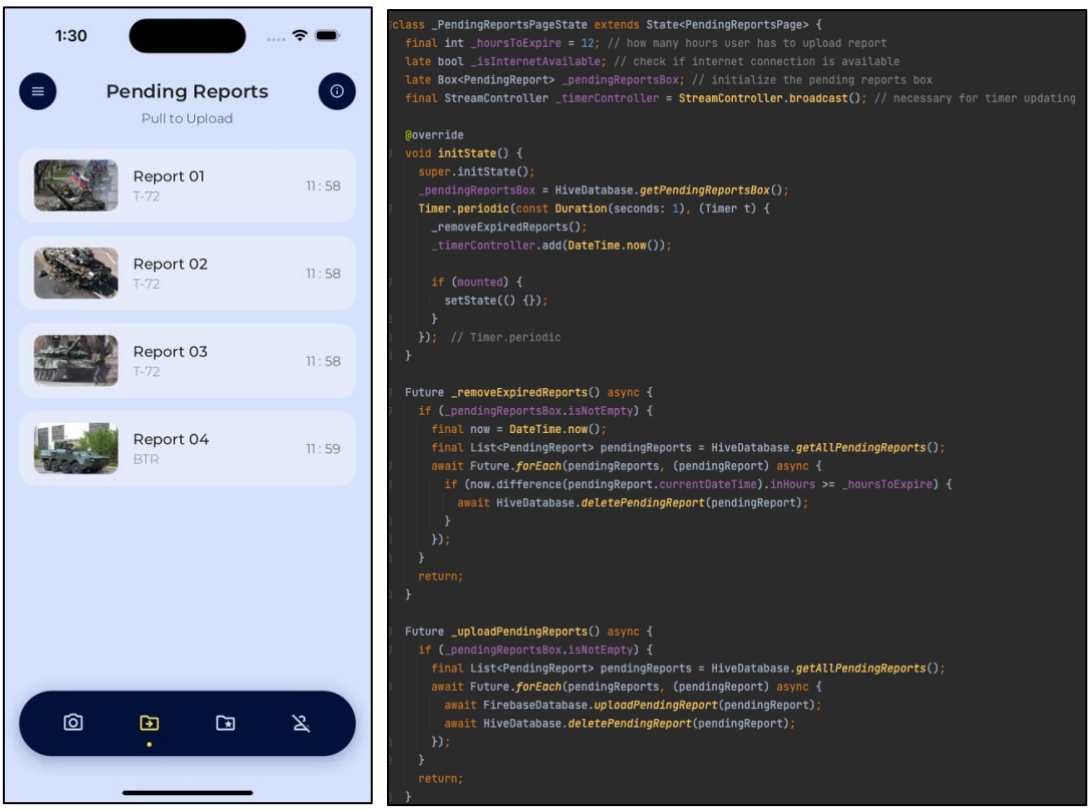
Pending Reports page is displaying the locally stored reports and the timer until they are expired. User can either upload all pending reports to the remote database or swipe the report left to delete it. The

initState method initializes *_pendingReportsBox* and sets up a periodic timer to check for expired reports and updates the UI by calling the *setState* method.

The *_removeExpiredReports* method is called by the periodic timer to remove any reports that have been pending for more than 12 hours. It retrieves all pending reports from the *_pendingReportsBox*, checks their timestamp, and deletes any that have expired.

The *_uploadPendingReports* method is called when the user pulls down to refresh the list of pending reports. It retrieves all pending reports from the *_pendingReportsBox*, uploads them to a remote database using *FirebaseDatabase* class, and then deletes them from the local *_pendingReportsBox*.

ValueListenableBuilder is a widget responsible for displaying a list of pending reports and it is actively listening for changes in the *_pendingReportsBox* and rebuilds the widget tree, if necessary.




```dart
class _PendingReportsPageState extends State<PendingReportsPage> {
  final int _hoursToExpire = 12; // how many hours user has to upload report
  late bool _isInternetAvailable; // check if internet connection is available
  late Box<PendingReport> _pendingReportsBox; // initialize the pending reports box
  final StreamController _timerController = StreamController.broadcast(); // necessary for timer updating

  @override
  void initState() {
    super.initState();
    _pendingReportsBox = HiveDatabase.getPendingReportsBox();
    Timer.periodic(const Duration(seconds: 1), (Timer t) {
      _removeExpiredReports();
      _timerController.add(DateTime.now());

      if (mounted) {
        setState(() {});
      }
    });  // Timer.periodic
  }

  Future _removeExpiredReports() async {
    if (_pendingReportsBox.isNotEmpty) {
      final now = DateTime.now();
      final List<PendingReport> pendingReports = HiveDatabase.getAllPendingReports();
      await Future.forEach(pendingReports, (pendingReport) async {
        if (now.difference(pendingReport.currentDateTime).inHours >= _hoursToExpire) {
          await HiveDatabase.deletePendingReport(pendingReport);
        }
      });
    }
    return;
  }

  Future _uploadPendingReports() async {
    if (_pendingReportsBox.isNotEmpty) {
      final List<PendingReport> pendingReports = HiveDatabase.getAllPendingReports();
      await Future.forEach(pendingReports, (pendingReport) async {
        await FirebaseDatabase.uploadPendingReport(pendingReport);
        await HiveDatabase.deletePendingReport(pendingReport);
      });
    }
    return;
  }
```

Figure 22: Pending Reports Page

Awards for Service page is supposed to keep user's morale high and to give them an incentive to keep reporting the enemy movements. In order to qualify for an award user has to have a certain number of verified reports. The *_buildAwardsGrid* method returns a *GridView* widget that displays all available

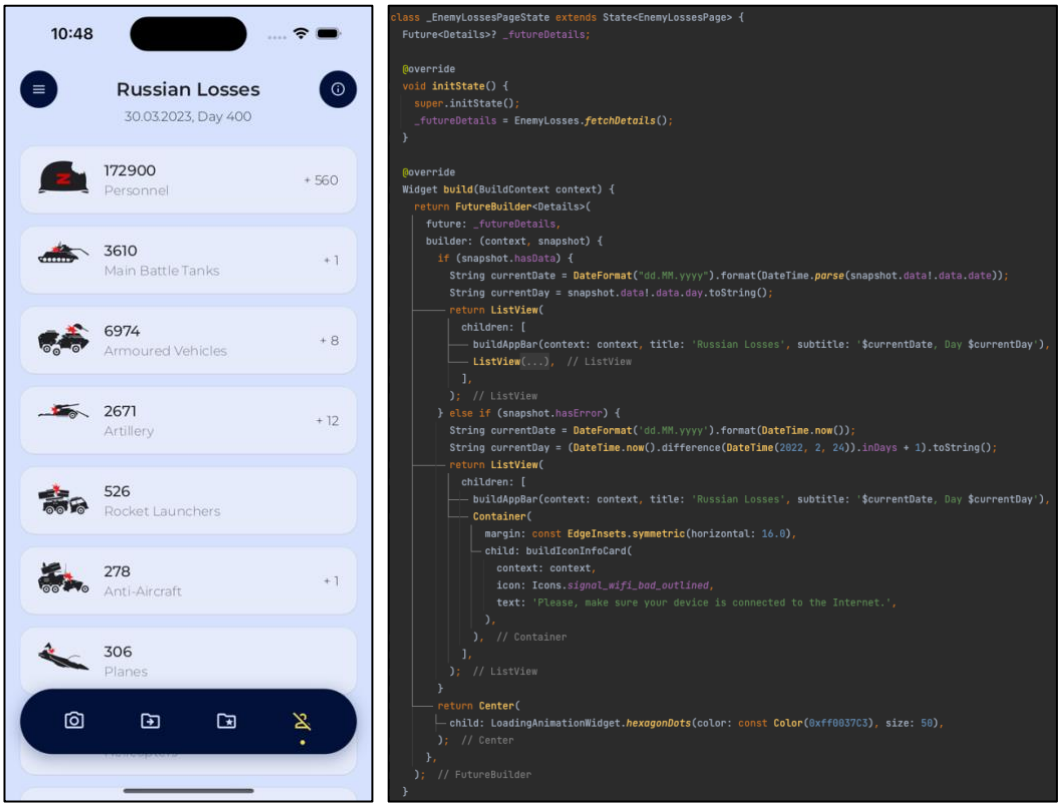
awards. It retrieves the current user's data from Firebase and uses it to calculate the progress towards earning each award. The *_buildAwardContainer* method returns a *CircularProgressIndicator* widget that displays an award's progress towards earning each award.


```
Widget _buildAwardsGrid() {
  final userID = FirebaseAuth.instance.currentUser?.uid;
  return StreamBuilder<DocumentSnapshot>(
    stream: FirebaseFirestore.instance.collection('users').doc(userID).snapshots(),
    builder: (context, snapshot) {
      if (!snapshot.hasData) {
        return Center(
          child: LoadingAnimationWidget.hexagonDots(color: const Color(0xff0037C3), size: 50),
        ); // Center
      }

      _verifiedReportsCounter = snapshot.data?['verified_reports'];

      return GridView.count(
        physics: const ScrollPhysics(),
        crossAxisCount: 3,
        crossAxisSpacing: 15.0,
        childAspectRatio: 0.85,
        shrinkWrap: true,
        children: [
          _buildAwardContainer('Badge_for_Services_to_the_AFU', 'Badge "For Services to the AFU"', 1),
          _buildAwardContainer('Medal_for_Assistance_to_the_AFU', 'Medal "For Assistance to the AFU"', 5),
          _buildAwardContainer('Medal_for_Defense_of_the_Motherland', 'Medal "For Defender of the Motherland"', 10),
          _buildAwardContainer('Order_for_Courage_III', 'Badge of the Order "For Courage" III class', 20),
          _buildAwardContainer('Order_of_Merit_III', 'Order of Merit III class', 25),
          _buildAwardContainer('Order_of_Merit_III_Swords', 'Order of Merit III class with Swords', 30),
          _buildAwardContainer('Order_of_Merit_II', 'Order of Merit II class', 30),
          _buildAwardContainer('Order_of_Merit_II_Swords', 'Order of Merit II class with Swords', 35),
          _buildAwardContainer('Hero_of_Ukraine', 'Hero of Ukraine', 50),
        ],
      ); // GridView.count
    },
  ); // StreamBuilder
}
```

Figure 23: Awards for Service Page

Russian Losses page keeps user's morale high and allows them to receive up-to-date information regarding the losses that the Russian Armed Forces and the so-called DPR/LPR Militia have sustained since the outbreak of war in Ukraine from the API ("Russian Warship" 2022).

The page is also quite simple, essentially being a *ListView* containing a list of widgets which display losses by category with a daily increase and corresponding icon. The *initState* method initializes a *_futureDetails* variable with the result of calling the *fetchDetails* method from the *EnemyLosses* class. This method returns a *Details* object, which is the type of the future used in the *FutureBuilder* widget. The data on the page is retrieved from the snapshot object returned by *FutureBuilder*. If the snapshot has data, the statistics are displayed; otherwise, an error message is shown.




```dart
class _EnemyLossesPageState extends State<EnemyLossesPage> {
  Future<Details>? _futureDetails;

  @override
  void initState() {
    super.initState();
    _futureDetails = EnemyLosses.fetchDetails();
  }

  @override
  Widget build(BuildContext context) {
    return FutureBuilder<Details>(
      future: _futureDetails,
      builder: (context, snapshot) {
        if (snapshot.hasData) {
          String currentDate = DateFormat("dd.MM.yyyy").format(DateTime.parse(snapshot.data!.data.date));
          String currentDay = snapshot.data!.data.day.toString();
          return ListView(
            children: [
              buildAppBar(context: context, title: 'Russian Losses', subtitle: '$currentDate, Day $currentDay'),
              ListView(...),  // ListView
            ],
          );  // ListView
        } else if (snapshot.hasError) {
          String currentDate = DateFormat('dd.MM.yyyy').format(DateTime.now());
          String currentDay = (DateTime.now().difference(DateTime(2022, 2, 24)).inDays + 1).toString();
          return ListView(
            children: [
              buildAppBar(context: context, title: 'Russian Losses', subtitle: '$currentDate, Day $currentDay'),
              Container(
                margin: const EdgeInsets.symmetric(horizontal: 16.0),
                child: buildIconInfoCard(
                  context: context,
                  icon: Icons.signal_wifi_bad_outlined,
                  text: 'Please, make sure your device is connected to the Internet.',
                ),
              ),  // Container
            ],
          );  // ListView
        }
        return Center(
          child: LoadingAnimationWidget.hexagonDots(color: const Color(0xff0037C3), size: 50),
        );  // Center
      },
    );  // FutureBuilder
  }
```

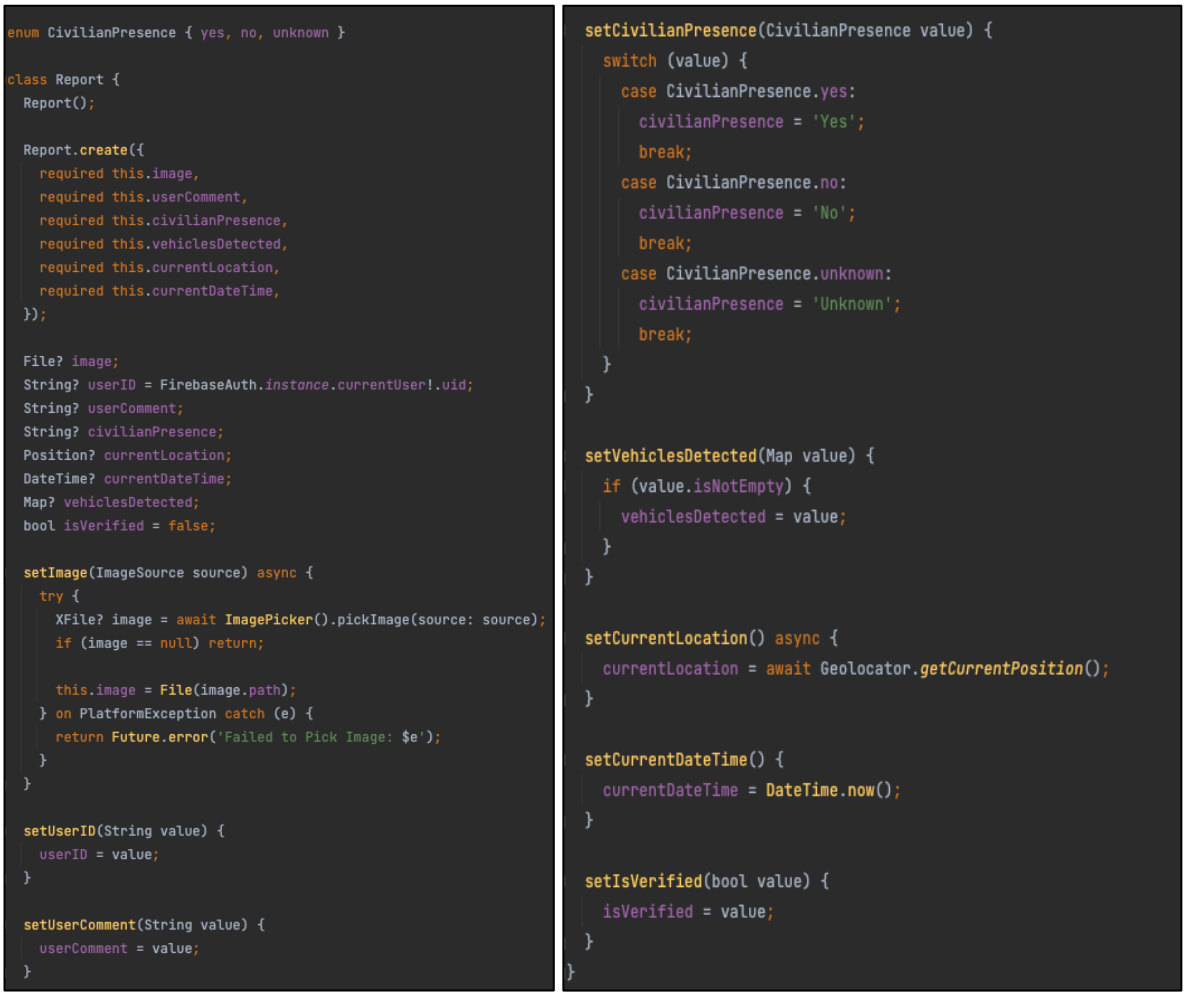
Figure 24: Russian Losses Page

4.2.2 Domain Layer

The logic behind the *Report the Enemy* page is in *report.dart* file and some of the important pieces of code can be seen in Figure 25 below. Class *Report* has several instance variables representing the properties of a report, such as image, *userComment*, *civilianPresence*, *currentLocation*, *currentDateTime*, *vehiclesDetected* and *isVerified*.

The default constructor initializes an empty report with all instance variables set to null or false. The named constructor create allows creating a new report by passing all information as arguments.

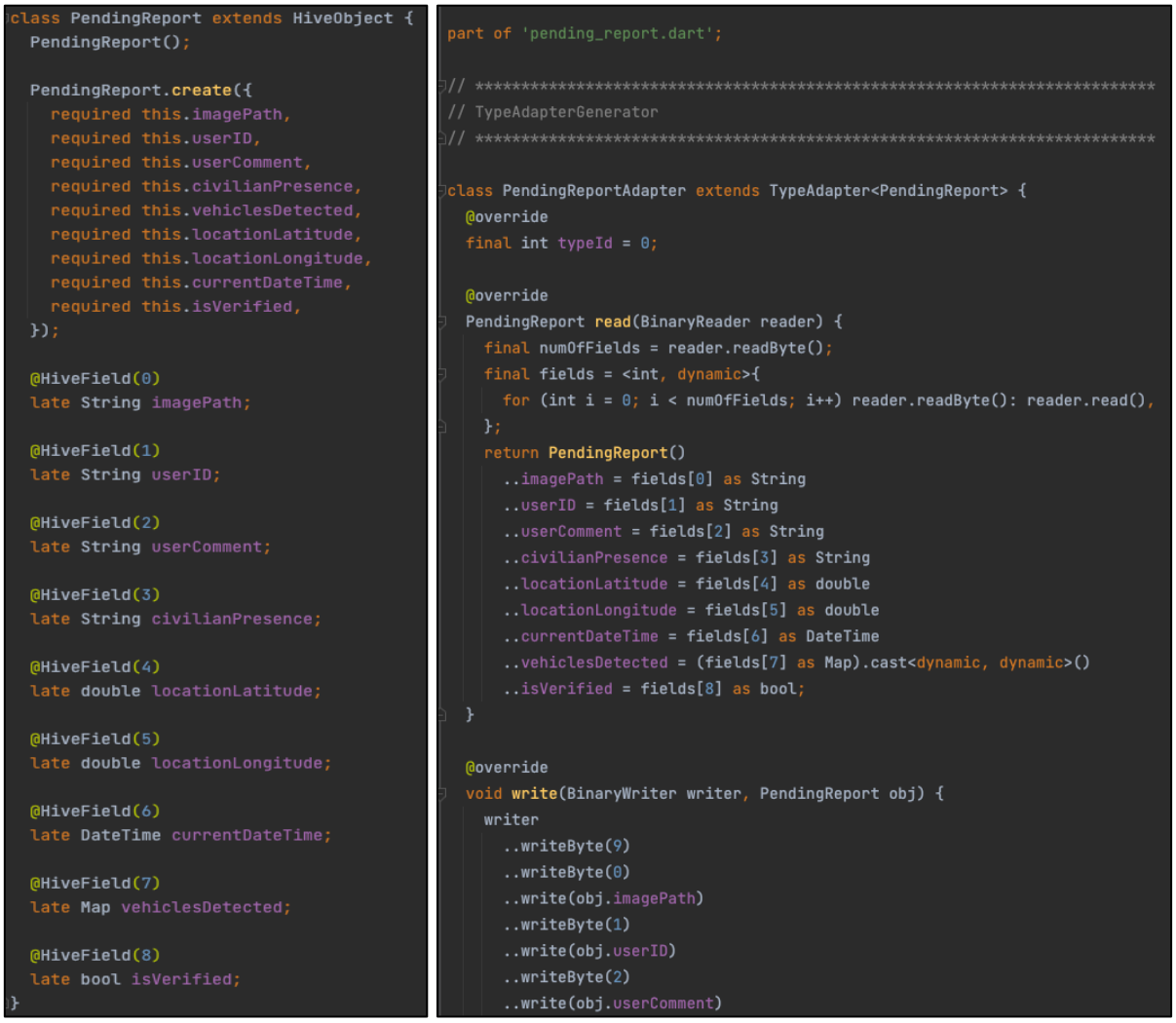
The setUserID, setUserComment, setCivilianPresence, setVehiclesDetected and setIsVerified methods are setters for their corresponding instance variables. The *setImage* method uses the *ImagePicker* plugin to select an image from the device's gallery or camera and set it as the report's image property. The other method, *setCurrentLocation* uses the *GeoLocator* plugin to get the current location of the device and set it as the report's current location property. The *setCurrentDateTime* method sets the current date and time as the report's current date and time property.



```dart
enum CivilianPresence { yes, no, unknown }

class Report {
  Report();

  Report.create({
    required this.image,
    required this.userComment,
    required this.civilianPresence,
    required this.vehiclesDetected,
    required this.currentLocation,
    required this.currentDateTime,
  });

  File? image;
  String? userID = FirebaseAuth.instance.currentUser!.uid;
  String? userComment;
  String? civilianPresence;
  Position? currentLocation;
  DateTime? currentDateTime;
  Map? vehiclesDetected;
  bool isVerified = false;

  setImage(ImageSource source) async {
    try {
      XFile? image = await ImagePicker().pickImage(source: source);
      if (image == null) return;

      this.image = File(image.path);
    } on PlatformException catch (e) {
      return Future.error('Failed to Pick Image: $e');
    }
  }

  setUserID(String value) {
    userID = value;
  }

  setUserComment(String value) {
    userComment = value;
  }

  setCivilianPresence(CivilianPresence value) {
    switch (value) {
      case CivilianPresence.yes:
        civilianPresence = 'Yes';
        break;
      case CivilianPresence.no:
        civilianPresence = 'No';
        break;
      case CivilianPresence.unknown:
        civilianPresence = 'Unknown';
        break;
    }
  }

  setVehiclesDetected(Map value) {
    if (value.isNotEmpty) {
      vehiclesDetected = value;
    }
  }

  setCurrentLocation() async {
    currentLocation = await Geolocator.getCurrentPosition();
  }

  setCurrentDateTime() {
    currentDateTime = DateTime.now();
  }

  setIsVerified(bool value) {
    isVerified = value;
  }
}
```

Figure 25: Report Class

Next, there is *pending_report.dart* file as demonstrated by Figure 26 below. *PendingReport* class is a Dart definition for a *PendingReport* object that uses the Hive library for object serialization and storage. This class is used to generate *PendingReportAdapter* in *pending_report.g.dart*.



```dart
class PendingReport extends HiveObject {
  PendingReport();

  PendingReport.create({
    required this.imagePath,
    required this.userID,
    required this.userComment,
    required this.civilianPresence,
    required this.vehiclesDetected,
    required this.locationLatitude,
    required this.locationLongitude,
    required this.currentDateTime,
    required this.isVerified,
  });

  @HiveField(0)
  late String imagePath;

  @HiveField(1)
  late String userID;

  @HiveField(2)
  late String userComment;

  @HiveField(3)
  late String civilianPresence;

  @HiveField(4)
  late double locationLatitude;

  @HiveField(5)
  late double locationLongitude;

  @HiveField(6)
  late DateTime currentDateTime;

  @HiveField(7)
  late Map vehiclesDetected;

  @HiveField(8)
  late bool isVerified;
}
```

```dart
part of 'pending_report.dart';

// **************************************************************************
// TypeAdapterGenerator
// **************************************************************************

class PendingReportAdapter extends TypeAdapter<PendingReport> {
  @override
  final int typeId = 0;

  @override
  PendingReport read(BinaryReader reader) {
    final numOfFields = reader.readByte();
    final fields = <int, dynamic>{
      for (int i = 0; i < numOfFields; i++) reader.readByte(): reader.read(),
    };
    return PendingReport()
      ..imagePath = fields[0] as String
      ..userID = fields[1] as String
      ..userComment = fields[2] as String
      ..civilianPresence = fields[3] as String
      ..locationLatitude = fields[4] as double
      ..locationLongitude = fields[5] as double
      ..currentDateTime = fields[6] as DateTime
      ..vehiclesDetected = (fields[7] as Map).cast<dynamic, dynamic>()
      ..isVerified = fields[8] as bool;
  }

  @override
  void write(BinaryWriter writer, PendingReport obj) {
    writer
      ..writeByte(9)
      ..writeByte(0)
      ..write(obj.imagePath)
      ..writeByte(1)
      ..write(obj.userID)
      ..writeByte(2)
      ..write(obj.userComment)
```

Figure 26: PendingReport Class

Some key features of this class: the *@HiveType* annotation specifies that this class can be serialized and stored using Hive. The *typeID* parameter is used to uniquely identify this class within the Hive database. The *@HiveField* annotation is used to mark instance variables as serializable and storable with Hive. Each instance variable is assigned a unique index for efficient storage and retrieval.

The *PendingReport* class has a default constructor and a named constructor create. The create constructor initializes all the required properties of a *PendingReport* object. This object includes an image path, user ID, comment, location, vehicles detected, timestamp and verification status.

In Figure 27 below we can observe implementation of the *Classifier* utility class in *classifier.dart* which contains various methods used for image classification. The first static method, *createFromFirebase* takes three required parameters: *assetPath*, *maxCount*, and *confidenceThreshold*. It creates an

instance of the *ImageLabeler* class using the provided parameters. The *ImageLabeler* class is provided by on-device Google ML Kit API which detects and extracts information about entities in an image.

The second static method, *createFromLocal* is similar to *createFromFirebase*, but it takes a local asset path instead of a Firebase asset path. It also includes an additional asynchronous method *_getModel* which retrieves the local model path.

```dart
class Classifier {
  static late ImageLabeler _imageLabeler;

  static createFromFirebase({required assetPath, maxCount = 10, confidenceThreshold = 0.5}) {
    final options = LocalLabelerOptions(
      modelPath: assetPath,
      maxCount: maxCount,
      confidenceThreshold: confidenceThreshold,
    );
    _imageLabeler = ImageLabeler(options: options);
  }

  static createFromLocal({required assetPath, maxCount = 10, confidenceThreshold = 0.5}) async {
    final modelPath = await _getModel(assetPath);
    final options = LocalLabelerOptions(
      modelPath: modelPath,
      maxCount: maxCount,
      confidenceThreshold: confidenceThreshold,
    );
    _imageLabeler = ImageLabeler(options: options);
  }

  static Future<String> _getModel(String assetPath) async {
    if (Platform.isAndroid) {
      return 'flutter_assets/$assetPath';
    }
    final path = '${(await getApplicationSupportDirectory()).path}/$assetPath';
    await Directory(dirname(path)).create(recursive: true);
    final file = File(path);
    if (!await file.exists()) {
      final byteData = await rootBundle.load(assetPath);
      await file.writeAsBytes(byteData.buffer.asUint8List(byteData.offsetInBytes, byteData.lengthInBytes));
    }
    return file.path;
  }
```

Figure 27: Classifier Class

The next method is probably one of the most important methods in this project. In Figure 28 below the method *classify* takes a *File* object and returns a map containing the classification results. The image is processed using the above-mentioned *ImageLabeler* instance created by either *createFromFirebase* or *createFromLocal*, and the resulting labels and their confidence scores are added to the map.

Then, another static method *_validate* takes a map of classification results and performs additional validation on them. If the map contains only one label and that is “Other” or has a confidence score below 40%, image does not pass the validation. Alternatively, if “Other” has the highest confidence score and the difference between “Other” and the second-highest label’s confidence score is more than 5%, it is also unlikely that there are any AFV in the image. In this case, the map is cleared.

Finally, the dispose method simply closes the *ImageLabeler* instance to preserve resources.

```
static Future<Map> classify(File file) async {
  Map<String, double> classifiedObjects = {};

  final inputImage = InputImage.fromFile(file);
  final List<ImageLabel> labels = await _imageLabeler.processImage(inputImage);

  for (ImageLabel label in labels) {
    num confidence = num.parse(label.confidence.toStringAsFixed(2));
    classifiedObjects[label.label] = confidence.toDouble();
  }

  // return empty collection if did not pass validation

  _validate(classifiedObjects);
  return classifiedObjects;
}

static _validate(Map<String, double> objects) {
  if (objects.length == 1) {
    // if the only label is 'Other' or the only label is an AFV, but confidence score is below 40%
    if (objects.containsKey('Other') || objects.values.every((confidence) => confidence < 0.40)) {
      objects.clear();
    }
  } else if (objects.length > 1) {
    // if 'Other' has most confidence
    if (objects.keys.first == 'Other') {
      // if confidence score difference between 'Other' and AFV is more than 5%
      if (objects.values.elementAt(0) - objects.values.elementAt(1) > 0.05) {
        objects.clear();
      }
    }
  }
}

static dispose() {
  _imageLabeler.close();
}
}
```

Figure 28: Classifier Class

As for *enemy_losses.dart* file, it is quite simple, as demonstrated in Figure 29 below. Thus, there is a class named *EnemyLosses* which has a static function called *fetchDetails* which uses *dio* package to make an HTTP GET request to the API.

Meanwhile, the rest of the file, more specifically, classes *Data*, *Stats*, and *Increase* are used for parsing response from the server with the *Details.fromJson* function of a *Details* class which returns a *Details* object that can later be used throughout the code.

The function also sets up caching for the request using the DioCacheInterceptor and HiveCacheStore classes from the dio_cache_interceptor package. This allows the response to be cached in a Hive database on the device for a maximum of 12 hours, with the option to force-refresh the cache if needed. If there is an error with HTTP request or response parsing, the function throws an exception.

```dart
class EnemyLosses {
  static Dio dio = Dio();

  static Future<Details> fetchDetails() async {
    final cacheDir = await getTemporaryDirectory();
    final cacheStore = HiveCacheStore(cacheDir.path, hiveBoxName: "losses_cache");

    var cacheOptions = CacheOptions(
      store: cacheStore,
      policy: CachePolicy.refreshForceCache,
      priority: CachePriority.normal,
      maxStale: const Duration(hours: 12),
      hitCacheOnErrorExcept: [],
      keyBuilder: (request) {
        return request.uri.toString();
      },
    );  // CacheOptions

    dio.interceptors.add(DioCacheInterceptor(options: cacheOptions));
    Response response = await dio.get('https://russianwarship.rip/api/v1/statistics/latest');

    try {
      return Details.fromJson(response.data);
    } catch (e) {
      throw Exception('Failed to Load Details');
    }
  }
}

class Details {...}

class Data {...}

class Stats {...}

class Increase {...}
```

Figure 29: EnemyLosses Class

Lastly, there is a *utility.dart* file which implements a *Utility* class that contains various helper functions, as shown in Figures 30 and 31 below. Two of the functions are image-related: one of them is *saveImage*, which saves the provided *image* file to the application support folder and returns a new *File* object pointing to the newly saved image. If the image could not be saved for any reason, a *Future.error* is returned. The other function is *deleteImage*, which deletes the image located at the provided *imagePath* from memory. If the image could not be deleted, a *Future.error* is returned.

Also, there is a *requestPermissions* function that requests necessary permissions for the app to function properly, all permissions are requested at the same time rather than when components requiring them are being used. This method request permissions for the camera, photos, and location. It first checks if the necessary permissions have already been granted. If not, it requests the permissions and opens the app settings page if the user has permanently denied the permission. However, there is an issue with Android: up until API level 32 (Android 12), to get access to the photos the application has to request storage permission, whereas for iOS and more recent versions of iOS the application has to request photos permission. But, photos/storage access is required only for testing purposes, in release version of the application only location and camera are required.

```dart
class Utility {
  static final DeviceInfoPlugin deviceInfoPlugin = DeviceInfoPlugin();

  static Future<File> saveImage(File image) async {
    try {
      final Directory appSupportDirectory = await getApplicationSupportDirectory(); // get the app support folder
      final String appSupportPath = appSupportDirectory.path; // get the path to the app support folder
      final String newImagePath = path.join(appSupportPath, path.basename(image.path));
      return await image.copy(newImagePath); // return locally saved image
    } catch (e) {
      return Future.error('Failed to Save Image: $e');
    }
  }

  static Future deleteImage(String imagePath) async {
    try {
      final image = File(imagePath); // get the image from the image path
      await image.delete(); // delete the image from memory
    } catch (e) {
      return Future.error('Failed to Delete Image: $e');
    }
  }
```

Figure 30: Utility Class

```dart
static Future<bool> requestPermissions() async {
  var cameraStatus = await Permission.camera.status;

  if (cameraStatus.isDenied) {
    await Permission.camera.request();
  } else if (cameraStatus.isPermanentlyDenied) {
    openAppSettings();
  }

  PermissionStatus galleryStatus;

  if (Platform.isAndroid) {...} else {
    galleryStatus = await Permission.photos.request();
    if (galleryStatus.isDenied) {
      await Permission.photos.request();
    } else if (galleryStatus.isPermanentlyDenied) {
      openAppSettings();
    }
  }

  LocationPermission permission = await Geolocator.checkPermission();

  if (permission == LocationPermission.denied) {
    permission = await Geolocator.requestPermission();
  } else if (permission == LocationPermission.deniedForever) {
    openAppSettings();
  }

  if (cameraStatus.isDenied || galleryStatus.isDenied || permission == LocationPermission.denied) {
    return false;
  } else {
    return true;
  }
}
```

Figure 31: Utility Class

4.2.3 Data Layer

Another important aspect of this project is using Firebase, the app development platform, because it is capable of providing such useful features as database, storage, machine learning, authentication, and testing, all of which will be introduced at different stages of this project. After behind-the-scenes configuration of the iOS and Android versions of the app through the terminal and Firebase dashboard, *firebase_options.dart* file, containing some default options, is automatically generated and it is possible to work with Firebase now.

Next, the *firebase_database.dart* file was created to encompass all of the database and authentication related methods in a single class called *FirebaseDatabase*. This is a static class, so we do not have to

instantiate this class to access its methods. Firstly, we are going to discuss methods related to user authentication.

In Figure 32 below, there are two methods: *createUserWithEmailAndPassword* and a private *_uploadUserDetails*. The first method creates a basic user account in Firebase with email and password, then it updates it's display name and calls the second method, which in turn creates a document with user's personal details and verified reports count, where a key is user account ID. In case an error has occurred, *createWithEmailAndPassword* will return an error message from a *codeResponses* map, or if the error code does not belong to that map, an unknown error message.

```dart
static Future<String?> createUserWithEmailAndPassword(
    String email, String password, String firstName, String lastName, String passportNumber) async {
  Map<String, String?> codeResponses = {
    "email-already-in-use": 'Error: Email Already in Use',
    "invalid-email": 'Error: Invalid Email',
    "operation-not-allowed": null,
    "weak-password": 'Error: Weak Password',
  };

  try {
    final userCredential = await FirebaseAuth.instance.createUserWithEmailAndPassword(
      email: email.trim(),
      password: password.trim(),
    );
    await userCredential.user?.updateDisplayName('${firstName.trim()} ${lastName.trim()}');
    await _uploadUserDetails(firstName.trim(), lastName.trim(), passportNumber.trim(), userCredential.user!.uid);
    return null;
  } on FirebaseAuthException catch (error) {
    return codeResponses[error.code] ?? "Error: Unknown Error";
  }
}
```

```dart
static Future _uploadUserDetails(String firstName, String lastName, String passportNumber, String userID) async {
  final userDocument = _usersCollection.doc(userID);

  Map<String, dynamic> dataToSend = {
    'passport_number': passportNumber,
    'first_name': firstName,
    'last_name': lastName,
    'verified_reports': 0,
  };

  try {
    userDocument.set(dataToSend).then((documentSnapshot) => debugPrint("Added Data with ID: $userID"));
  } catch (e) {
    return Future.error('Failed to Upload User Details to Firebase: $e');
  }
}
```

Figure 32: Firebase User Creation

The other methods, *signInWithEmailAndPassword*, *updatePassword* and *sendPasswordResetEmail* are shown in Figure 33 below. They are rather straight-forward and follow the same principle as previously discussed methods. However, Firebase requires user to be recently authenticated in order to change their password, so user has to enter their current password to reauthenticate.

```dart
static Future<String?> signInWithEmailAndPassword(String email, String password) async {
  Map<String, String?> codeResponses = {
    "invalid-email": 'Error: Invalid Email',
    "user-disabled": 'Error: Disabled Account',
    "user-not-found": 'Error: User Not Found',
    "wrong-password": 'Error: Incorrect Password',
  };

  try {
    await FirebaseAuth.instance.signInWithEmailAndPassword(email: email.trim(), password: password.trim());
    return null;
  } on FirebaseAuthException catch (error) {
    return codeResponses[error.code] ?? "Error: Unknown Error";
  }
}
```

```dart
static Future<String?> updatePassword(String oldPassword, String newPassword) async {
  User user = FirebaseAuth.instance.currentUser!;
  AuthCredential credential = EmailAuthProvider.credential(email: user.email!, password: oldPassword.trim());

  Map<String, String?> codeResponses = {
    "user-mismatch": null,
    "user-not-found": 'Error: User Not Found',
    "invalid-credential": null,
    "invalid-email": null,
    "wrong-password": 'Error: Wrong Password',
    "invalid-verification-code": null,
    "invalid-verification-id": null,
    "weak-password": 'Error: Weak Password',
    "requires-recent-login": null
  };

  try {
    await user.reauthenticateWithCredential(credential);
    await user.updatePassword(newPassword.trim());
    return null;
  } on FirebaseAuthException catch (error) {
    return codeResponses[error.code] ?? "Error: Unknown Error";
  }
}

static Future sendPasswordResetEmail(String email) async {
  await FirebaseAuth.instance.sendPasswordResetEmail(email: email.trim());
}
```

Figure 33: Firebase Authentication

Now, we will be discussing data management related methods. In Figure 34 below, *getUserDetails* attempts to retrieve a document from Firebase collection using the user ID passed as input. If the

document exists, it returns its data as a map. If the document does not exist, it throws and exception. If there is any error while trying to get the user details, the method catches an error and then rethrows the error to propagate it further.

```dart
static Future<Map<String, dynamic>> getUserDetails(String userID) async {
  try {
    final userDocument = await _usersCollection.doc(userID).get();
    if (userDocument.exists) {
      return userDocument.data() as Map<String, dynamic>;
    } else {
      throw Exception('User Details Not Found');
    }
  } catch (e) {
    debugPrint('Failed to Get User Details: $e');
    rethrow;
  }
}
```

Figure 34: Firebase User Details

In Figure 35 below there are three methods: *_uploadImage*, *uploadReport* and *uploadPendingReport*. The first method takes a *File* object as input and uploads it to Firebase Storage under the "images" directory with a unique file name generated from the current timestamp. If the upload is successful, it returns the download URL of the uploaded image. If it fails, it returns a Future with an error message.

The second method, *uploadReport*, takes a *Report* object as input. First of all, it calls the *_uploadImage* method to get the image URL because it will be stored in Firebase Storage, whereas reports are stored in Firebase Database. It then constructs a map containing the report data, including the image URL, user ID, comment, civilian presence, vehicles detected, location, time, and verification status. Finally, it attempts to add the report data to a Firebase collection using the *add()* method. If the addition is successful, it logs the ID of the added data. If it fails, it returns a Future with an error message.

The third method, *uploadPendingReport*, is basically the same as *uploadReport* method, there is a very minor difference in a type of object that they are taking and how they pass the image to the *_uploadImage* method. Ideally, those two methods should have had the same name, but Flutter does not support function overloading.

```dart
static Future<String> _uploadImage(File image) async {
  final String uniqueFileName = DateTime.now().millisecondsSinceEpoch.toString();

  final Reference referenceRoot = FirebaseStorage.instance.ref();
  final Reference referenceDirImages = referenceRoot.child('images');
  final Reference referenceImageToUpload = referenceDirImages.child(uniqueFileName);

  try {
    await referenceImageToUpload.putFile(image);
    return await referenceImageToUpload.getDownloadURL();
  } catch (e) {
    return Future.error('Failed to Upload Image to Firebase: $e');
  }
}

static Future uploadReport(Report report) async {
  final imageURL = await _uploadImage(report.image!);

  Map<String, dynamic> dataToSend = {
    'image': imageURL,
    'user': report.userID,
    'comment': report.userComment,
    'civilians': report.civilianPresence,
    'vehicles': report.vehiclesDetected,
    'location': GeoPoint(report.currentLocation!.latitude, report.currentLocation!.longitude),
    'time': report.currentDateTime,
    'verified': report.isVerified,
  };

  try {
    _reportsCollection
        .add(dataToSend)
        .then((documentSnapshot) => debugPrint("Added Data with ID: ${documentSnapshot.id}"));
  } catch (e) {
    return Future.error('Failed to Upload Report to Firebase: $e');
  }
}
```

```dart
static Future uploadPendingReport(PendingReport pendingReport) async {
  final imageURL = await _uploadImage(File(pendingReport.imagePath));

  Map<String, dynamic> dataToSend = {
    'image': imageURL,
    'user': pendingReport.userID,
    'comment': pendingReport.userComment,
    'civilians': pendingReport.civilianPresence,
    'vehicles': pendingReport.vehiclesDetected,
    'location': GeoPoint(pendingReport.locationLatitude, pendingReport.locationLongitude),
    'time': pendingReport.currentDateTime,
    'verified': pendingReport.isVerified,
  };

  try {
    _reportsCollection
        .add(dataToSend)
        .then((documentSnapshot) => debugPrint("Added Data with ID: ${documentSnapshot.id}"));
  } catch (e) {
    return Future.error('Failed to Upload Report to Firebase: $e');
  }
}
```

Figure 35: Firebase Upload

Lastly, the screenshot below in Figure 36 shows the example of how the data is stored in Firebase Database. Owing to the fact that Firebase is a NoSQL database, it is very easy to set it up and it is possible to change the database design multiple times during the development process without issues.

Figure 36: Firebase Documents

Another file is *hive_database.dart* where all Hive related methods are located. Hive is another NoSQL database and in this project it is primarily used to locally store pending reports until they are expired or uploaded to the Firebase.

Below in Figure 37 we can see the code that provides a simple interface for managing pending reports and defines *HiveDatabase* class with several static methods. The first method, *getPendingReports*, returns a Hive box of type *PendingReport* which is used to store and retrieve the pending reports. The second method, *getAllPendingReports*, returns a list if all the pending reports stored in the Hive box. The third method, *getPendingReportByKey*, takes a key as input and returns the pending report with that key if it exists in the Hive box. The fourth method, *getPendingReportByIndex*, takes an index as input and returns the pending report at that index in the Hive box if it exists.

Another method, *savePendingReport*, takes a *Report* object and a *File* object as input, creates a new *PendingReport* object with the report data and image path, and adds it to the Hive box.

Next, *deletePendingReport* takes a *PendingReport* object as input, deletes the image associated with the report, and then deletes the report from the Hive box. The last method, *deleteAllReports*, deletes all the pending reports stored in the Hive box by iterating over them and calling the *deletePendingReport* method on each report. Although Hive has a simpler of deleting all the documents from a box, deletion of images from the application directory would impose a problem as at some point there might be other images not related to the pending reports.

```
static Box<PendingReport> getPendingReportsBox() {
  return Hive.box<PendingReport>('pending_reports');
}

static List<PendingReport> getAllPendingReports() {
  return getPendingReportsBox().values.toList().cast<PendingReport>();
}

static PendingReport? getPendingReportByKey(key) {
  return getPendingReportsBox().get(key);
}

static PendingReport? getPendingReportByIndex(int index) {
  return getPendingReportsBox().getAt(index);
}

static savePendingReport(Report report, File image) {
  final pendingReport = PendingReport.create(
    imagePath: image.path,
    userID: report.userID!,
    userComment: report.userComment!,
    civilianPresence: report.civilianPresence!,
    vehiclesDetected: report.vehiclesDetected!,
    locationLatitude: report.currentLocation!.latitude,
    locationLongitude: report.currentLocation!.longitude,
    currentDateTime: report.currentDateTime!,
    isVerified: report.isVerified,
  );

  getPendingReportsBox().add(pendingReport);
}

static deletePendingReport(PendingReport pendingReport) async {
  await Utility.deleteImage(pendingReport.imagePath);
  pendingReport.delete();
}

static deleteAllPendingReports() async {
  if (getPendingReportsBox().isNotEmpty) {
    await Future.forEach(getAllPendingReports(), (pendingReport) async {
      await deletePendingReport(pendingReport);
    });
  }
}
```

Figure 37: Hive Database

4.3. Deep Learning Model

Originally, we planned to use the combination of Google AutoML object classification model and Google ML Kit for Flutter which includes object detection API. After training a model for the interim report, we decided to make a separate application as a testbed to deploy that model there and see how it performs. The behaviour of the model was quite unstable: sometimes it worked well and one or multiple types of vehicles were detected correctly, sometimes a single vehicle was detected as two or three vehicles, sometimes no vehicles at all were detected, as demonstrated in Figure 38.

Figure 38: Object Detection Output

Apparently, the issue was in object detection model. Unfortunately, it turned out that it is not possible to use custom object detection models, as Google ML Kit only supports custom image classification models, according to the official documentation (Google Developers 2023).

Moreover, one of ML Kit developers on StackOverflow confirmed that information and said that there are plans to make the detector swappable. The same developer also said that “The object detector does not only use colour to distinct the object from the background, it also use 'feature points' of the objects. For example, a colourful box would be easier to be detected comparing to a red box given the same white background. Our model was designed to detect large / prominent objects, with visual search as a use-case (IKEA, Adidas, etc. the user is not a robot, and can frame), and efficiency

(power/latency) in mind. It can detect some small objects sometimes, but it's not really good at it." and that an alternative model in a sample app from TensorFlow team is at least 5 times larger / slower than the ML Kit (Shiyu 2020). Another person on StackOverflow mentioned that ML Kit object detection API can only detect up to 5 largest objects and suggested using TensorFlow directly if it necessary to detect more than 5 objects (Chrisito 2020).

Thus, object detection with Google ML Kit was not suitable for the purposes of this project because very often there is more than 5 armoured fighting vehicles in a column and they are intentionally painted to blend with the environment, so using colour and texture for their detection would be more difficult than any other objects.

It was rather evident that a different library was necessary to deploy the model in Flutter application. However, official TensorFlow and TensorFlow Lite plugins for Flutter have not been updated in almost two years and GitHub repositories are full of open issues, with many open simply because of the deprecated dependencies that those plugins are using. Meanwhile Google ML Kit's last update was just 4 months ago and it seems like there are no other options available for the time being.

So we decided to keep using Google ML Kit, but instead of very unreliable object detection, we decided to simply do image classification with slightly lower threshold and some additional validation rules to allow for situations when two or more types of AFV are present in the same image. A small benefit of using Google ML Kit is that it works equally well with both local models and Firebase-hosted models.

4.4. Conclusions

To conclude, we have demonstrated how the Flutter application has been developed according to the principles of Clean Architecture. We have also covered the most important parts of our codebase across all layers: presentation, domain and data.

Despite having issues with choosing an appropriate library for armoured fighting vehicles detection, we still managed to find a solution and deploy an image classification model, both locally and hosted on Firebase, allowing users automatically download new versions of models when they are available.

Overall, the application is simple to use and fulfils its purpose well: user can quickly and effectively report enemy vehicles without being concerned about absence of reliable internet connection.

5. Testing and Evaluation

5.1. Introduction

It is vitally important to ensure that the final product is stable and works as intended, especially considering its intended purpose in a warzone, where some people might potentially risk their lives to report enemy movements. As a result, thorough testing of all features is required. In this chapter, we will cover manual testing, image classification evaluation, user feedback, as well as any modifications made to the application based on that feedback.

5.2. Software Evaluation

We employed manual testing in our project for several reasons. Firstly, manual testing allows for a more comprehensive evaluation of the application as it involves human testers interacting with the application in various ways to identify any defects or issues. This approach is especially useful when testing the application's usability, user experience, and visual design.

Moreover, manual testing also enables us to identify edge cases and scenarios that may not be covered by automated testing. This is important as these edge cases may be critical in ensuring the application's overall reliability and performance. The following tests were carried out on the application to ensure that errors would not disrupt a user's experience:

Test	Test Summary	Expected Result	Actual Result	Status
1	Login using valid credentials	User logged in and directed to the Main Page	User logged in and directed to the Main Page	Pass
2	Login using invalid credentials	User cannot login and notified of invalid credentials	User cannot login and notified of invalid credentials	Pass
3	View Russian Losses Page without Internet	Cached data from the API is displayed within 12 hours	Cached data from the API is displayed within 12 hours	Pass

4	View Awards for Service Page without Internet	Last known verified reports counter is used to display awards received up to date	Last known verified reports counter is used to display awards received up to date	Pass
5	Upload pending reports without Internet	Pull down to upload feature is disabled when Internet is unavailable	Pull down to upload feature is disabled when Internet is unavailable	Pass
6	Upload pending reports with Internet	Refresh indicator is spinning, reports are being uploaded and deleted from the page in sequence	Refresh indicator is spinning, reports are being uploaded and deleted from the page in sequence	Pass
7	Take a photo of an object which is not an AFV	User notified of invalid image and cannot submit report	User notified of invalid image and cannot submit report	Pass
8	Submit a valid report on iOS device	Refresh indicator is spinning until the report is uploaded, page is reset	Refresh indicator is spinning until the report is uploaded, page is reset	Pass
9	Submit a valid report on Android device	Refresh indicator is spinning until the report is uploaded, page is reset	Image cannot be validated	Fail
10	Disable Internet after submitting a valid report	Refresh indicator stops spinning, the uploading process is stopped	Refresh indicator spins infinitely or until the Internet is available again	Fail
11	Disable Internet prior to submitting a valid report	Report saved locally and user notified about it	Report saved locally and user notified about it	Pass
12	Pending Report expires after 12 hours	Expired pending report removed from the database and from the UI	Expired pending report removed from the database and from the UI	Pass
13	Pending Report deleted on user swipe left	Pending report removed from the database and from the UI	Pending report removed from the database and from the UI	Pass
14	Logout	User directed to the login screen and pending reports deleted from the database	User directed to the login screen and pending reports deleted from the database	Pass

As we can see, only 2 out of 14 test cases failed. The first issue, not being able to validate the image on Android devices, was already discussed in previous chapter of this report. As a temporary fix, Android is using local image classification model, while iOS devices still benefit of using Firebase-hosted models. The second issue, infinite spinner while trying to upload a report with no internet connection, was also fixed: if the report is not submitted withing 15 seconds, it will be stored in Hive as a pending report. This will prevent losing a report due to disrupted internet connection or if the file is too big to be uploaded in a spot with a relatively slow internet connection.

5.3. Feedback Evaluation

Gathering user feedback was a critical step in testing the application. Engineers may develop biases towards the application and assume certain features are self-explanatory, but non-technical individuals may struggle to understand what the application requires from them. Hence, we decided to reached out to some people from Ukraine to be our test users.

The first test user, Serhii, is a serviceman of Armed Forces of Ukraine and a former Squad Leader. His general impression was very positive, he really liked the UI of *Tank Hunter* and did not have any suggestions on how it can be improved even further. The only request was to enforce security of the application to prevent bad actors from uploading false information. Also, Serhii had some questions regarding the geolocation usage. He was satisfied to hear that geolocation is used only once when the photo is taken, rather than being in constant use. Also, he was happy that the situation when user accidentally denied some permission requests is handled and the application redirects user to settings.

The second test user, Oleksandr, is a civilian. He found the design of *Tank Hunter* quite pleasing to the eye. Regrettably, Oleksandr's non-native proficiency in the English language posed a hindrance to his seamless navigation of the application. Specifically, he encountered difficulties in utilizing the submit button and instead, mistakenly clicked on one of the bottom navigation bar buttons and thought that his report was already uploaded. Consequently, we implemented a design revision to increase the size of the submit button and included Ukrainian translations in our future plans. Another issue Oleksandr raised pertained to the application's request for passport details. As a result of personal information concerns, he expressed a strong reservation towards this particular feature. We have reassured him that it is only a temporary measure and in future another way of enforcing user identity will be used.

5.4. Model Evaluation

The following armoured fighting vehicles were included in our dataset: T-64, T-72, T-80 main battle tanks, BTR and MT-LB armoured personnel carriers, BMD and BMP infantry fighting vehicles, BMP-T tank support fighting vehicles, various infantry mobility vehicles and multiple launch rocket systems. They were included because they are the most widespread vehicles in the Russian and DPR/LPR Army.

An early version of a dataset used for training was a combination of a dataset of images of Russian captured and damaged military vehicles (Stijn Mitzer and Jakub Janovsky 2022) and some web-scraping. Unfortunately, as illustrated in Figures 39 and 40 below, the above-mentioned dataset has plenty of images of heavily damaged or even completely destroyed vehicles that cannot be used for training and took some time to remove.

Figure 39: Intact BMP-1

Figure 40: Destroyed BMP-1

After the first experiment took place, decision was made to keep increased the number of images in a dataset. We used an existing dataset of images of T-72 and BMP (Tuomo Hiippala 2017) and obtained more images through web-scraping. Moreover, the above-mentioned dataset also provided us with a solid amount of images to create negative class to allow the model to recognize images that do not belong to any of the specific classes that we are interested in.

For the final experiment, we decided to use synthetic data to further increase our dataset and we have added 825 images to some of our classes in our dataset, all the images are screenshots from tank combat simulator *War Thunder* ("War Thunder" 2011).

In Figure 41 below we can see a detailed list of labels that were included in each experiment and the number of images that were assigned to each label.

	Version 1	Version 2	Version 3
T-64	205	420	475
T-72	325	1325	1565
T-80	205	270	415
BTR	155	440	440
MT-LB	195	230	285
BMD	135	200	320
BMP	135	1335	1545
BMP-T	–	100	100
IMV	50	220	220
MLRS	120	180	180
Other	–	1500	1500

Figure 41: Dataset Overview

	Experiment 1	Experiment 2	Experiment 3
Average Precision	87%	92%	94%
Precision at 50% Threshold	94.7%	92.8%	95.1%
Recall at 50% Threshold	60.4%	87.6%	86.6%
Total Images	1525	6220	7045
Training Images	1220	4976	5636
Validation Images	156	623	707
Test Images	149	621	702

Figure 42: Experiment Results

After conducting all three experiments, we can compare the results shown in Figure 42 above. The original model did achieve quite good average precision of 87%, but it had a mediocre recall of 60.4%, which was possibly caused by a limited size of a dataset. After a significant increase in a size of dataset, average precision increased by 5%, but the recall at 50% confidence threshold went up by 27.2%. Introducing synthetic data did slightly improve the performance of the model: average precision went from 92% to 94%, whereas recall decreased by 1%, but it's withing the margin of error.

In Figure 43 below we can see precision-recall curves and precision-recall by threshold. Precision-recall curve shows the trade-off between precision and recall at different confidence thresholds. Precision-recall by threshold shows how the model performs on the top-scored label along the full range of confidence threshold values. Charts are shown for Experiments 1 and 3, because there is not much of a difference between the results of Experiments 2 and 3.

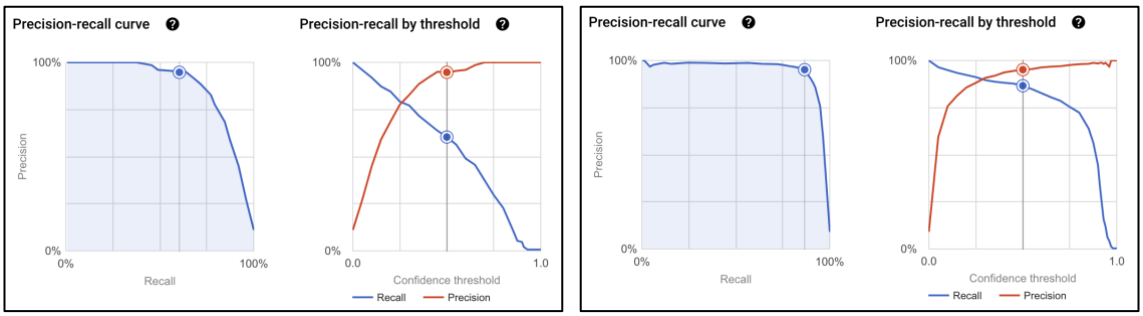


Figure 43: Precision-Recall Charts

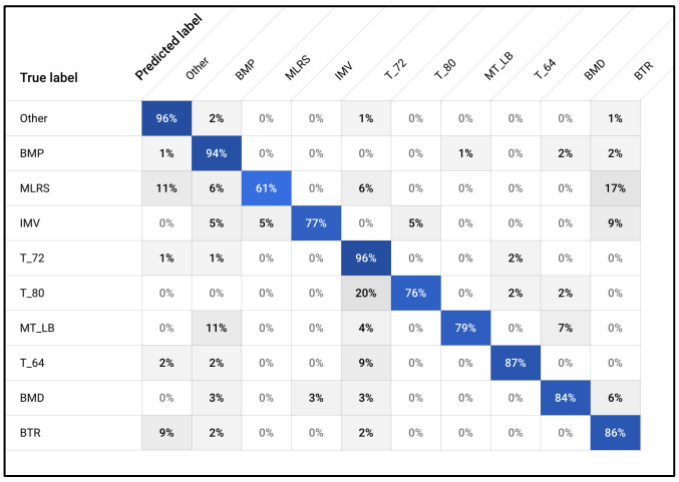


Figure 44: Confusion Matrix

After analysing the confusion matrix for Experiment 3 in Figure 44, some findings can be made: BMP and T-72 have the highest amount of correct predictions; T-64 is relatively easy to distinguish from other Soviet/Russian tanks in most cases (hence only 9% confusion for both T-72), whereas T-80 and T-72 are much more similar (hence 20% confusion); MLRS have the lowest score, but it is probably caused by a lack of images; As for the rest, the results look rather natural, occasionally tracked personnel carriers are predicted as other tracked vehicles (MT-LB and BMP) and wheeled personnel carriers are predicted as other wheeled vehicles (IMV and BTR), but nothing extraordinary.

5.5. Conclusions

In conclusion, *Tank Hunter* has undergone comprehensive testing and evaluation to ensure the delivery of a fully functional application. We employed manual testing, and our image classification model went through several iterations to guarantee a seamless user experience.

Furthermore, the inclusion of user feedback from both a civilian and soldier has enabled us to identify and address any potential issues with the application's usability, ensuring that it is accessible to a diverse range of users.

6. Conclusions and Future Work

6.1. Introduction

In this chapter, the project's conclusions and the application that was created will be discussed, along with any future plans for the application that could not be implemented before the project's deadline.

6.2. Conclusions

In this project, we developed a cross-platform mobile application using Flutter. While designing and developing the application we were adhering to Clean Architecture principles as much as possible, however, we had to make some compromises, given not very complicated nature of our application and lack of experience with the framework.

This application has very modern and minimalistic design, perfectly suitable for Ukrainian people, our target audience. *Tank Hunter's* primary purpose is to allow user to upload reports of enemy armoured fighting vehicles in a very efficient and user-friendly manner. Moreover, the application comes with a few secondary features which purpose is to keep user's morale high and to encourage them to help the Armed Forces of Ukraine.

The application takes into account the fact that reliable internet connection in a warzone is a luxury, so in case user cannot upload the report on the spot, the report is saved locally and user has 12 hours to upload all pending reports.

Prior to being submitted, the report's image is validated in the background using deep learning algorithms for presence of military vehicles. As a result, having some preliminary information regarding armoured fighting vehicles detected facilitates workload for military analysts.

We have created a custom dataset of about 7000 images of armoured fighting vehicles used in Russian Army. With this dataset and Google Vertex AI platform we trained a custom image classification model which has an average precision of 95%. This model is capable of quick and accurate validation of the photo that user takes.

6.3. Future Work

Even though *Tank Hunter* is mostly ready in its current state, prior to officially being deployed In Ukraine a few more things should be made. First of all, the Ukrainian government has to be contacted. If they are interested and give the project a green light, this is what needs to be done:

The dataset used for model training has to be significantly increased. It can be done not only with real images, but with more synthetic data. Since there were reports of Russian Army using even more older tanks and other equipment, new classes should be added to the dataset. Also, a more thorough research regarding possible alternatives for on-device Image Classification / Object Detection in Flutter should take place. Maybe it would be possible to get the TensorFlow plugin to work.

Some test users voiced a concern about the application storing passport numbers. In future, if the government is interested in this project, the registration process can be reinforced by using a governmental API to verify the identity of the user ("Diia" 2023). Thus, no personal will be collected. However, this approach would restrict access to the application to Ukrainian citizens only. Another approach can be implemented to accommodate for foreign volunteers, which involves using an identification document with an NFC chip. This feature was originally planned to implemented in the current version of *Tank Hunter*, but reading data from NFC requires an Apple Developer account, which was request from TU Dublin, but this request, unfortunately, could not be fulfilled in time.

First of all, application has to take a photo of Machine Readable Zone of a document (a few lines at the rear of an identity card or a first page of the passport) in order to get the password (document number, date of birth and expiration date), or alternatively, users can manually input it. Secondly, using the newly obtained password, the application has to read the necessary information from an NFC chip. Access to sensitive information, such as fingerprint and retina images, is restricted and can only be read by authorities, like border control officers (Christian Henzl 2020).

Next, quality of life improvements can be made. Ukrainian translations are a must, because not everyone in the target audience has a level of English enough to use the application without issues. Possibly, additional UI / UX changes will be done in order to better fit Ukrainian audience.

It might be a good idea to make the whole awards system slightly more complicated. For example, some awards can be awarded manually for something exceptional, like a one-off report of the highest degree of importance to the intelligence or providing highly detailed reports on a constant basis.

Moreover, more thorough testing of all component should take place. For example, the behaviour of Image Classifier when photos of AFV were taken from a greater distance or with a worse quality

camera. Or, the behaviour of the system when internet connection is extremely unstable during the bulk upload of pending reports.

Lastly, despite Firebase being a very good tool for developers, it was never intended to be used by anyone outside of a development team. Also, it is too risky to provide access to the application development platform to military analysts. Hence, a separate platform for the military to view and manage reports will be developed, provided they do not have a similar solution already.

Bibliography

1. Andrew Baisden. 2022. “Flutter vs. React Native.” *LogRocket* (blog). June 1, 2022. https://blog.logrocket.com/react-native-vs-flutter/.
2. Anton Klimenko. 2020. “Decoupling Logic from UI in Flutter Application.” *Medium* (blog). November 16, 2020. https://antklim.medium.com/decoupling-logic-from-ui-in-flutter-application-9c07d14a2fd7.
3. “Bachu.” 2022. IOS, Android. Ukraine: Boco Solutions. https://play.google.com/store/apps/details?id=com.ykotmoar.bachu&hl=en_US&gl=US.
4. Bogdan Guriev. 2021. “Firebase Pros and Cons: When You Should and Shouldn’t Use Firebase.” *OSDB* (blog). March 15, 2021. https://osdb.io/firebase-pros-and-cons-when-you-should-and-shouldnt-use-firebase-osdb/#ib-toc-anchor-23.
5. Chrisito. 2020. “How to Recognize and Count Objects with Firebase / ML Kit.” https://stackoverflow.com/questions/63917566/how-to-recognize-and-count-objects-with-firebase-ml-kit.
6. Christian Henzl. 2020. “Access Controls for Electronic Machine-Readable Travel Documents.” *Jumio Engineering & Data Science* (blog). November 25, 2020. https://medium.com/jumio/access-controls-for-electronic-machine-readable-travel-documents-430a6e511d22.
7. Colin Contreary. 2022. “The Best Firebase Alternatives for 2022.” *Embrace* (blog). March 17, 2022. https://blog.embrace.io/firebase-alternatives/amp/.
8. Daniel Legendre and Jouko Vankka. 2020. “Military Vehicle Recognition with Different Image Machine Learning Techniques.” In . Kaunas, Lithuania. https://doi.org/10.1007/978-3-030-59506-7_19.
9. David Ramel. 2022. “Did .NET MAUI Ship Too Soon? Devs Sound Off on ‘Massive Mistake.’” *Visual Studio Magazine* (blog). September 29, 2022. https://visualstudiomagazine.com/articles/2022/09/29/net-maui-complaints.aspx.
10. Den Prystai. 2022. “From Ukrainians to Ukrainians. 5 Digital Tools and Products Created to Help in Wartime.” *Ukraine Now* (blog). October 5, 2022. https://war.ukraine.ua/articles/digital-tools-created-to-help-in-wartime/.
11. “Diia.” 2023. Ukraine: Ministry of Digital Transformation of Ukraine. https://diia.gov.ua/en.
12. Firebase. 2022. “AutoML Vision Edge.” https://firebase.google.com/docs/ml/automl-image-labeling.
13. Google Developers. 2023. “Custom Models with ML Kit.” https://developers.google.com/ml-

kit/custom-models.

14. Harshit Dwivedi. 2020. “Creating a TensorFlow Lite Object Detection Model Using Google Cloud AutoML.” *Heartbeat* (blog). January 13, 2020. https://heartbeat.comet.ml/creating-a-tensorflow-lite-object-detection-model-using-google-cloud-automl-d83f997c1848.
15. Harshit Kumar. 2019. “Quick Intro to Object Detection: R-CNN, YOLO, and SSD.” *Technical Fridays* (blog). March 15, 2019. https://kharshit.github.io/blog/2019/03/15/quick-intro-to-object-detection.
16. “IePPO.” 2022. IOS, Android. Ukraine: quick.ua. https://play.google.com/store/apps/details?id=ua.quick.brpg.pathfinder&hl=uk&gl=US.
17. Krunal Lathiya. 2022. “Firebase vs Heroku: Which Cloud Service Is Better in 2022.” *GCPFirebase* (blog). May 2, 2022. https://gcpfirebase.com/firebase-vs-heroku/.
18. Lionel Sujay Vailshery. 2022. “Cross-Platform Mobile Frameworks Used By Software Developers Worldwide From 2019 to 2021.” Statista. https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/.
19. Matthew Stewart. 2019. “Simple Introduction to Convolutional Neural Networks.” *Towards Data Science* (blog). February 26, 2019. https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac.
20. Michał Kochmański. 2022. “Flutter or Xamarin for Mobile Development in 2022?” *Monterail* (blog). April 7, 2022. https://www.monterail.com/blog/flutter-vs-xamarin.
21. Mike Bentley. 2021. “Agile Feature Driven Development (FDD).” *FeatureFlow* (blog). September 22, 2021. https://www.featureflow.io/feature-driven-development-fdd-a-cheats-guide/.
22. Nick Hotz. 2022. “What Is CRISP DM?” *Data Science Process Alliance* (blog). November 13, 2022. https://www.datascience-pm.com/crisp-dm-2/.
23. Phuong Anh Nguyen. 2021. “The Best Software Development Methodologies for Small Team.” *Fram Blog* (blog). October 9, 2021. https://wearefram.com/blog/software-development-methodologies/#Feature_Driven_Development_FDD.
24. Project Pro. 2022. “Keras vs TensorFlow - Deep Learning Frameworks Battle Royale.” September 16, 2022. https://www.projectpro.io/article/keras-vs-tensorflow-the-differences/454.
25. “Russian Warship.” 2022. Ukraine: WebSpark. https://russianwarship.rip/api-documentation/v1.
26. Scott Reinhard. 2022. “Maps: Tracking the Russian Invasion of Ukraine,” November 14, 2022, The New York Times edition.

https://www.nytimes.com/interactive/2022/world/europe/ukraine-maps.html.

27. Shiyu. 2020. “Improving ML Kit Object Detection.” *StackOverflow*. https://stackoverflow.com/questions/64272741/improving-mlkit-object-detection.
28. Stijn Mitzer and Jakub Janovsky. 2022. “Documenting Russian Equipment Losses During The 2022 Russian Invasion Of Ukraine.” Oryx. https://www.oryxspioenkop.com/2022/02/attack-on-europe-documenting-equipment.html.
29. Sydney J. Freedberg Jr. 2019. “ATLAS: Killer Robot? No. Virtual Crewman? Yes.,” March 4, 2019, Breaking Defense edition. https://breakingdefense.com/2019/03/atlas-killer-robot-no-virtual-crewman-yes/.
30. Tuomo Hiippala. 2017. “Recognizing Military Vehicles in Social Media Images Using Deep Learning.” In . Beijing, China. https://doi.org/10.1109/ISI.2017.8004875.
31. Tyler Crowe. 2022. “What’s New from Firebase at Google I/O 2022.” *The Firebase Blog* (blog). May 11, 2022. https://firebase.blog/posts/2022/05/whats-new-at-google-io.
32. Ville Rissanen, Emil Toivonen, Ruslan Lagashkin, Kalle Saastamoinen, Antti Rissanen, and Jouko Vankka. 2022. “Instance Segmentation and Classification of Armoured Fighting Vehicles.” In . Durban, South Africa. https://doi.org/10.1109/icABCD54961.2022.9855933.
33. “War Thunder.” 2011. Windows, Linux, macOS, PlayStation, Xbox. Russian Federation: Gaijin Entertainment. https://warthunder.com/ie.

Appendix

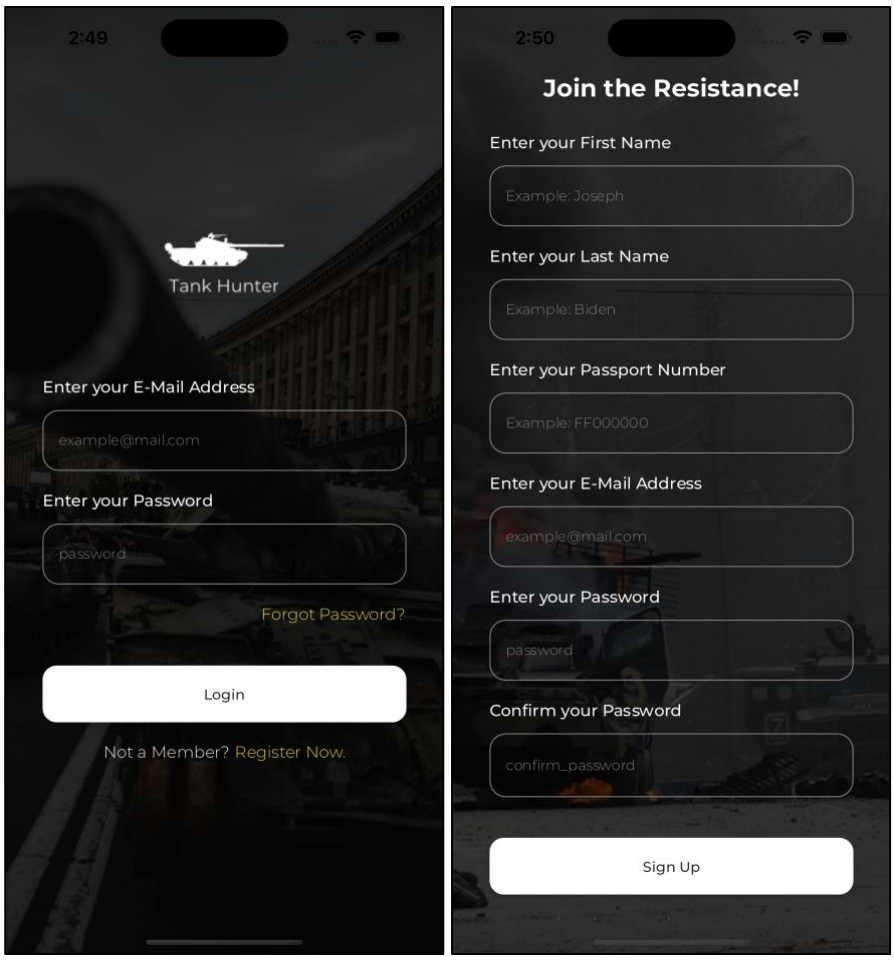


Figure 45: Login and Registration Pages

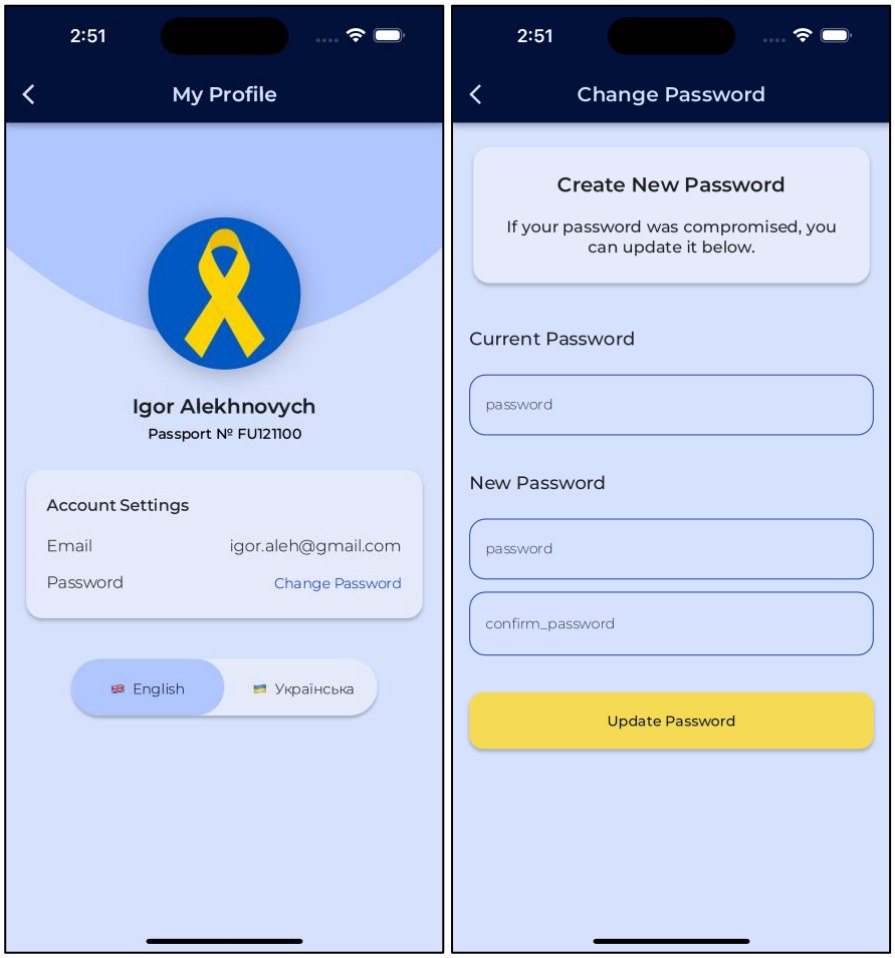


Figure 46: Settings and Change Password Pages

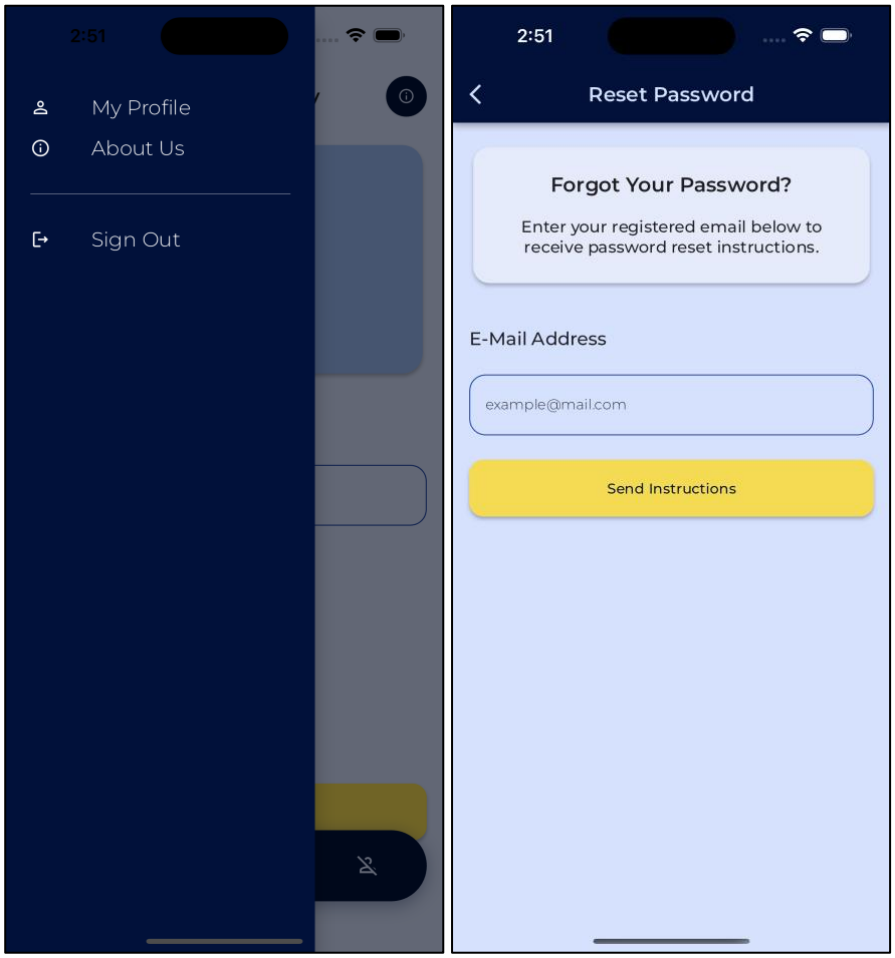


Figure 47: Navigation Drawer and Reset Password Page