**Group: TeenTitans**

**Overall Summary and Achievement:**

Flutter and Rust app. Flutter is used for front end and Rust is used for back end processing. This is a very ambitious app – Bluetooth connections are notoriously tricky – and the model of Flutter+Rust is also quite ambitious, but the code responsibilities are appropriate and mostly well written. Some improvements could be made in some of the readability and ease of understanding the Dart codebase by refactoring HomeScreenState to be broken into small sections, and improvements to robustness by handling errors in a more user-friendly way. As this app handles quite a few things that can hard-error – eg Bluetooth and file opening – handling these in a user-friendly way is important.

**User Interface**

Intuitive and efficient: Generally it looks nice! The No Devices Found page is a bit stark

Looks professional?: Looks nice.

**Code Design**

Project Structure: Good – Dart code is broken into rational folders.

Modularity: Good – see above. As well as AppColors, Further style definitions could be extracted from the code so that the look and feel could be easily updated – eg AppTextStyles

Maintainability: It was fairly easy to find which file contained which code for maintenance and updates.

**Code quality**

Efficiency: The file reading doesn’t seem to work. This could be because FilePicker.platform.pickFiles().then()… is asynchronous, but is used as though it is synchronous and will wait for the file to be chosen before continuing, so the code continues to try and read the file even though its name hasn’t been chosen yet. Instead use FilePickerResult? result = await FilePicker.*platform*.pickFiles(); and use onPressed: () async. Also there are no error messages sent back to the user – eg if DataHandler.readDataCsv(fileDirectory: filename).then’ fails, then the code carries on regardless. Exceptions just return the app to the previous page without letting the user know that something is wrong, and the same for Bluetooth connectivity failures.

Error handling: Exceptions are mostly caught but errors are not reported to the user. If exceptions happen in code where this is quite likely – eg in file loading, they are swallowed and it appears to the user as if nothing happened in response to their clicking on a button. The situations where panics are invoked could also be handled more gracefully.

I get the following error on the graph page in the emulator: “Bottom overflowed by 57 pixels.”



Run-time logging: No logging seen for connecting to Bluetooth, file loading, or Rust calls. You could use the logger package. This is to help debugging errors in retrospect. Also use the log crate in Rust.

Method length: Many build() methods (e.g., HomeScreen, LivePlotPage) exceed 50–60 lines. These could be broken down into smaller private methods for improved readability and unit testing. process(), read\_data\_csv(), and filter() are all quite long too and could be broken down and refactored.

Security: Input values need sanitizing – eg comments.

**Code Readability**

Formatting Consistency: Good consistent formatting.

Local Commenting: Very good in Dart and Rust.

Contextual Commenting: Very good in the Dart code – explains the context for each class. Very good in the Rust for describing the function of each method.

Naming: Good naming of functions and variables that helps understand the codebase.

**Management**

Unit testing: Good unit testing at the Dart level. I couldn’t find any granular unit testing at the Rust level that doesn’t depend on the Flutter Rust bridge integrity.

Git use: Good evidence of branching but not pull requests.