Implementation & Testing – App Development Phase 1

Now that the substantial model development process has been completed, the next task is to create an app which can house the model and allow it to be deployed and used in real-world scenarios. Various approaches could be taken to create this. For one, a full mobile application could be created, using either native technologies such as Swift or Kotlin, or cross-platform frameworks such as Flutter or React Native. Alternatively, a desktop application could be created, like a Windows *WinForm* application using C# and *.NET*, or a MacOS application using Swift. Or finally, a web-based application could be created using any of the many available technologies, such as HTML and CSS for a simple front-end design, or React.js or Vue.js for more dynamism, and then for the back-end, Node.js, Django or Flask would be suitable tools.

Initial experimentation was performed with Kotlin and Android studio to create a native mobile application, as well as with Dart and Flutter to explore the possibility of creating a cross-platform application, with TensorFlow integrated seeming highly plausible due to both these frameworks being supported by Google, just like TensorFlow itself. Various tutorials were completed, primarily from the Google and Android Developer services, however the complexity and learning process were daunting, and deemed excessive for a simple application such as this. Therefore, the decision was made to explore the creation of a web-based application, as this approach would be relatively simple and efficient yet completely appropriate for the task at hand. The technologies targeted for use are HTML and CSS for the front-end due to their simplicity and familiarity, as well as Flask for the back-end due to its light weight, flexibility, and ease of TensorFlow integration.

Additional important technologies will be a hosting service such as Google Cloud Run, as well as a containerisation service such as Docker. The former will facilitate a smooth deployment process and provide scalability and reliability, while the latter will package the entire application into a light-weight container, including the front-end, back-end, model and all required tools and libraries, ensuring consistency between devices.

Wireframe

The figure below is a wireframe depicting the desired design of the web application. It features a text box containing an appropriate title; two buttons, one for uploading an image and another for requesting a prediction; a section to display the most recently uploaded image; and a text box at the bottom containing the model’s predicted label alongside its confidence percentage.

A screenshot of a phone

AI-generated content may be incorrect.

Although rather basic in its design, this wireframe contains all essential components, while maintaining a clean and elegant aesthetic.

Front-End

As previously stated, the front-end design is intentionally kept minimal and clean while ensuring all necessary functionality is incorporated, using HTML and CSS. The purpose of the front-end code is to provide the user with an interactive, comprehensible interface through which they can obtain predictions on uploaded flower imagery.

During operation, the front-end accepts the user’s image input and sends it to the back-end, where it is classified by the trained TensorFlow model. Once processed, the model’s prediction and confidence percentage are returned and displayed via the front-end. Additionally, the uploaded image is shown to allow the user to verify that the correct image has been classified.

The first key section of the front-end code is the snippet shown in the figure below, which specifies the document type and language used. The <title> tag at the bottom of this section sets the text that appears on the browser tab when the web application is running.

A black screen with white text

AI-generated content may be incorrect.

Following this, the next important section is the CSS styling block, responsible for the visual appearance of all UI elements. The upper body styling area defines the background colour, font, text alignment, and padding, ensuring consistency across the application. The .container section specifies details of the central content box, which houses all core UI features, including the header (h2), the upload form, buttons, and the image container. This section is primarily responsible for the clean and user-friendly appearance of the interface.

A screen shot of a computer program

AI-generated content may be incorrect.

Next is the header section, containing the visible title displayed at the top of the web application. This header provides clear instructions, ensuring that users understand the purpose of the application at first glance.

A black background with white text

AI-generated content may be incorrect.

The following section of vital importance is the image upload form, which handles the receipt of the user’s uploaded image. In the opening <form> tag, the action="/" attribute sends the data to the back-end’s root route, the method="post" attribute specifies that the form will use the HTTP POST method to transmit the file securely, and enctype="multipart/form-data" ensures that the binary file data is properly encoded. The next line contains <input type="file" name="file" required>, where type="file" creates a file picker button in the UI, allowing the user to select an image from their device. The name="file" attribute is essential because it sets the key name used by the back-end to retrieve the uploaded file. Including the required attribute prevents form submission without a file, thereby improving usability by enforcing correct input. The final line within the form is the <button type="submit">Predict</button> element, which creates the Predict button. When pressed, this button triggers the submission of the form and the selected image, sending a POST request to the back-end as defined earlier. The back-end processes this request, passing the image to the TensorFlow model to generate a classification result.

A black screen with white text

AI-generated content may be incorrect.

The final section of significance is responsible for displaying the classification result once the model has processed the uploaded image. This block is conditionally displayed only after a prediction has been made by the back-end. It dynamically shows the uploaded image, the predicted class label, and the confidence percentage. These features ensure that the user not only sees the classification result but can also verify the image they uploaded and view the model’s confidence in its prediction.

A computer code with text on it

AI-generated content may be incorrect.

Back-End

The back-end of the application is implemented using Flask, a lightweight Python-based web framework, and is responsible for handling the core functionality of the system. Specifically, it manages image uploads from the user, processes the uploaded image using the trained TensorFlow model, and returns the resultant prediction and confidence score to the front-end. The back-end ensures smooth interaction between the user interface and the machine learning model, allowing the system to provide accurate, real-time classifications.

The first important section of the back-end code is the list of imported dependencies, as shown in Figure 4.1 below. This section imports several essential libraries, including Flask and its *render\_template* function, which allows the application to render dynamic HTML pages. TensorFlow and NumPy are imported to load the pre-trained model and process numerical data, while PIL (Python Imaging Library) is imported for image processing tasks. Additionally, *secure\_filename* from the werkzeug.utils library is imported to safely handle file names when saving uploaded images.

*A screen shot of a computer

AI-generated content may be incorrect.*

*Insert Figure 4.1 – Imported Dependencies and Libraries*

The next key section of the back-end, shown in Figure 4.2, involves the initialization and configuration of the Flask application. Here, the Flask app is initialized, and the UPLOAD\_FOLDER variable is defined to specify the directory where uploaded images will be stored. The folder is created if it does not already exist, ensuring the application can store incoming files. The ALLOWED\_EXTENSIONS set is also defined, listing the supported image formats (PNG, JPG, JPEG) that the system will accept for classification.

*A computer screen shot of a program code

AI-generated content may be incorrect.*

*Insert Figure 4.2 – Flask Initialization and Upload Configuration*

Following this, the TensorFlow model is loaded, as illustrated in Figure 4.3. The trained model is loaded from the specified file path using TensorFlow’s load\_model function. Alongside this, a list of class names is defined, corresponding to the output indices of the model. This allows the system to map the model’s numeric output to a human-readable class label, which will later be displayed to the user.

*A screen shot of a computer code

AI-generated content may be incorrect.*

*Insert Figure 4.3 – Model Loading and Class Name Definitions*

The next section of significance is the file validation function, shown in Figure 4.4. The allowed\_file() function checks the uploaded file’s extension to ensure it matches one of the supported image formats defined earlier. This step is crucial in preventing invalid files from being processed and improves system reliability and security.

**

*Insert Figure 4.4 – Allowed File Type Check Function*

The image preprocessing function, presented in Figure 4.5, is responsible for preparing the uploaded image to match the input requirements of the TensorFlow model. This function converts the image to RGB format, resizes it to 224x224 pixels, normalizes the pixel values, and adds a batch dimension. Proper preprocessing ensures that the model receives input in the same format as it was trained on, which is essential for achieving accurate predictions.

*A screen shot of a computer code

AI-generated content may be incorrect.*

*Insert Figure 4.5 – Image Preprocessing Function*

A simple health check route is also included in the back-end, as seen in Figure 4.6. This route returns an HTTP 200 response when accessed, indicating that the server is running correctly. While not directly related to user-facing features, it plays an important role in deployment, particularly on platforms like Google Cloud, where health monitoring is essential.

*A black background with white text

AI-generated content may be incorrect.*

*Insert Figure 4.6 – Health Check Endpoint*

The main route definition of the application is shown in Figure 4.7. This route handles both GET and POST requests sent to the root URL. When the user accesses the application initially, a GET request is made, prompting the back-end to render the front-end interface. When the user submits an image, a POST request is sent, triggering the subsequent processing steps.

**

*Insert Figure 4.7 – Main Route Definition*

The next section, displayed in Figure 4.8, handles the file upload process. The system first checks if a file has been submitted and validates its format using the previously defined allowed\_file() function. If the file is valid, it is safely saved to the designated upload folder using a secure filename. If no file is submitted, or if the file is invalid, an appropriate error message is rendered back to the front-end.

*A computer screen with text

AI-generated content may be incorrect.*

*Insert Figure 4.8 – File Upload Handling and Validation*

Following this, the model prediction process is conducted, as demonstrated in Figure 4.9. The uploaded image is preprocessed, passed through the TensorFlow model, and the resulting prediction is obtained. The model’s output is processed to extract the predicted class index and the associated confidence score, which is converted into a percentage for display purposes. The class index is then mapped to the corresponding species name from the predefined list.

*A computer code on a black background

AI-generated content may be incorrect.*

*Insert Figure 4.9 – Model Prediction and Result Extraction*

Once the prediction has been completed, the system dynamically renders the front-end HTML page, passing the prediction result, confidence score, and filename of the uploaded image back to the user interface. This is achieved through Flask’s render\_template function, as shown in Figure 4.10. This step ensures that the front-end can display the results to the user clearly and accurately.

**

*Insert Figure 4.10 – Rendering Prediction Results to Front-End*

In the case of a GET request, or if no valid image has been uploaded, the system simply renders the front-end page without any prediction data, as illustrated in Figure 4.11. This allows the user to view the interface and upload an image when desired.



*Insert Figure 4.11 – Rendering Initial Front-End Page*

The final section of the back-end, shown in Figure 4.12, involves starting the Flask web server. The application is configured to listen on a dynamic port, which is particularly useful for cloud-based deployments, and runs on all available IP addresses to ensure accessibility.

A black background with white text

AI-generated content may be incorrect.

*Insert Figure 4.12 – Starting the Flask Web Server*

In summary, the back-end handles all essential functionality behind the scenes, including validating user input, preprocessing images, executing the TensorFlow model, and returning dynamic results to the front-end interface. Its clean and efficient design allows for seamless interaction between the user and the classification model, ensuring a smooth user experience throughout the application.

Containerisation

To ensure the application can be deployed and run consistently across different environments, containerisation is utilised through Docker. Docker allows the entire application, including its dependencies and runtime environment, to be packaged into a single, lightweight container image. This eliminates potential issues arising from differing system configurations and simplifies deployment.

The container is built based on the configuration defined in the Dockerfile, shown in *Figure Y.1*. The Dockerfile begins by specifying an official lightweight Python 3.9 base image. It then sets the working directory within the container to /app and copies all necessary files from the current directory into this directory. Following this, it installs the application's dependencies using pip and the requirements.txt file. Port 8080 is exposed to allow communication with the Flask web server. Finally, the application is configured to run using Gunicorn, a production-ready WSGI HTTP server, which serves the Flask app on all available network interfaces at port 8080.

*A screenshot of a computer program

AI-generated content may be incorrect.*

*Insert Figure Y.1 – Dockerfile Configuration*

To further optimise the containerisation process, a .dockerignore file is included, shown in *Figure Y.2*. This file specifies files and directories that should be excluded from the Docker image, such as Python cache files, virtual environments, environment files, and version control metadata. By excluding unnecessary files, the build context is kept lightweight, resulting in a smaller, more efficient container image.

*A black screen with white text

AI-generated content may be incorrect.*

*Insert Figure Y.2 – .dockerignore Configuration*

Through the use of Docker, the application can be reliably built, tested, and deployed in a reproducible manner, ensuring consistency across different environments.

Hosting

For hosting the containerised application, Google Cloud Run is employed. Google Cloud Run is a fully managed serverless platform that enables the deployment of containerised applications without the need to manage infrastructure.

The container image, built locally using Docker, is pushed to Google Container Registry or Artifact Registry. From there, Google Cloud Run is used to deploy the container, allowing the application to scale automatically in response to incoming traffic. The platform handles all aspects of provisioning, managing, and scaling the infrastructure, ensuring high availability and minimal operational overhead.

Additionally, Cloud Run supports dynamic port assignment, which is accommodated within the Flask back-end code by retrieving the port from the environment variable PORT, defaulting to 8080 if not specified. This seamless integration ensures that the application runs correctly within the Cloud Run environment.

By leveraging Google Cloud Run, the application benefits from scalability, security, and reliability, while minimising the complexity involved in managing servers or infrastructure manually.

Version 1

Below is the initial application design. As stated previously, although it is basic, it fulfils all requirements, and maintains a clean and simple aesthetic. Possible improvements will be contemplated in order to enhance the design further, although the fundamental requirements have already been met.

A close up of a yellow flower

AI-generated content may be incorrect.

Implementation & Testing – App Development Phase 2

Improvements

Following the successful deployment of version 1, some consideration went into determining what enhancements could be implemented. The figure below is the wireframe, updated to include the new front-end additions:

A screenshot of a phone

AI-generated content may be incorrect.

Firstly, it has been deemed necessary to include a title, for clarity and aesthetic improvement. ‘Flower Power’ has been contemplated as a name for the application, and so the decision has been made to use this for the title. To resonate with the floral theme of the application, the ‘O’s will be replaced with a flower PNG created using Canva. The other less significant addition is the inclusion of my name and student number at the base of the application.

A purple and yellow flower

AI-generated content may be incorrect.

Regarding the back-end, there are two changes to make. Firstly, the list of accepted file types will be expanded to enable the support of another popular image file type, WEBP. This implementation is as simple as adding ‘WEBP’ to the set of file types stored within the ALLOWED\_EXTENSIONS variable.



The other implementation in the back-end will be some additional fundamental error handling. Already implemented within the allowed\_file function is the functionality to inform the user in the event that they request a prediction without uploading an image. The other necessary error to handle is when a user attempts to submit an invalid file type. In version 1, the user would not be informed that the file they attempted to submit was of an invalid type, the application simply would not work. In version 2, if they attempt to submit a TXT file for example, a message will appear, providing clear explanation to the user which file types are permissible. Following this implementation, a fundamental yet comprehensive level of error handling will be in place.

Version 2

Below is the improved web application. The new title captures attention and adds colour, while the addition of the details at the bottom credits the creator.

A screenshot of a flower

AI-generated content may be incorrect.