2023

CODE CRITTERS: PEST DETECTION

Prepared by
Dhiraj Penumala
Helen Lu
Joshua Chung
Kei Hara
Lance Nillo
Rukun Aaron

EXECUTIVE SUMMARY

Invasive insects are a plague afflicting many countries worldwide, including New Zealand. These insects often lack a natural predator in their new habitat, allowing them to grow their population and outcompete native species for resources rapidly. This disruption in the ecosystem can cause the decline or extinction of native species. As a result of this, New Zealand business that involves agriculture, fishing, horticulture and forestry can suffer acute economic consequences.

The objective of our project is to provide a solution that facilitates the task of detecting invasive insects in New Zealand. The solutions we have created are a web-based and local application that allows users to upload images of insects; the images are then run through a neural network and classified. The classification and their confidence levels are then displayed to the user, who can export these results.

The backend, built with FastAPI, allows for user-friendly and efficient handling of uploads for insect identification. The system's flexible design allows for new machine-learning models to be added without any disruptions. Furthermore, users can easily export their results to XSLX or CSV formats and access more insect information through the GBIF API integration.

The frontend was developed using React and TypeScript and styled using Tailwind CSS and DaisyUi, offering a user-friendly interface. Using Firebase and Firestore, classification results can be directly saved for logged-in users, and images can be retrieved as needed, allowing a smooth workflow from detection to analysis.

Our application will facilitate the process of pest classification and can help expedite the process of identifying and securing invasive species. It achieves this by providing users with accurate and near-instant classification and information pertaining to an image. If a pest is identified and contained early, our application can mitigate the damages caused by it.

We have compiled a list of future improvements we can make to our application. Some include automatic flagging of pests, displaying more related information about a pest and adding a proper user profile page.

TABLE OF CONTENTS

03 Introduction

05 Background

09 Project Specification

11 Project Design

15 Project Implementation

23 Results and Evaluation

27 Future Work

29 Conclusion

30 References

31 Declaration of Authorship

32 Appendices

INTRODUCTION

Aims

Images have always played a pivotal role in insect identification, forming the foundation of global biosecurity systems to pinpoint pests. With the emergence of recent technological advances, notably machine learning, it's becoming evident that images will be paramount in identifying and detecting biosecurity threats in the future.

In collaboration with our client, Landcare Research, our goal for this project is to design a user-friendly graphical user interface (GUI) tailored to complement their own machine learning models. These models are aimed at recognising insects that are either rare, endangered, or classified as pests.

Our primary objective is to craft a user interface tailored to the needs of biosecurity professionals and lab personnel, which will enable them to efficiently query images of unidentified insects against existing Machine Learning models.

Scope

Based on our initial consultation with Landcare Research, our scope has been established as follows:

- Have a GUI that displays insect predictions, and their respective confidence score.
- Have code written in Python/R to allow for ease of interpretation and adoption after project completion.

Approach

The following approach was used to complete the project:

- Complete the backend in Python, and complete the frontend using TypeScript, utilising React, Tailwind CSS and daisyUI for a responsive, modern, and cohesive design.
- Use Firebase and Firestore as our database to store user predictions, and display in the GUI.
- Utilise Jira as our main project management tool, with weekly meetings to communicate progress and any blockers impacting the project.

Outcomes

- A GUI that creates a query by uploading images either by a ‘drag/drop’ or via a menu to ‘choose the file’, and display insect predictions, and their respective confidence score.
- An option to download/save the results as a csv file.
- Functionality for batch queries, when multiple images are inputted at the same time.
- Deployment on a local machine (desktop) or a web-based login.

BACKGROUND

Currently, in the field of Entomology, studying insect populations is difficult, partially due to the inefficiencies in most monitoring methods, which are often labour intensive (Høye et al., 2021).

As a solution for this problem, machine learning, deep learning and computer vision has been touted as a tool which can reduce the workload in insect identification significantly when applied alongside existing methods. Example applications include the use of deep-learning models in use for identifying and counting pest species, the use of Conventional Neural Networks (CNNs) to monitor and detect insects using audio recordings (Høye et al., 2021).

In particular, the application of deep learning is very useful in the field of taxonomic identification, where the accuracy of deep-learning techniques are approaching the accuracy of human experts and a deep-learning model can be made much more accessible compared to consulting experts, which also frees experts time to focus on problems which need their expertise more (Høye et al., 2021).

However, existing platforms in the field of entomology such as INaturalist, tend to use models trained on more common insects, due to their relative ease in dataset acquisition. These models, which although useful are not particularly suited for the field of taxonomic which tend to have a concentration of rarer insects, to which the models' datasets are more likely underrepresented. Existing platforms which are more specialized such as Insect.Id do not allow users to share machine learning models with each other.

Currently, despite the enormous potential in deep learning in the field of taxonomic identification, it seems that the field lacks a good platform that would facilitate the easier distribution of institutional knowledge. Hence we propose a platform which can take the strong points of existing platforms and cater more for taxonomic identification, particularly in helping enable the ease of use and sharing of information while enabling our clients to use more specialised models which would cater more towards their needs.

Tools and Technology Used

Backend

Tools/Tech	Description
Python	The programming language used for backend development. The preferred language for machine learning
ASGI	A standard for asynchronous python web servers, designed to handle multiple concurrent requests efficiently without blocking. It is the core technology used in the backend for asynchronous communication.
FastAPI	An ASGI framework utilized in the backend for building APIs and serving endpoints. It operates on top of an ASGI server and is part of the asynchronous ecosystem in the backend.
Uvicorn	An ASGI web server used to serve the FastAPI endpoints. It provides the necessary platform for hosting and directing requests, particularly for running inference on machine learning models.
Tensorflow	An open-source machine learning library used for training machine learning models that are served by the backend.
PyTorch	An open-source machine learning library used for training machine learning models, providing an alternative to Tensorflow.
Self-hosted Server	The server infrastructure where the ASGI web server (Uvicorn) and the backend application are hosted. This server is self-managed and is made accessible over the internet through a CloudFlare tunnel.
CloudFlare Tunnel	A secure method used to forward traffic from the internet to the self-hosted server, ensuring the backend is accessible while providing additional security and performance benefits.
HTTPX	Used for making HTTP requests within local clients, supports asynchronous operations.

Pywebview	Lightweight alternative to Electron, leverages native web rendering, Python-based.
Pyinstaller	Converts Python programs into executables that can run on Windows, macOS, and Linux.
Innosetup	Used to create installers for Windows applications, well-documented and feature-rich.
SQLAlchemy	A python middleware which allows operations onto a SQLite or PostgreSQL database
SQLite	A library which facilitates a database using SQL

Tools and Technology Used

Frontend

Tools/Tech	Description
React	A JavaScript library for building user interfaces, used in the frontend for creating a responsive and dynamic user experience.
TypeScript	A superset of JavaScript that adds static types, used in the frontend to enhance the development experience through type safety and other features.
Tailwind CSS	A utility-first CSS framework used in the frontend for styling purposes, providing a range of utilities for a more efficient styling process.
daisyUI	A plugin for Tailwind CSS that provides additional UI components, enhancing the range of UI elements available for frontend development.
Firebase	A platform developed by Google, used in the frontend for hosting the application and providing backend services such as Authentication and database hosting.
Firestore	A NoSQL cloud database that is part of the Firebase platform.
Axios	A popular JavaScript library used in the frontend for making HTTP API calls to the backend, facilitating the communication between the frontend and the backend.
CloudFlare Tunnel	A secure method used to forward traffic from the internet to the self-hosted server, ensuring the backend is accessible while providing additional security and performance benefits.

PROJECT SPECIFICIATION

User Requirements

Make and Save Predictions on Insects
Users can insert or drag and drop insect images into our detection page using any model they want. After this, they will be given the scan results. Users can then select images of the scan results and will be able to delete or download them in batches. They will be able to download in two formats: one is a CSV file, and the other is an XSLX file.

View User History
If the users are logged in, they will be given access to the history page, which will automatically save the results of scans done while logged in. In the user history page, the users can also delete and download in batches by selecting images of previous predictions.

Deployed Web Application

Standard Functionality
If a user wants to use the web application, the only requirement is a device that can access the internet. This will allow users to interact with the web app through a browser.

Full Functionality
If a user wants the software's full functionality, they would require an additional requirement: a Google, GitHub or user account. This would give users access to the Firebase database to store results on user's history page.

Android/IOS mobile application
The user simply needs to download and run the CritterSleuth APK from the CodeCritters FlutterApp Github release. Alternatively, the user can compile the application. The user would need Flutter installed on the machine. Afterwards, you can use Flutter to build the application on your mobile device using the CritterSleuth Flutter App. Instructions for the full installation guide is on the CodeCritters GitHub.

Portable Application
The user simply needs to download and run the CritterSleuth setup or executable from the CodeCritters Github releases. Alternatively, the user can compile the application using Python 3.11 and build the CritterSleuth.exe. Instructions for the full installation guide is on the CodeCritters GitHub.

User Interaction

After the user has access to the software, the user will start at the homepage. Here, five different interactions can take place.

Homepage
The user can learn about the software and its features on this page.

Detect Page
The user can interact with the different machine-learning models on the detection page and upload or drag and drop images. After the scan, the user will have access to more interactions. They can view prediction results by clicking on the results, where they will be able to receive more details about the result. Users can now select images in batches by clicking select All or ticking the boxes individually. Then, they can download them as CSV or XLSX. The user can also clear prediction results by clicking the trash icon.

Login
The user can also login if already registered in the system. Once logged in, it gives users functionality to the Firebase database user history page and logging out status. If the user is logged in, the homepage, login and sign up will no longer be accessible.

Sign Up
The user would use this to register an account for the software.

User History Page
In this user history page, there are many interactions. Like the Detection page, users are given the same interactions with the results. The additional interactions allow the user to filter predictions using Model Name, Date and Image name. A search filter also accompanies them. In this user history page, the user can select how many items they want to view on this page and be given options to change which page they want to view.

Alternatively, if the user has access to the local software, the user can interact with the application via the command line allowing the user to bypass the user interface and interact with the back-end directly, which can be useful in for automation.
The Command line allows the user to interact with it to specify the image/s which are to be inferenced, the models available and the model to use for inference, whether to output a CSV or XLSX export file, what to name the output files and where to store the output files.

Use Cases

- The project was built for the needs of Landcare Research (Auckland) to identify and detect insects. It was built in a way where non-programming specialists could understand the code and where the user interface is simple to understand for entomologists and biologists.
- This tool can also be used by anyone who wants to detect insects.

PROJECT DESIGN

For the design of our system, the design choice was inherently shaped by the necessary integration of different technologies. Our backend, developed with FastAPI and running on Uvicorn, was chosen for its asynchronous capabilities. This design choice follows our ambition to handle a multitude of user requests concurrently, thereby optimising performance and ensuring a seamless user experience.

On the frontend, we opted for React in unison with TypeScript, creating a modular and maintainable codebase. This decision, dictated by the need for a robust component-based architecture, also ensures that our application remains scalable and easily navigable. The integration of Tailwind and DaisyUi has streamlined the styling process, ensuring a consistent and modern aesthetic throughout the user interface.

For user authentication and data storage, Firebase was chosen as an to simplify a potentially a complex integration process. Axios, with its intuitive API, was the natural choice for handling our system's HTTP requests, further smoothing the communication between our frontend and backend.

Regarding DevOps, GitHub Actions was at the heart of our CI/CD pipeline, automating crucial stages of our development workflow. ESLint ensured code consistency, and Firebase CLI managing deployments and automated our build process. This approach significantly accelerated our development cycle, ensuring that our application remains both current and reliable.

Backend

Framework/Library Choices and Justifications

FastAPI: We chose to retool the entire backend early on, switching from Flask to FastAPI, some considerations were that:

- Unlike Flask, which is designed for WSGI, FastAPI is designed for ASGI, a next-generation python web server standard which allows the program to be asynchronous.
- FastAPI has native support for Swagger documentation unlike in Flask, where we had to rely on a community supported plugin (Flask-RESTX)
    - Swagger is a significant factor as for backend devs, it is a easy tool to debug the backend and for Frontend devs, it is a great tool to understand the endpoints available in the backend.
- FastAPI is more popular than its competitors (Quart), which should make it easier to find support and documentation.

Uvicorn

- It is the default ASGI web server recommendation for FastAPI
- It is production ready

Tensorflow/Pytorch: Although the backend was initially tooled to be Tensorflow specific for the MVP, we reworked the backend to allow more modular models and be more framework agnostic.

- When we made the backend more flexible, we chose to support both Tensorflow and Pytorch because they are the most popular Python ML libraries.
    - Although our approach makes it trivial to add new libraries, we chose not to include SciKit-learn as with each ML framework added, the file size becomes more bloated. Pytorch by itself added roughly half a gigabyte to the file size.

Pywebview: We chose to utilise PyWebview over Electron, due to several considerations

1. Electron is extremely bloated compared to PyWebview
    a. Electron essentially bundles an entire Chromium browser with your application
    b. Pywebview borrows Windows's built-in Webview2 instead
2. Electron is JavaScript/TypeScript based, and PyWebview is Python based
    a. Electron would require compiling a separate executable file and linking that executable to the backend executable as they are written in different languages.
    b. PyWebview can simply be bundled in the same executable as the backend as it is written in Python
3. Build difficulty
    a. Electron is difficult to package into an executable and would require external packages such as electron-builder.
    b. In comparison, Pyinstaller is relatively straightforward.

InnoSetup: Innosetup was chosen as it is well used, it has all the functions we required and more and there appeared to be a large existing knowledge base to tap into.

Frontend

The user interface is built using React, which provides a component-based architecture that allows for modular and reusable code. Typescript is used to ensure type safety, which reduces runtime errors. Tailwind and DaisyUi are used for styling and UI components, ensuring a consistent and modern look and feel.

Framework/Library Choices and Justifications

React: Chosen for its gentle learning curve, component-based architecture, and vast community support.

Typescript: Offers type safety, which reduces runtime errors and enhances code quality.

Tailwind & DaisyUi: Allows for rapid prototyping, ensuring a consistent design without the messiness of traditional CSS.

Firebase: Provides an all-in-one solution for user authentication, data storage, and security. Its ease of use and integration with React make it a top choice for rapid prototyping.

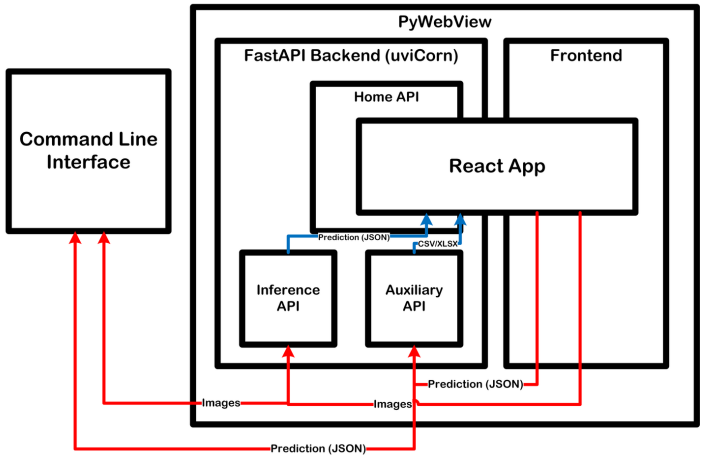
Axios: Simplifies the process of making API calls, offering a cleaner and more intuitive interface than the native fetch.

Data Flows

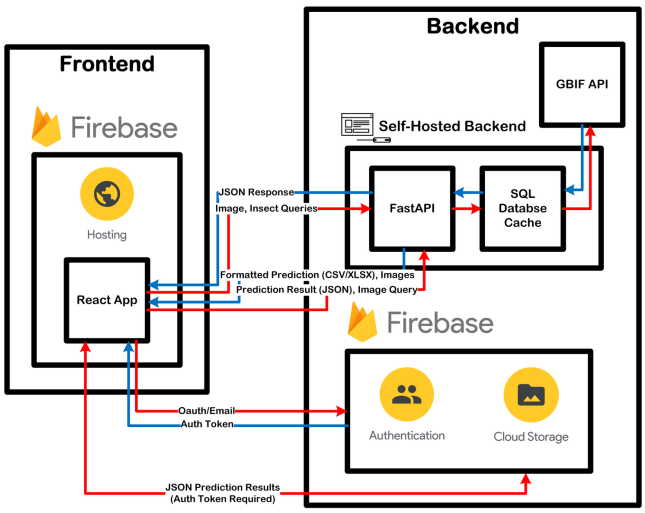
User Authentication: When a user tries to log in or sign up, the frontend communicates with Firebase to authenticate the user. Once authenticated, the user can access the system's features such as user history.

ML Prediction: When a user wants a prediction, the frontend makes an API call using Axios to the backend. Once the prediction is received, it is stored in Firebase for future reference (history page).

Code Critters - Insect Identification App Local Architecture



Code Critters - Insect Identification App Remote Architecture



DevOps

DevOps is a set of practices that combines software development (Dev) and IT operations (Ops) aiming to shorten the systems development life cycle and provide continuous delivery with high software quality. We used GitHub Actions as a Continuous Integration/Continuous Deployment (CI/CD) tool to automate the deployment and testing processes.

Pipeline Design

The pipeline is divided into two main stages

Pull Request: This is the stage where code changes are proposed, reviewed, and tested before they are merged into the main branch

- **Eslint:** Ensures that the code adheres to specified coding standards and catches syntax and runtime errors.
- **npm build:** Validates that the code can be successfully built without errors.
- **Firebase CLI (preview):** Deploys the changes to a preview URL so that they can be reviewed and tested in a live environment before being merged.

Commit to Main Stage: This is the stage where approved changes are merged into the main branch and deployed to production.

- **Firebase CLI (production):** Deploys the merged changes to the production URL.

Justifications

- **Automated Deployment & Testing:** By automating these processes we ensure that they are consistent, repeatable, and less prone to human error. It also speeds up the development cycle, allowing for more frequent releases and faster feedback.
- **Eslint:** With multiple team members working on the project, it's crucial to maintain a consistent coding style and catch potential errors early. Eslint ensures that the codebase remains clean and error-free.
- **Firebase CLI:** By integrating the Firebase CLI into our pipeline, we are able to automatically deploy to preview and production environments, ensuring that changes are always available for review and that the production environment is always up-to-date.
- **GitHub Actions:** GitHub Actions allows us to easily set up our CI/CD pipelines

Benefits of the DevOps

- **Faster Deployment:** Automated pipelines reduce the time between our team writing code and deploying it to production.
- **Increased Quality:** Automated testing and linting ensure that code quality remains high and consistent, furthermore errors are caught early in the development process.
- **Increased Efficiency:** Automation results in less manual intervention, allowing our team to focus and emphasise more on adding features and functionality to the project.

PROJECT IMPLEMENTATION

Backend

Specifics of backend implementation are:

Predictions

- Images are sent with a POST request
- The request is received and validated
- The images are then parsed and hashed to check if the file is on disk already
  - If not, they are saved to static/uploads with the format: name.hash.ext
- The image path is then forwarded along with the model used by the prediction to another function
- The new function then gets the model name and looks up its metadata.txt file which is formatted to the model spec.
- The function then loads the ML model functions (preprocess and predict) from the preprocess.py and predict.py files respectively.
  - This is significant because the ML model functions are NOT imported beforehand, the functions are loaded from file at runtime
  - This allows the models to be added live, without needing to rewrite the backend to import the models or reloading the application
  - This also allows the backend to be ML Framework agnostic as the prediction code does not directly utilise any framework specific code
- The model then sends the image path to the newly loaded preprocess function which should preprocess the img according to the model spec.
- The preprocessed img object is then forwarded to the also newly loaded predict function.
- The backend then checks the metadata to check if it needs to label the predictions or if it will be handled by the predict function.
- The labeled predictions are then returned to the POST handler and returned back to the client.

XLSX/CSV Export

- The export functions take a list of raw predictions and then format them into a XLSX/CSV file.

- The XLSX is slightly different from the CSV because XLSX allows the embedding of images.
  - The XLSX functions are handled by the openPyXL library
  - If the image is not found in the backend, then the image slot is left empty
- Each XLSX/CSV file is saved to disk before being returned to the client app.
- The XLSX/CSV name is a hash of the predictions, this allows the backend to check if the prediction set has been previously exported and if it has, the previous file becomes a cache

Insect Information (GBIF)

- There are a set of optional features which can be enabled by adding GBIF credentials to the .env (GBIF_USER, GBIF_PASSWORD).
- These endpoints allow querying of the GBIF API to find more information about insects detected by the ML models
- Since the frontend queries these APIs quite frequently if the feature is utilised, the backend caches each successful GBIF API call for a day, into a SQL database handled via a SQL alchemy substack (This could also easily be ported to a postGreSQL with a single line of code)

Home API (Local-App Only)

- These are a set of endpoints which serve the customised frontend build (Disabled homepage + all online functionality)
- These include /robots.txt, /favicon.ico which serves their respective files
- The rest of all endpoints is handled by home(), which serves the files saved in the static folder as required by the frontend
  - This API is filtered to supply only .js,.css or .svg files to prevent the leaking of uploaded images, exported XLSX files and other important files

Debugging Interface (Disabled by default)

- For debugging, there is an optional .env variable LOGGING which opens a websocket at /ws/logs which feeds the server logs to clients connected to the websocket for remote debugging without SSH access

Frontend

The frontend is written in React and TypeScript, and the styling is done using Tailwind CSS and DaisyUI. The frontend is hosted on Firebase and the HTTP requests to the backend are done through Axios.

Architecture

- The frontend is split up into several modular chunks, which makes reusing components easier
  - E.g. the navbar is shared between the different pages in the frontend
- The detect page presents a user interface for the user to upload files with
  - The available models are fetched from the backend from /api/v1/available_models/ via Axios
  - Once the user uploads files for inference, a request to the backend is made through Axios.
  - Once the results are returned, the results as well as the model used are sent to Firebase's database if the user is logged in (This is used for user history)
- Users are handled via Firebase Authentication
- User history fetches the saved results from Firebase's Firestore using Firebase Authentication's user as a key
  - The results are saved into a table, which allows the easy use of filters
  - The user history page then fetches the corresponding images from the backend using the /api/v1/get_image endpoint via Axios

Feature Implementation

- Results Export (User History + Detection page)
  - The selected result's JSON is concatenated into a list of JSON objects and is sent to the backend endpoint /api/v1/create_csv or /api/v1/create_xlsx depending on the export button clicked
- Dark Mode:
  - Most UI objects have an additional class (dark: DARK STYLING)
  - This triggers only if the local theme is set to Dark
  - There is also a bypass button, which sets the theme manually
  - We set the localStorage theme to dark or light to store the manual override
  - A dark object is added/removed to the html to trigger the specified theme
  - If the button is not clicked, there is a bypass to set the default theme to the system theme
  - This causes issues with the Ubuntu local application as Linux does not have a webview object, and hence PyWebview uses GTK to create one.
    - This GTK webview is not completely compliant with most browser implementations and most significantly lacks support for local storage which breaks the entire application
    - We bypassed this by creating and setting a flag for the Frontend which disables the manual theme override feature altogether

Local Application

The backend has a set of endpoints (Home API) that are designed purely for the local application.

Architecture

- The GUI is handled by Pywebview
  - Pywebview is a Python package which utilises the Edge browser WebView2 on Windows and uses a lightweight GTK stack on Linux/MacOS
- Pywebview was chosen over Electron as
  - Electron is JavaScript based and linking with a Python backend would become tedious and difficult
  - Pywebview is Python based, which makes it easier to integrate with the Python backend
  - Compared to Electron, Pywebview is very lightweight and takes much less space.
- The CLI is handled by utilising FastAPI's testclient, essentially a wrapper around HTTPX, a lightweight HTTP client for Python.
  - Because of the testclient roots, the CLI can directly bypass Uvicorn and talk directly to FastAPI, which is theoretically more performant

On App Startup

- (GUI) Spawns a Uvicorn ASGI web server hosted on localhost:6789 and also spawns a Pywebview window pointed at localhost:6789/upload
- (CLI) Evaluates the arguments passed to the app and instructs a testClient to execute the instructions passed through the arguments

Build Process of Local Application

1. The frontend is build with certain flags enabled in the .env
    a. REACT_APP_BACKEND_URL is set to the local app: http://localhost:6789/
    b. The Firebase credentials are all set to null
        i. This is not actually necessary, as the Firebase credentials are not private, but they have been set to null as the functionality is unused and we wish to avoid extra bugs
        ii. REACT_APP_DISABLE_NAVBAR is set to true, this disables the navbar, as it is unnecessary, since Account functionality is non-functional
        iii. REACT_APP_DISABLE_UPGRADE_INSECURE_REQUESTS is set to true, as the local API is served over http not HTTPS
2. Secondly, the frontend build is copied over to the backend, where the css, js, robots.txt files are sent to /static/ and index.html is sent to /templates/
3. The virtual environment is then invoked and the requirements are installed if not already
4. The python script is built into an executable for the current platform by pyinstaller with the following flags
    a. These packages are added as necessary packages which are not imported in the backend (keras, PIL, glob, numpy, tensorflow, cv2, torchvision, torch)
    b. The icon is added (windows/macos only)
    c. Clean package, this generally is to avoid bugs
    d. No Confirm, this is avoid user input as the entire build process is 20 mins+
    e. Extra necessary data, (library folder, sql_app.db)
    f. The –hide-console “hide-early” flag is invoked for windows/macos as it is a compromise between windowed and console, as without the flag, the application would either not work with the cmd-line or it would have a terminal hanging around which is quite unsightly and confusing.
5. The application is then bundled either with 7z (portable) or Inno Setup (Installed), renamed and moved to the root of the project
    a. 7z simply zips the folder for the user to unzip and run on their own computer
    b. InnoSetup adds an Icon and compresses the folder using lzma2, we also set up InnoSetup to run extra tasks when the setup is run, this is elaborated below in *Installed local Application (Inno Setup Executable)*

Issues with the Local Application

- Time needed to run the build process
    - Building and debugging the application requires 20 minutes to build the app
- We utilise the PyInstaller flag –hide-console which is meant to hide the console window created by the GUI unless the application is spawned from a console

- This flag is utilised as otherwise the Command Line Interface of the application would not be printed to console correctly
    - On Windows 11, if you set Windows Terminal as your default console handler, the terminal will still show up as Windows Terminal and completely ignores whatever pyinstaller implemented as part of the bootloader process for –hide-console
    - Our workaround is an additional check to see if the application owns the console it is running on
        - As part of the application, this code runs after the PyInstaller bootloader
        - If –hide-console worked, the check would trigger and do nothing
        - If –hide-console failed, the check would trigger and run SW_HIDE on the console
            - Although better than nothing, this workaround still shows a console for a couple of seconds and only minimises the console to the taskbar, as Windows Terminal incorrectly handles SW_HIDE and instead treats it as SW_MINIMISE
        - If the check does not trigger, that means that the console is supposed to run and so the application does nothing and continues with its execution

Local Application Installation

Portable local Application

- Does not require installation

Installed Local Application (Inno Setup Executable)

- A customised setup wizard will show up (courtesy of Inno Setup)
- The application will be installed by default to C:/ProgramFiles (x86)/Code Critters/Crittersleuth/
- The application will create two folders upload and exports in %User%/Appdata
    - These folders have been separated from the rest of the application as Appdata does not require Administrator privileges for write permissions while ProgramFiles (x86) does require Admin privileges.
- Several registry keys are written to the Windows Registry
    - One is a value which denotes that the application is installed
    - One denotes the the Path to the applications (Placeholder)
    - One writes the version number (Planned for Model Expansion use)
    - One places the application into the System PATH for system-wide CLI access

Miscellaneous

Model Expansion Packs

Model expansion packs are setup executables which add more models to an existing local-application without the need of a complete reinstallation of a newer version. They work by taking models which follow the model specification and packaging them with Inno Setup into a setup executable.

- The setup checks if the Registry key is set, which indicates that the application has been installed.
- If not, it will show a error message and cancel the installation
- If true, the models packaged are written to the Application's model folder
- Due to the Application Model implementation, these models are immediately available for use for the application (Provided that the local-application's version supports the model)

Notes

We chose not to support the use of expansion packs for the backend server for the website. It was decided that it was largely redundant as it is relatively easy to add new models to the backend server and that, unlike end-users, server admins can be expected to learn/understand the easy process on how to add/update models in the backend.

Hosting

The backend is self-hosted on a VM with the given specs:

- 16GB of DDR4 3200 ECC ram
- 4 Cores 8 threads from a EPYC 7302p
- 100 GB of HDD storage
- Tesla p4 GPU
- Guest OS: Ubuntu Server 23.04
- Host OS: Truenas Scale

Issues and Limitations

- The VM is not performant enough with 8GB of ram, we even ran out of storage space as it was used for swap space.
- We ran out of space twice before expanding to 100GB of storage, however this is only good enough for the short-term and is not enough storage for a prolonged deployment, as the storage space is nearly full.
    - Although we occasionally have issues with slow storage, most of the HDD storage is likely sped up due to the suspected caching of the disk by ZFS's ram cache in the Host OS.

- We had significant issues with our initial Guest OS: Ubuntu 22.04 LTS due to the lack of native Python 3.11
  - We then switched to Ubuntu Server 23.04 as it has native Python 3.11 support and overall better virtual environment support.
- The Tesla P4 had given quite a few issues as it was difficult to get CUDA drivers working for it on both Ubuntu 22.04 LTS and 23.04, although the latter can be partially attributed to the lack of native NVIDIA Ubuntu 23.04 support.

The hosting setup was originally intended to be hosted in AWS before we hit an issue of a budget overruns of over $400 NZD

- This has been root-caused by SageMaker's Jupyter notebook's EC2 instances which we did not realise were full hosted and dedicated EC2 instances.
  - As we created multiple Jupyter notebooks to get our SageMaker endpoints working, this quickly multiplied to a significant cost

Our initial planned architecture was:

- The primary backend would be hosted in a t2 mini/micro EC2 instance.
  - This would be achieved by either manual setup via git or
  - Setting up via the docker image setup.
- The inference would be done by a p2 medium/large Sagemaker endpoint
  - This was problematic because we needed to retrain the ML model to achieve how SageMaker wants the model to be formatted
- If we were able to continue this path, our development efforts would have been to
  - Connect the primary backend to the SageMaker endpoints using AWS Api Gateway
  - Share the uploads/exports folder to be able to load-balance the backend

RESULTS AND EVALUATION

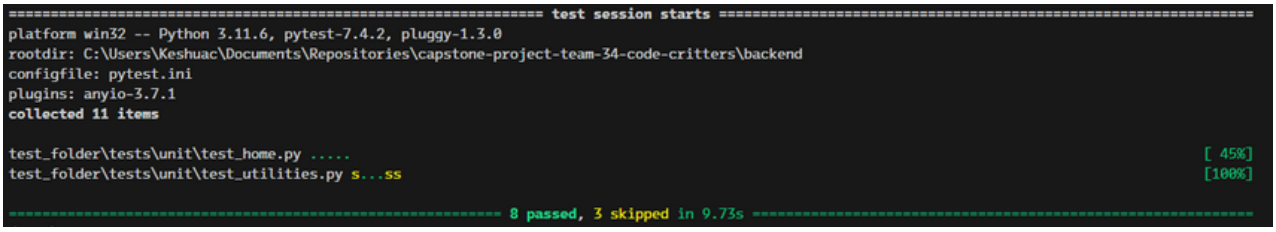
Project Goals

We have effectively achieved our main project goals by delivering a highly functional and user-friendly application tailored to the specific needs of biosecurity professionals. Our primary objective was to create an application that seamlessly integrates image recognition machine learning models and prioritises the ability to upload and export predictions, enhancing the identification of insects crucial for biosecurity efforts in New Zealand. With an emphasis on usability, we aimed to develop an application that encourages users to confidently identify and manage insects, making it a valuable asset in real-world applications. The successful integration of machine learning models ensures that users can trust the system for accurate insect classification, ultimately contributing to the region's biosecurity efforts. Furthermore, our commitment to providing additional features, such as batch query and export support, information display, additional models, user accounts, and comprehensive documentation, has expanded the application's functionality and user-friendliness. Thus, our project has not only met but exceeded its primary objectives, presenting a useful tool to support New Zealand's biosecurity initiatives.

Testing

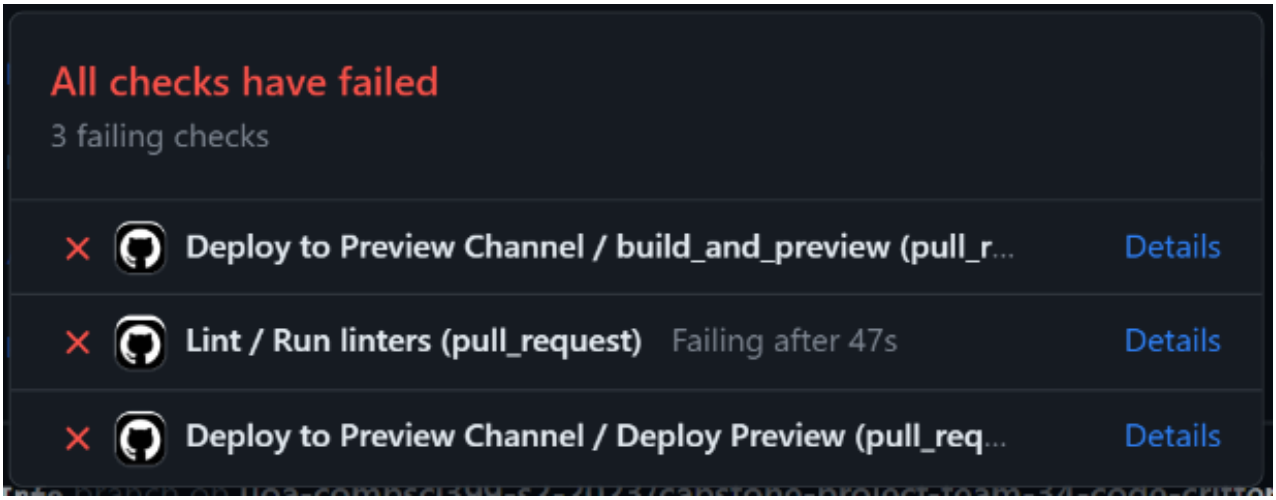
Throughout the development process, we conducted both manual and automated testing at various stages of the project, ensuring the reliability of our application.

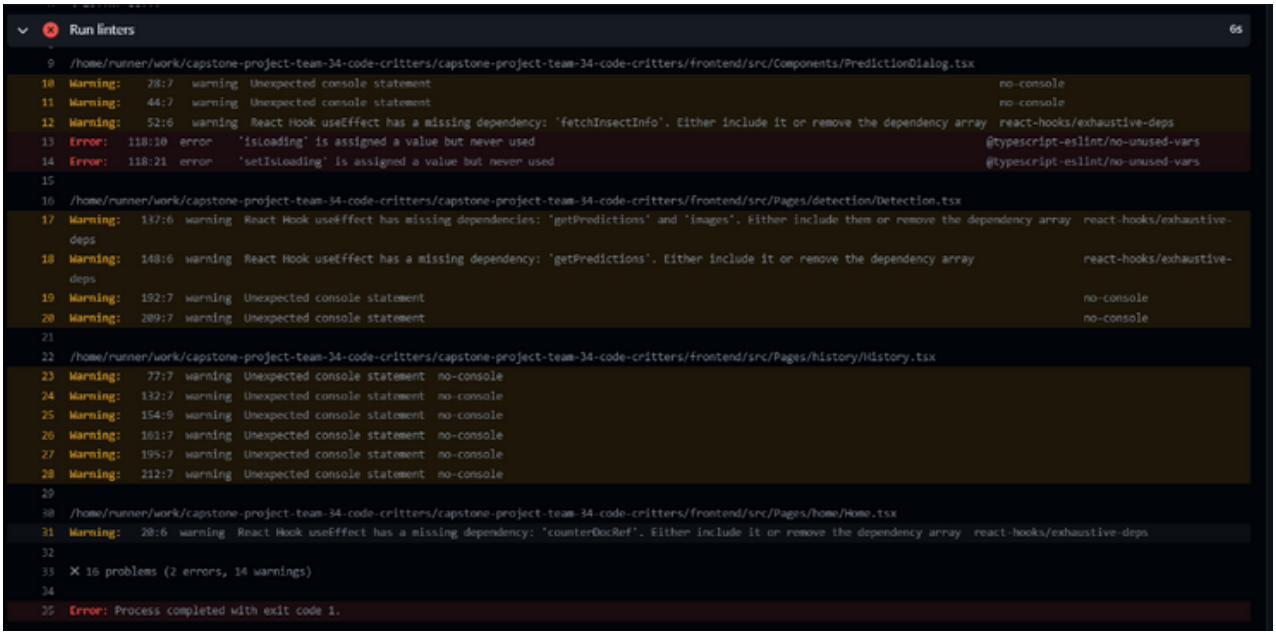
The backend had a limited set of automated tests written for PyTest, which were run before building the application. We conducted unit tests to evaluate the individual components and functions of our application and ensure the reliability and correctness of our backend code. These tests primarily focused on functional testing for different endpoints and API controllers.

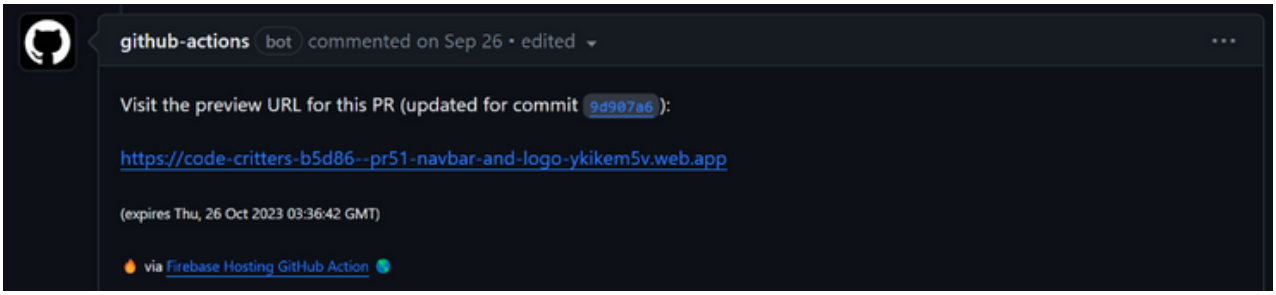


For DevOps testing, we incorporated GitHub Actions into our workflow, enabling a continuous and automated process. Upon creating pull requests, we set up GitHub Actions to run ESLint, ensuring that the code adheres to coding standards and maintains consistency, enhancing code quality. We also deployed the changes to a preview URL hosted on Firebase. This allowed the development team to review and validate the changes before merging them into the main branch.

When we committed our code to GitHub, specifically to our respective feature or component branches, we followed a collaborative approach. We initiated pull requests and had our teammates review and test the code thoroughly. This process helped us identify and rectify any bugs, weaknesses, or opportunities for improvement. The primary objective was to maintain code quality and consistency throughout the project.







During a later stage of development, we conducted user testing with several design students and clients to gain valuable insights into the user interface of our application. In these sessions, the testers interacted with the app and provided us with essential feedback. Specific examples of feedback included difficulties encountered by users in understanding certain UI elements. For instance, some users had trouble deciphering the meaning of certain icons and the labeling of action buttons, recommending the use of more industry standard terms and clearer instructions. An example was given where "select all" was considered more user-friendly than "toggle all." They also noted minor consistency issues, such as variations in the design of similar UI components. These insights raised concerns about our design coherence and comprehensibility and played a crucial role in refining the UI to ensure a more user-friendly and cohesive user experience in CodeCritters.

Strengths

Our application's multi-platform support, including web-based, local, and mobile versions, ensures users can access its insect identification features conveniently on their preferred devices, whether it's a desktop, laptop, or smartphone. This adaptability enhances the user experience and accommodates various usage scenarios.

The backend server allows easy addition of new machine learning models, allowing for model expansion and enhanced versatility and usefulness of our application.

Implementing a DevOps pipeline with GitHub Actions automated deployment and testing processes. This lead to consistent, error-free deployments and faster feedback, ultimately resulting in more efficient development and releases.

The frontend built with React and TypeScript offers a responsive and dynamic user experience. The use of Tailwind CSS and DaisyUI for styling ensures a modern and visually appealing interface. Users can easily interact with the application.

Our user history feature allows users to keep a record of their past interactions and insect identifications, providing quick access to previous results.

Weaknesses

Our current mmh3 hashing algorithm does not provide the highest level of data security. This could potentially lead to vulnerabilities in data protection. Implementing a more secure hashing algorithm would mitigate this weakness, ensuring robust data integrity and user privacy.

The absence of a Firebase user check when exporting data is a potential security gap. Unauthorised access to exported data could compromise user privacy and confidentiality.

One of the weaknesses identified during user testing was the design and user interface. It revealed that our design and overall UI of the application were subpar. The user feedback emphasised issues related to the layout, consistency, and overall aesthetics that made us prioritise UI more.

FUTURE WORK

This section will outline some new features that can be integrated into our project. The following section is split into frontend, backend and deployment improvements.

Frontend

- Implement more detailed insect information features in the frontend
  - This would require the application to pull information about an insect from a database.
- Implement a user profile page
  - This profile page can display relevant information pertaining to a user and could include a summary of their past predictions
- Add support for more languages
- Improve the filtering functionality of the user history table
  - Eg: Search of images using a date / time rage instead of a specific date

Backend

- Change hashing algorithm (mmh3) or make it more secure
- Add Firebase user check to export CSV/XLSX for verification
  - This would prevent unauthorized users from getting access to these predictions
- Add/update automated tests to be more extensive
  - This would speed up the deployment of new features and builds
- Centralise images and models to allow load balancing
  - This would improve the speed and responsiveness of the application

Deployment

- Implement local user history
- Create a MacOS version of the application
- Bundle the Linux application as an AppImage instead of a zip file.
- Add license page to InnoSetup setup
- Integrate portable and installed executables into one executable, which gets differentiated by an flag
- Add command line arguments to the web application
- Get Linux version to build properly
- Make CI/CD pipeline to build the application and add to releases (Hampered by the size restrictions which disallows model.h5)

- Add check to expansion models to check if the currently installed local-application's version supports the expansion model.
- Complete the rewrite the mobile application to utilise CameraAwesome instead of Flutter's default camera library
    - Complete the Real-time Inferencing feature for the mobile application
- Complete the framework necessary to add IOS support to the application
- Add the feature of inference from an existing application to the mobile app
- Add a toggle which allows the switching from local inference to backend server inference
- Add a feature which allows the mobile app to download extra models from the backend server
- Re-introduce the feature which allows switching between different ML models in the mobile application

CONCLUSION

The aim of this project was to design a user-friendly graphical user interface (GUI) tailored to complement machine learning models provided by Landcare Research, allowing users to upload an image, followed by recognition and detection of rare, endangered and invasive insects. The final version of the project fulfills these requirements, featuring batch uploads, the ability to export to CSV/XSLX, fully functional user history to view past predictions.

A key finding was the robust utility of the FastAPI framework for our backend, which handled asynchronous execution, was compatible with the ASGI standard, and provided the convenience of auto-generated swagger-documentation. This enabled dynamic addition of machine learning models without the need for backend modifications or app restarts. On the frontend, we embraced React and TypeScript, ensuring components were modular for efficient reuse. In terms of local applications, our choice of Pywebview over Electron was allowed for seamless integration with our Python backend. Our VM-based hosting did present performance and space challenges, shedding light on optimal system requirements. Additionally, while our initial aim to deploy on AWS seemed promising, we encountered budgetary and model compatibility hurdles with SageMaker.

Another key finding was the effectiveness of working in an agile environment for our project progression. Our consistent communication, facilitated by weekly meetings and the use of an agile board, ensured that we continuously understood and addressed challenges, paving the way for smooth advancement throughout the semester.

The major outcomes for our project encompass a seamlessly deployed local and web application, enabling users to utilise both the GUI and advanced Machine Learning models for precise insect predictions. Moreover, users can create an account to retain their prediction histories, complete with the option of exporting their findings in CSV/XSLX formats. This innovative tool will benefit Landcare Research, offering them a user-friendly platform for rapid insect detection, therefore amplifying their research capabilities and making data accumulation and pest detection more efficient than ever

REFERENCES

Høye, T. T., Ärje, J., Bjerge, K., Hansen, O. L. P., Iosifidis, A., Leese, F., Mann, H. M. R., Meissner, K., Melvad, C., & Raitoharju, J. (2021). Deep learning and computer vision will transform entomology [Review of Deep learning and computer vision will transform entomology]. PROCEEDINGS of the NATIONAL ACADEMY of SCIENCES, 118(2).

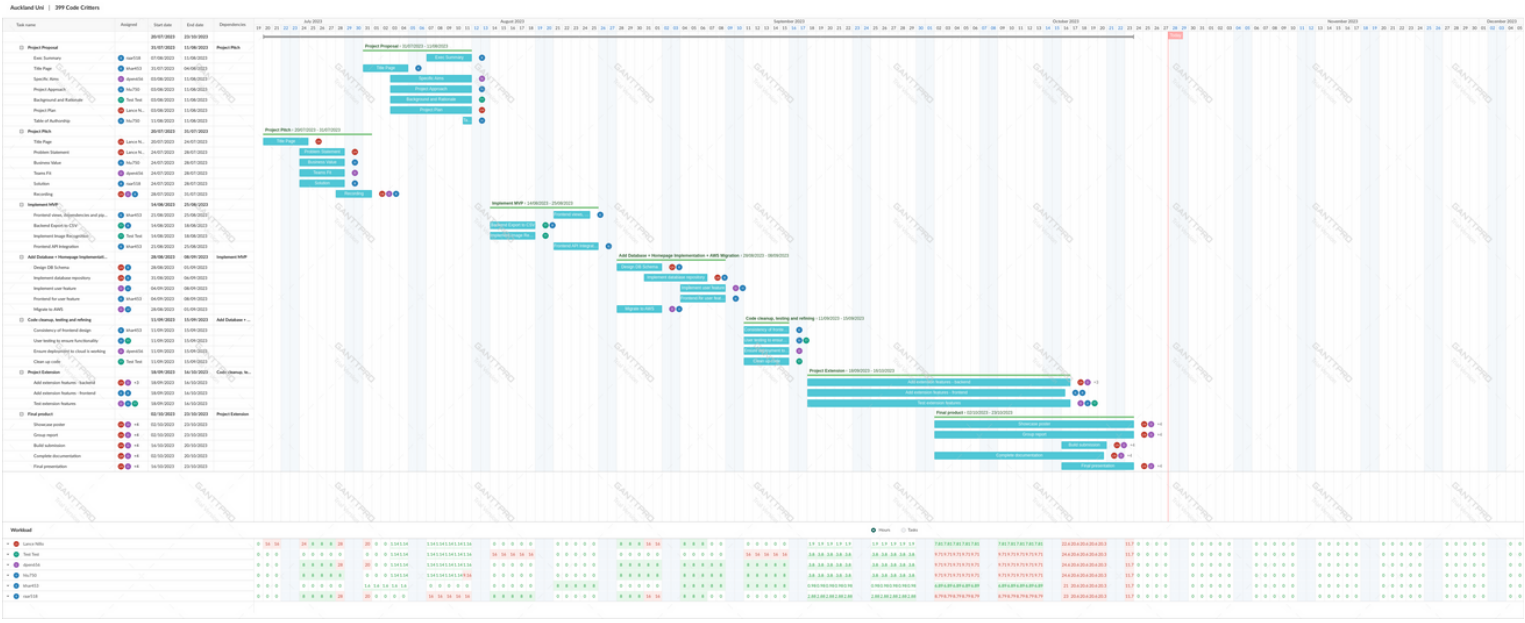
Inaturalist [https://www.inaturalist.org/]
Insect.Id [https://www.kindwise.com/insect-id]

DECLARATION OF AUTHORSHIP

Section	Author(s)	Signature(s)
Introduction	Lance	L.N
Executive Summary	Rukun	R.A
Background	Joshua, Lance	J.C, L.N
Project Specification	Dhiraj	D.P
Project Design	Kei, Joshua	K.H, J.C
Project Implementation	Joshua	J.C
Results and Evaluation	Helen	H.L
Future Work	Rukun, Lance	L.N, R.A
Conclusion	Lance	L.N

APPENDICES

Gantt Chart



Link to Website

https://codecritters.live/