40005: iOS Application for Accessing Air Quality Prediction - PolluteChecker

Duong Anh Tran

Student ID: 24775456

University of Technology Sydney

40005: Advanced iOS Development

Supervisor/Tutor: Firas Al-Doghman

3 October 2025

Table of Contents

I. Application introduction and Information:

PolluteChecker is a seamless application that is created to allow searching and accessing same-day air quality forecast. The main objective of the app is to create a portable information viewer that raises users' awareness of health problems that can be caused by poor air quality. The app provides predictions on air quality to encourage users to actively protect their own health, especially during the time where air pollution has become a problem that can cause severe health issues.

1. Application Information and Dependencies:

PolluteChecker is created to mainly support the iOS platform. iPadOS devices can also use the app but will not receive the best experience compared to iOS devices. The application is created using Swift as the programming language and SwiftUI to deliver the user interface.

The minimum deployment of this program is iOS 18.6. However, iOS 17 devices will be able to use the application but will experience disruption due to version incompatibility. Any iOS version from iOS 16 and below will not be able to run this application

PolluteChecker does utilise information from API to address its objective. The API that this app utilise is Open-Meteo Air Quality API, which can be accessed through the following links:

- Documents: https://open-meteo.com/en/docs/air-quality-api
- Official GitHub: https://github.com/open-meteo/open-meteo

Notes: This app mainly utilise the SDK version of this API (even though it allows normal JSON fetching as an alternate method). This is imported directly using SwiftUI Package Dependencies. The URL for the Swift SDK version: https://github.com/open-meteo/sdk.git

Requirements: The application needs to be connected with internet all the time in order to work. This is due to the fact that almost all air quality forecast related information is fetched directly from the API. Without internet connection, user can possibly use map search, save, modify, delete functionalities, but won't be able to view air quality forecast.

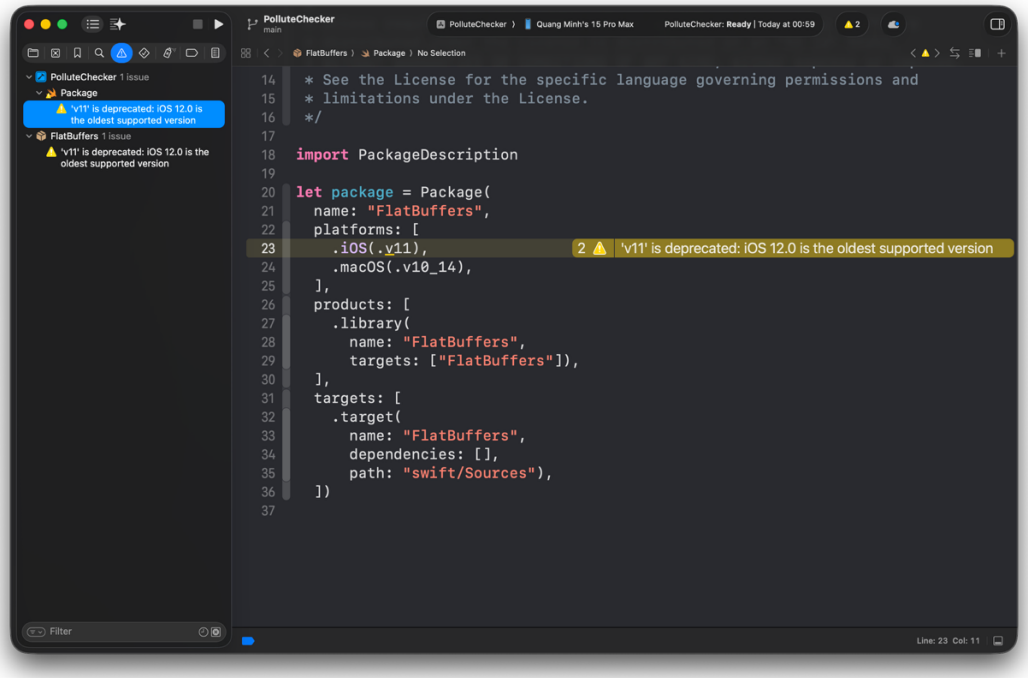
II. How to run the app:

Prior to initiating, building and running PolluteChecker, users will be required to access the GitHub page of the app and clone the repository to access the most recent version of the application. The link to the repository is as follow: PolluteChecker. After this is done, users can now open the project on Xcode to initiate and start using the application. The full installation can be described in steps as below:

Installation Steps:

1. Clone the repository using this command (in terminal): git clone https://github.com/DuongAnhTran/PolluteChecker.git
2. Go to the directory where the project is clone and open the project in XCode
3. Change application's signing in the project's settings (located in Signing and Capabilities) based on intended use (This is only required if the app is going to be used on an actual device)
4. Build and run the application

Note: This application utilise Open-Meteo API Swift SDK package (v11), which can be found at using the following URL: https://github.com/open-meteo/sdk.git. Normally, the package will also be attached with the application when cloned into local directory. However, if the application is missing the package, please use Xcode Package Management feature to add this package. Additionally, since the API is updated too recently, the app will still use the older version of the package. Please check the package version (through **Package.swift** file in the package) and ensure that it is similar with the following screenshot:



III. Features Overview:

In terms of the current functionalities of the app, there are 3 main functions that were created to address the aim of providing an air quality forecast application for the iOS platform. The functionalities are as below:

1. Searching for location and display search result in a map:

User can search for their desired location that they want to check air quality forecast by using the three search options provided by the application: **Query**, **Coordinate** and **Current Location**

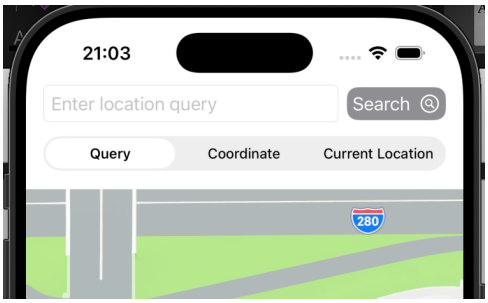


Figure 1: All available search options

- **Query:** User can search using a prompt, which can present specific information such as addresses and cities. However, this search option also supports vague prompts (e.g. “Officeworks Pitt Street”). The result will be the most relevant item that is output after the search.

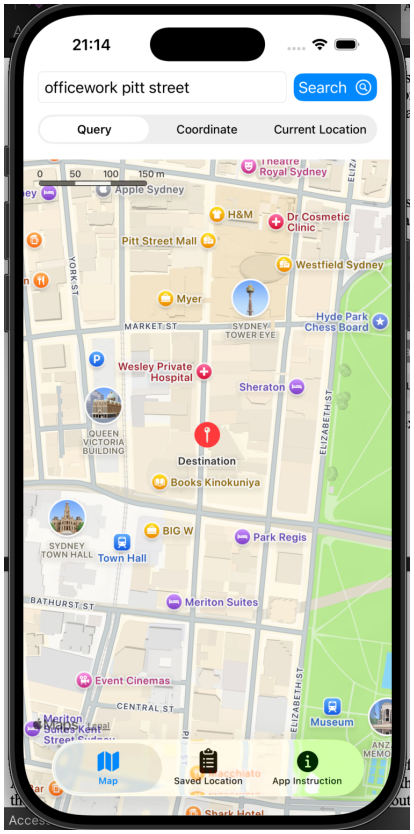


Figure 2: Example search and result location pin (red)

- **Coordinate:** This is a search option that provide 2 different text fields, which are used to get latitude and longitude input information from user. The search result will be the location with the given coordinate. However, for extreme values such as latitude > 80 and longitude = 180, users will be informed of extra guidance to find the location pin as this is an unusual case where there is not much support is provided.

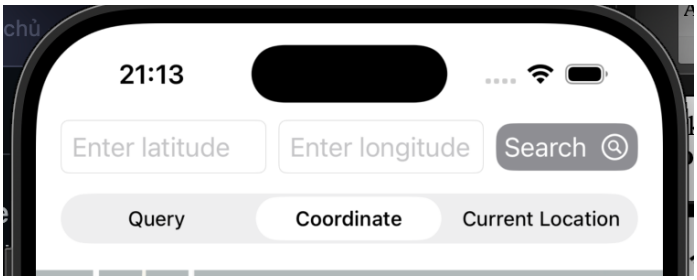


Figure**: Coordinate search input text field

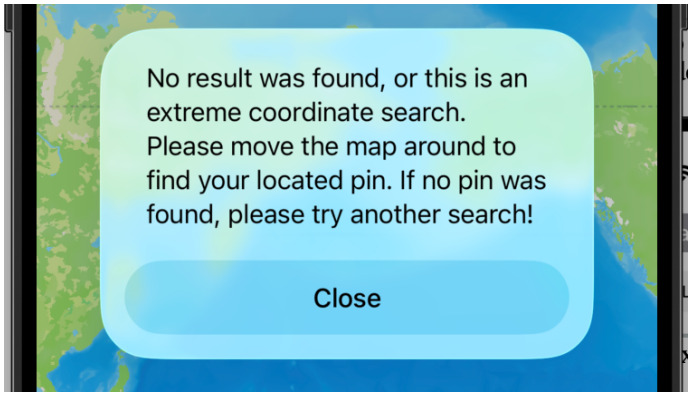


Figure 3: Alert example for extreme coordination search (lat: 90, lon: 180). This will also happen if the app is unable to find a result for the search query/coordinate

- **Current Location:** This search option will only be available if user allow location access whenever the app is used. This will provide a pin showing user's current location, allow quick access to air quality information.

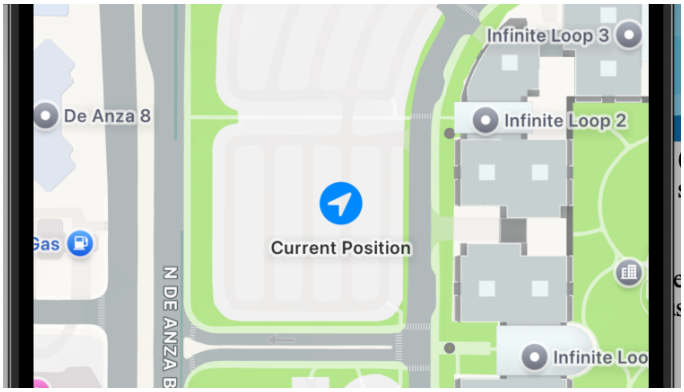


Figure 4: Example of a "Current Location" pin (blue)

2. View location's geological information and air quality forecast for the current day

After getting the result for the location search, user can tap on the map pin that demonstrate the location to get access to a view filled with information about the air quality prediction in the day (from 00:00 to 23:00 of the current day).

The view will show graphs, minimum, average and maximum values of multiple elements that can persist in air and determine if it is safe or suitable for outdoor activities or not. These information and explanation will also be included in the app, at the end of the data view.

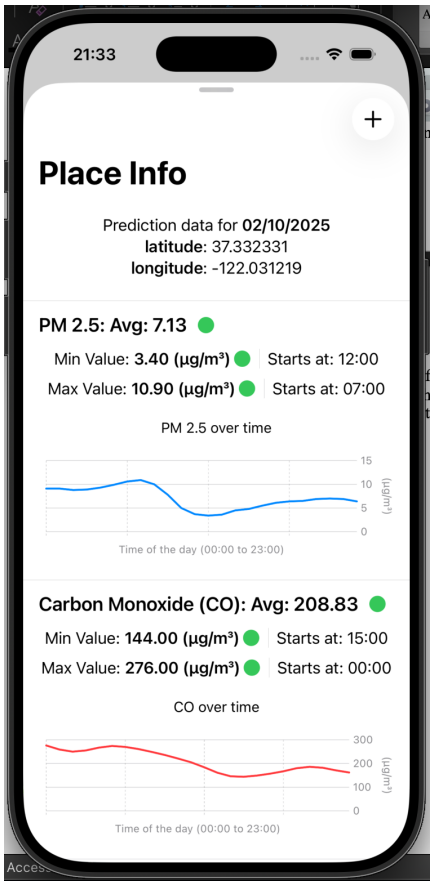


Figure 5: Example of air quality forecast information view

3. Management of saved locations

User can also decide to save their favourite location using the add button ("+"), which is located at the top-right of the air quality forecast information view in Figure 5. The saved information can then be viewed from "**Saved Location**".

A textbox will pop up to ask for the user's input of the location name, which is shown in the figure below:

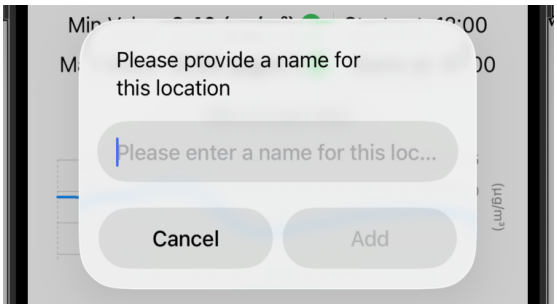


Figure 6: The textbox asking for the name of the location when trying to add a new location

The saved location is displayed in a list, where each location is one list item. There will also be a map preview to give user a better comprehension of where the actual location is on the map. Filter is available for user to search for their location without needing to scroll through their list of saved location, and delete saved location can also be done by swiping that list item to the left.

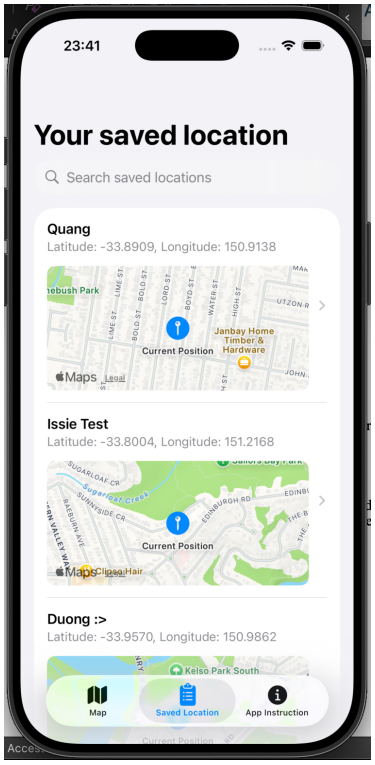


Figure 7: An example of the view with some existing location saved

Accessing the saved location will also provide the same data view. However, there will be a modify button on the top-right instead. When clicked, this button will ask for a new location name that the user want to put in for this location as a modification (Since most of the data related to the location is predetermined by the API and the MapKit Coordination, users can't change those).

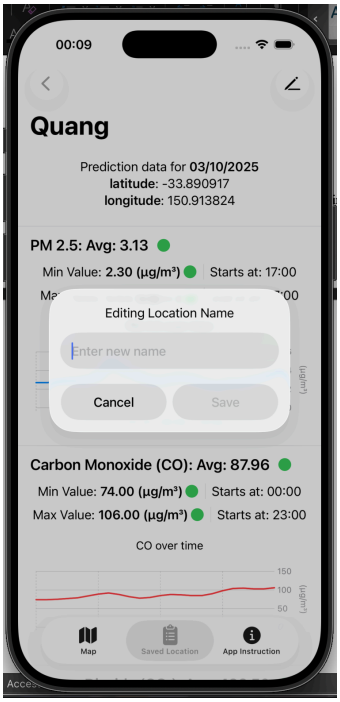


Figure 8: Location data view when accessed from saved location and prompt shows when user click on modify button

IV. Standard Application Flow:

In terms of the standard application flow, the figure below will illustrate the progress of the app (assuming this is the first time the user opens the application). The path in red shows the progress from the starting screen to processes such as searching for location, access location's air quality forecast data, perform save location, access saved data and saved data modification.

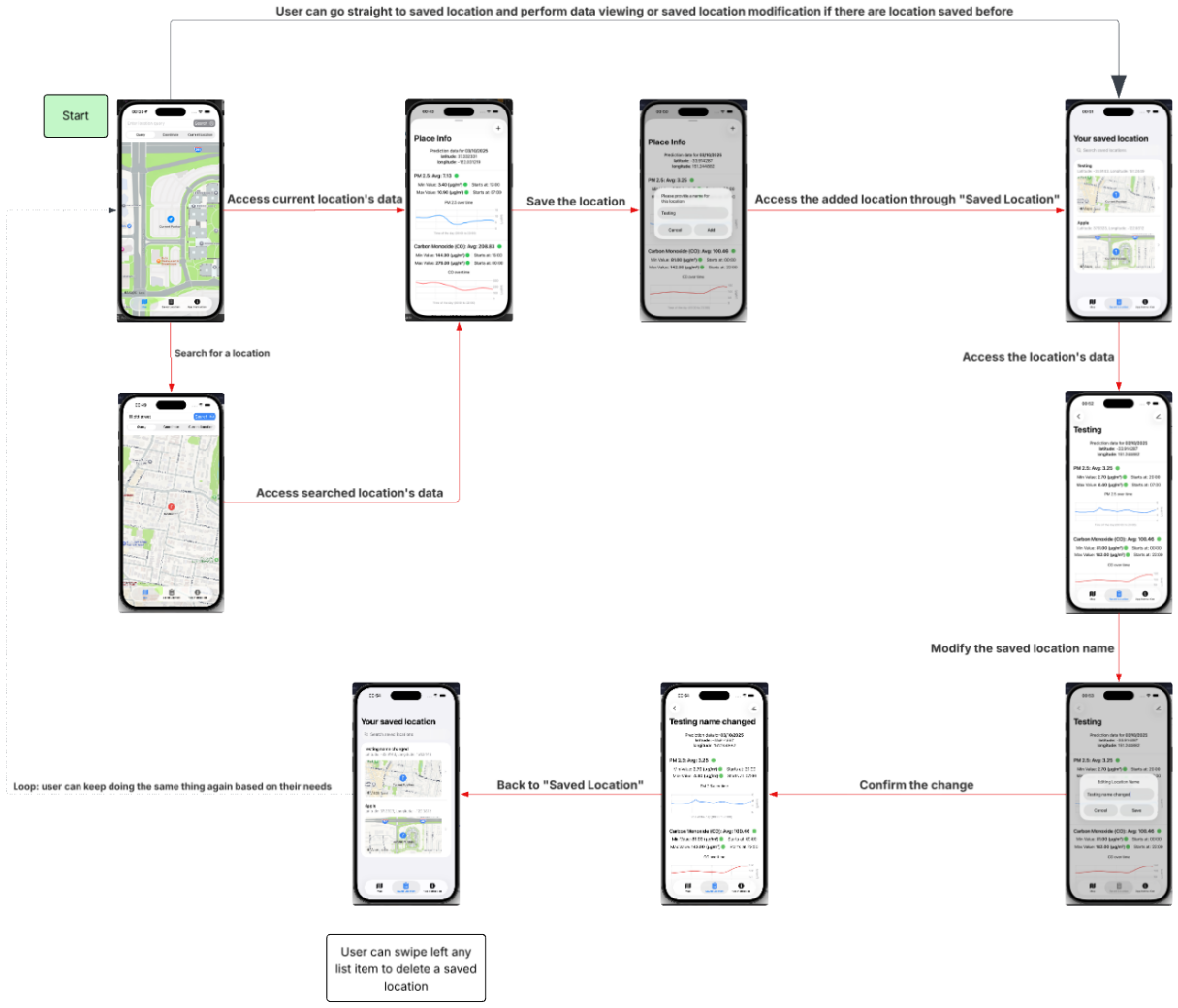


Figure 9: The screen progression of the app. The red-line-path is the standard action sequences, black-line-path show alternative route(s)

Note: Users can follow Figure 9 to start using and explore the functionalities of the application. However, the app does not require a certain fixed sequence of action to properly functioning. The standard application flow is just an example that can be used to assist users in comprehending and understanding the application. More information can be found ion section

V. Error Handling:

In order to ensure that the app works normally without any flaws and disruption to user experience, error handling strategies were implemented into the application during the development process. The following section will explain some of the main strategies that are utilised in the back end of PolluteChecker to prevent crash and unexpected error.

- **Safe unwrap procedures and practices:**
  PolluteChecker does operate with optional values, which requires the program to perform unwrapping when parsing data from a view/process to another view/process. All unwrapping processes were implicitly done using either "**if-let statement block**", "**guard let blocks**" or **nil coalescing** to prevent nil values from appearing in the data. Some examples of these error handling techniques are shown below:

```
//Assigning the published variable with the found data from the API
//The API will be returning data in a list of array, each position is corresponding to each of the fetched data
weatherData = WeatherData(
    hourly: .init(
        time: hourly.getDateTime(offset: offset),
        pm25: hourly.variables(at: 0)?.values ?? [],
        carbonMonoxide: hourly.variables(at: 1)?.values ?? [],
        carbonDioxide: hourly.variables(at: 2)?.values ?? [],
        nitrogenDioxide: hourly.variables(at: 3)?.values ?? [],
        sulphurDioxide: hourly.variables(at: 4)?.values ?? [],
        ozone: hourly.variables(at: 5)?.values ?? [],
        pm10: hourly.variables(at: 6)?.values ?? []
    )
)
```

Figure 10: Example of nil coalescing used in gathering data from the API (**APIFetcher.swift**)

```
//Unwrap the coordinate from the response and set new camera position and create corresponding location pin
search.start { response, error in
    guard let coordinate = response?.mapItems.first?.placemark.coordinate else { return }
    DispatchQueue.main.async {
        self.locationPin = LocationPin(coordinate: coordinate)
        self.camPos = .camera(MapCamera(centerCoordinate: coordinate, distance: 1000))
    }
}
```

Figure 11: Example of "guard let block" to unwrap coordinate of a search response

- **Notice and alert when searching for extreme values that could harm the performance of the application**
  Due to the limitation of normal Swift UI's Map Kit when displaying extreme coordination, which could raise unexpected error or inconvenience, PolluteChecker ensures to explicitly notice users of this limitation (This was mentioned in **section III – Figure 3**). The application will still try to display the location pin, but it is not guaranteed.

- **Disable/Unengaging UI elements when the input is invalid or not suitable**
  In PolluteChecker, UI elements such as text fields are crucial for collecting information. However, not every text field accepts the same data type. An example of this is the two text

fields used for coordination search function. Those elements do collect input from users, but they can only process numerical information. Therefore, to ensure that when searching using coordinate, the input for coordinate is valid, a restriction was created. The search button for coordinate search will change colour to indicate if the information is valid or not, where grey is not valid and blue is valid. Figures 12 and 13 will show the different response of this button when the text fields receive valid/invalid input information

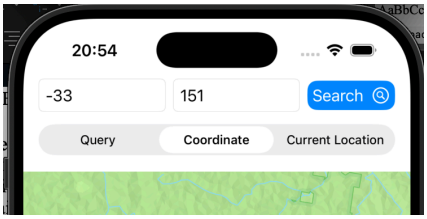


Figure 12: Coordinate search when entering the valid input

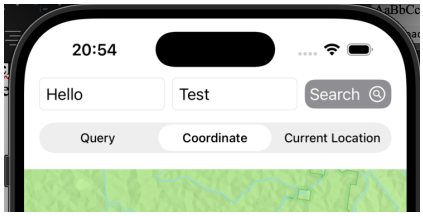


Figure 13: Coordinate search when entering invalid input

```
//The search button for the coordinate search
Button(action: {
    //Call corresponding search from MapSearch
    mapSearch.coorSearch(lat: queryLat, lon: queryLon)
    isCurrentLoc = false
    //Dismiss keyboard
    UIApplication.shared.sendAction(#selector(UIResponder.resignFirstResponder), to: nil, from: nil, for: nil)
    //Show alert if extreme location coordination is used to search
    if mapSearch.locationPin == nil || mapSearch.locationPin?.coordinate.latitude ?? 86 >= 85 ||
        mapSearch.locationPin?.coordinate.latitude ?? 86 <= -85 || mapSearch.locationPin?.coordinate.longitude ??
        181 > 180 || mapSearch.locationPin?.coordinate.longitude ?? 181 < -180 {
        isAlert = true
    }
}) {
    //Label for the button
    HStack {
        Text("Search")
        Image(systemName: "magnifyingglass.circle")
    }
    .padding(5)
}
//Disable if the coordinate input is not valid, gray button for invalid input, blue for valid input
.disabled(!isValidCoor())
.foregroundStyle(.white)
.background(isValidCoor() ? Color.blue : Color.gray)
.cornerRadius(10)
```

Figure 14: The backend of this error handling feature

This approach was done not only for coordinate search, but also for location saving, location name modification, etc. Through the use of disabling UI elements when the conditions do not satisfy, the application can explicitly handle and prevent unexpected events from happening without manually implementing a backend procedure.