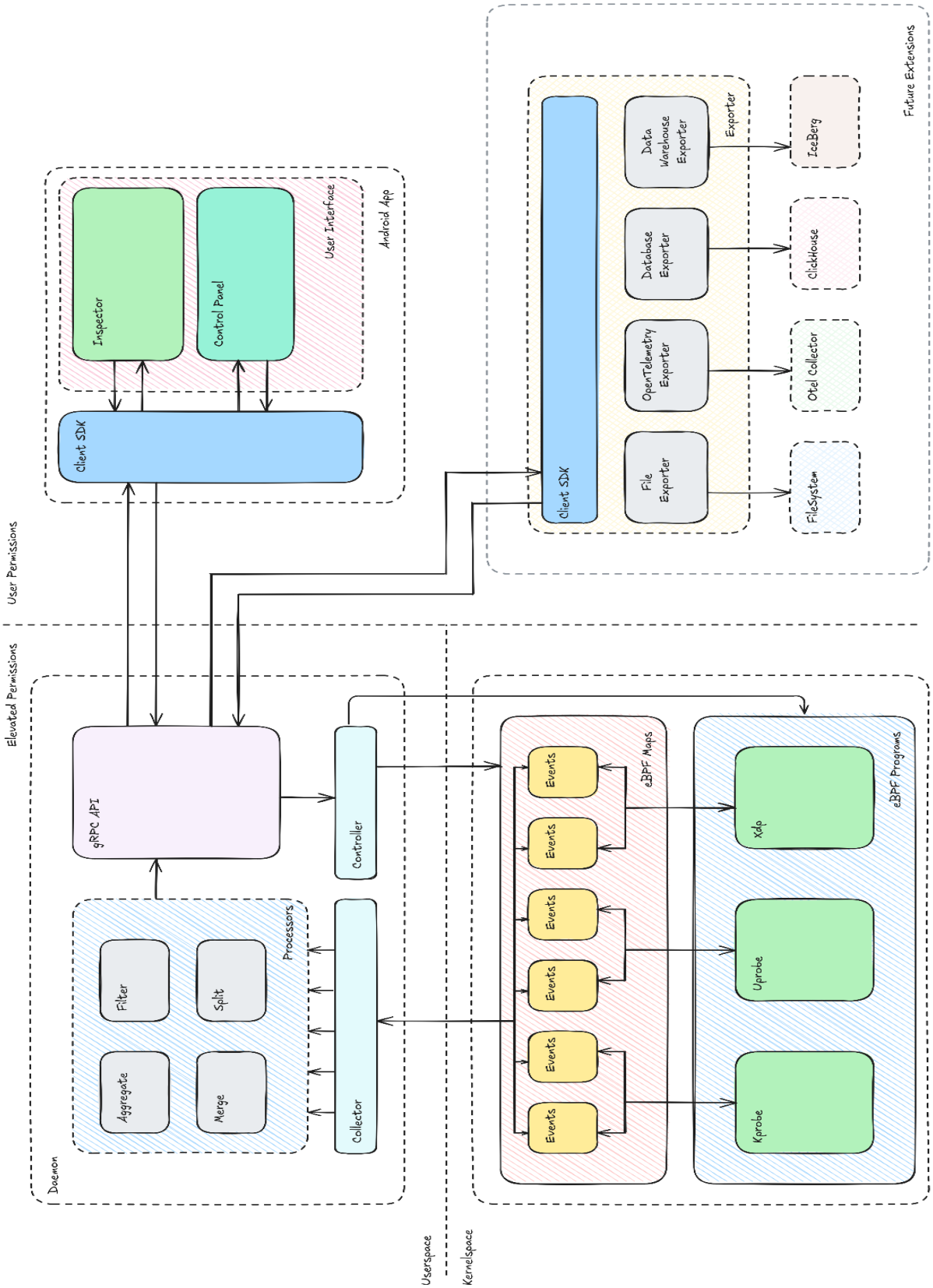
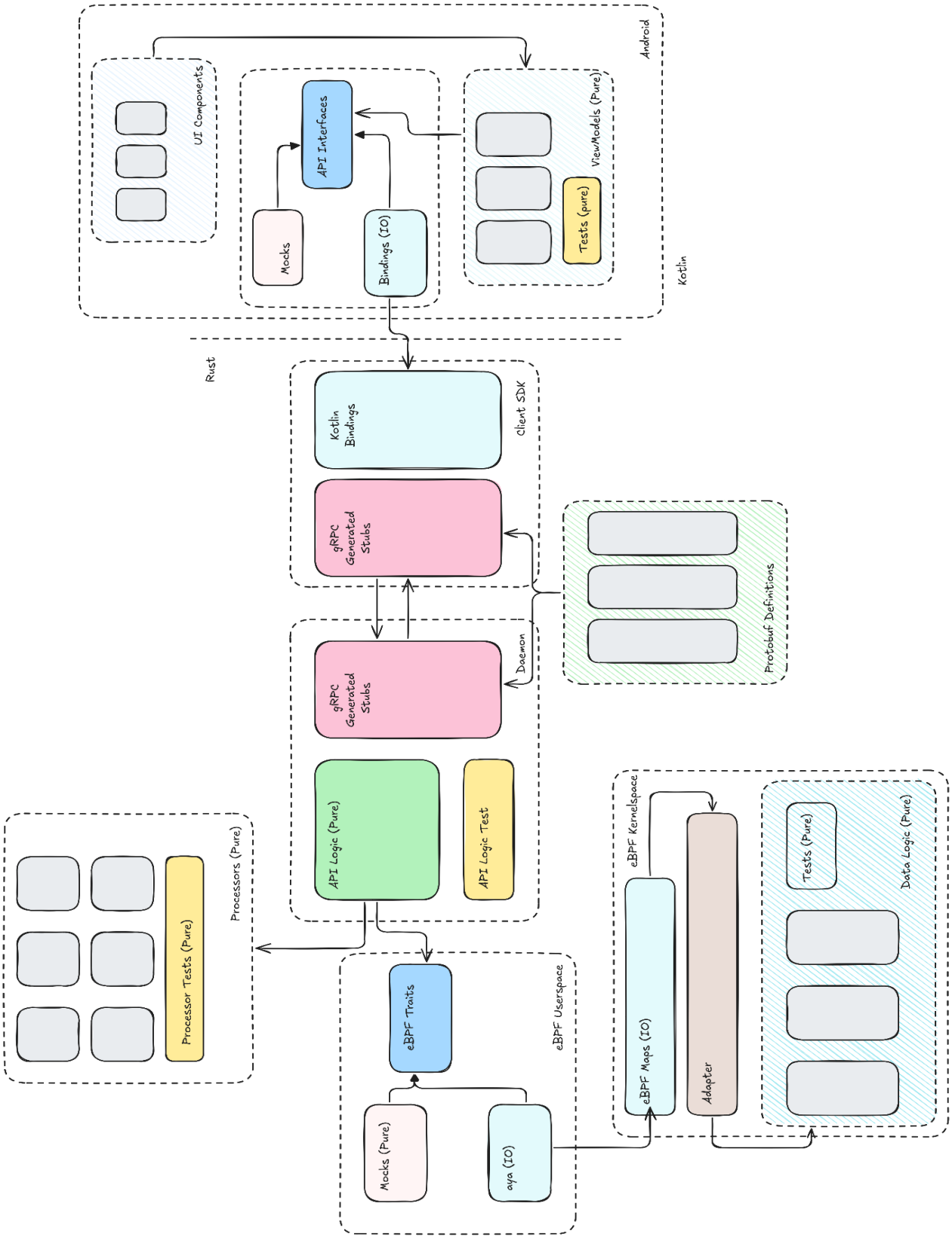
Runtime Components



Code Components



Technology Stack Summary

Our industry partner has two main requirements:

- Extract useful data from processes on the system via eBPF
- View the extracted data and control which data to extract via an Android app

These initial requirements require us to have at least four different components for our project:

- Several eBPF programs running in kernel space
- At least one eBPF loader program in userspace
- An Android application to display extracted data and control what to extract
- Some kind of communication between the two

1. eBPF Programs

For writing eBPF programs, there are two options suggested by our industry partner:

- C via libbpf
- Rust via aya

We ultimately decided to use aya because:

- eBPF programs and the loader can be written in the same language
- BTF support (compile once, run everywhere)
- Cross Compilation, Package Manager, Package Ecosystem

2. eBPF loader Program

Since we chose aya for our programs, we also chose the corresponding userspace library. For our async runtime we use tokio as it is already integrated into aya and works out of the box.

3. Android Application

We decided to use Jetpack Compose for our Android application because it is the way Google recommends for writing new Android apps. It has first-hand Material 3 support to match the look and feel of other Android apps. It was also familiar to several team members.

4. Communication

Our initial plan was to use Binder with AIDL for IPC, which is the typical way Android applications communicate. However, we realised that our application model required streaming event data from the daemon to the Android app. Also, testing on an emulator using AIDL proved cumbersome due to selinux permissions and visibility of methods in the Android SDK.

We finally decided to use gRPC instead, which allows for typesafe communication and has good support in Rust via the tonic library. Both ends of the communication will be written in Rust, as this increases portability and allows us to use the client not only in the Android app, but also in native apps. The bindings will be generated using the uniffi library created and used by Mozilla for Firefox.

The bindings allow file descriptors to be passed over the IPC, our plan is to use a socket pair between the client sdk and the eBPF loader that is passed over the bindings in our production setup to secure the connection. For testing, we simply bind the gRPC api to localhost instead of UDS, which makes testing a breeze. This gives us the best of both worlds.

Diagram Explanations

Runtime Components

The runtime component diagram shows our 4 component architecture, with possible extension points that may be of interest in the future.

The diagram is best explained from the bottom left. This part of the diagram shows our eBPF programs. We will have several, as an example there are shown "kprobe", "uprobe", "xdp" programs, which are the highest priority features requested by our industry partner.

These programs collect data that is put into what are called eBPF maps. These maps are data structures to share data from eBPF programs with userspace programs without unnecessary context switching between kernel and usermode. We are mainly interested in collecting data, so the maps will mostly contain events requested by the loader process.

The loader process, which sits on top of the maps, is divided into four smaller components. One component collects data from the eBPF maps and passes it to the next component, which is there to process the data. A third component controls the eBPF programs (e.g. loads and configures them). The last component provides the loader functionality via the gRPC API. The corresponding client for this API is called "Client SDK" and is located on the right. This library is part of our Android application and is used to display events and control the loader program. Finally, below the Android app are the future extension points. As we have decoupled the Client SDK from the Android application, it would be possible in the future to create various exporters to send the events to remote destinations such as databases, OpenTelemetry collectors, etc.

Code Components

The general structure of the code components is that we have tried to decouple as much as possible from the start, so that it is both easier to test and easier to collaborate and not step on each other's toes.

The bottom left shows the code structure of our eBPF programs. The code is divided into three parts. The first part contains all the logic for processing and converting the data to make it usable by our loader process. This part should be kept pure (no IO) to make testing easier and less time consuming. These components are then adapted and put into real eBPF programs that use IO and exchange data with the eBPF maps.

Above this section is the interaction of our loader process with the eBPF programs. We try to abstract the API provided by aya into our own interface, which can then be mocked to allow unit testing.

The next part is the processing logic shown above. This logic is pure and should be IO free to allow better testing. As data processing is a bug-prone task, testing is especially important here.

Using these parts we implement our API, this API is then used to implement the grpc stubs. These stubs are generated from .proto files for both client and server. The client uses these stubs and exports the API to Kotlin by generating bindings.

On the Kotlin side, we follow the same principle as for our eBPF interaction. We create an interface for our client SDK so that we can mock it in our viewmodels. These viewmodels can then be IO free and run even without an Android emulator, further reducing the need for testing.

The last missing piece is our UI components. These make use of our viewmodels and can therefore be properly tested.