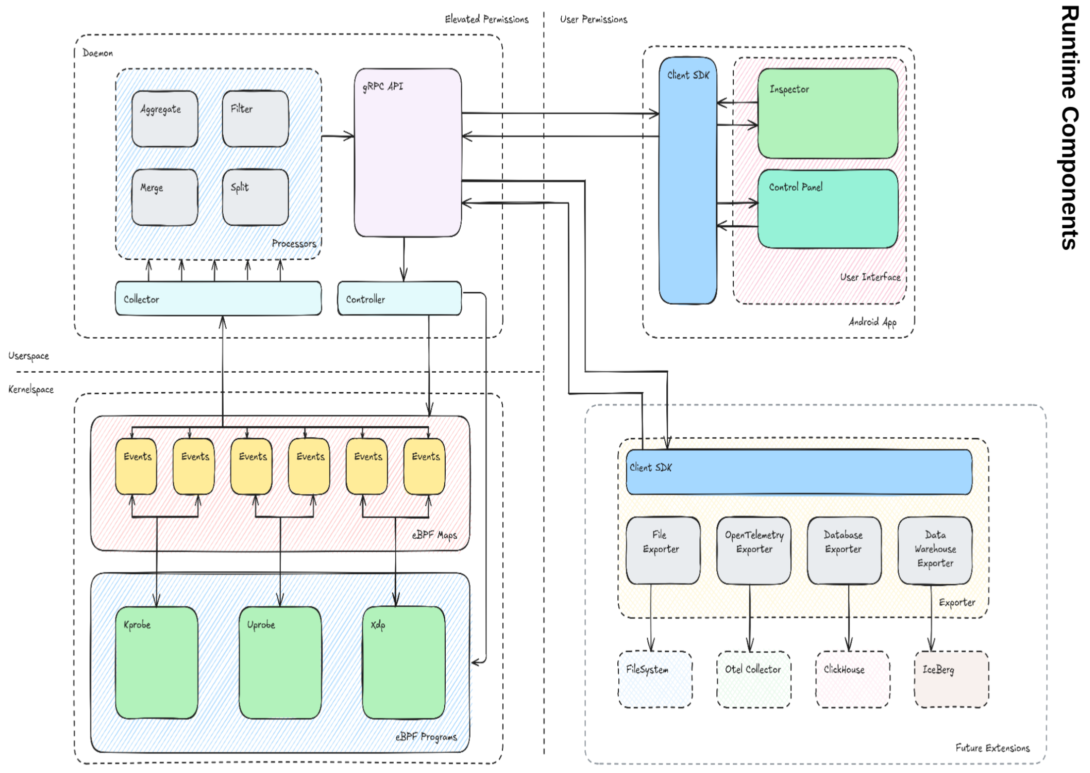
Runtime Components



Components

Our Architecture is made up of four smaller components to allow for possible extension in the future.

eBPF Programs

A single rust crate contains all the eBPF functionality, this is currently called `ebpf-refactored`. It defines multiple programs for collecting different kinds of events:

- [x] any form of write to the disk
- [x] measuring the duration of syscalls
- [x] capturing events which open or close file descriptors
- [x] sending and receiving signals
- [x] monitoring adding and deleting jni references

- [x] monitoring art garbage collection

For details have a look at goals document

These eBPF programs are configurable at runtime, allowing dynamic attaching and detaching of programs. This makes it possible to adjust what events and from what apps to collect.

For communication between Userspace and eBPF programs, maps are used. The main variants are the `BPF_MAP_TYPE_HASH` and `BPF_MAP_TYPE_RINGBUF` .

The first one is mainly used for event filtering. Different maps store different kind of filters (pid, comm, exe_path, ...). The value inside these maps is a bitmap, signifying to which event kind this filter applies and how to apply the filter.

The latter is used for sending events from eBPF to the daemon. Strong ordering and `epoll` makes it the ideal map type for collecting event data.

Userspace Daemon

The userspace daemon has four jobs.

Configuration of eBPF programs

Through aforementioned maps, the userspace daemon can configure the eBPF programs at runtime. This capability is exposed through a `grpc` API, to make configuration possible from other applications.

Collecting events

Also through maps, the userspace daemon collects events and sends them to client over a `grpc` API.

Processing events

Between collecting events from eBPF and sending them to clients, events can be further processed:

- [x] Aggregate
- [ ] Filter (moved to ebpf)
- [x] Combine (we merge the event stream)
- [ ] For more see [goals document](https://github.com/amosproj/amos2024ws03-android-zero-instrumentation/blob/main/Documentation/

Client SDK

The client SDK offers convenient APIs targeted at developers over the `grpc` infrastructure. It

The client SDK offers convenient API targeted at developers over the gRPC infrastructure. It is used by our Android application but can also be used in the future to write other clients.

Possible usecases:

- [x] Visualizing events in an Android App
- [ ] Exporting events in Open Telemetry format
- [ ] Sending events to a data warehouse
- [ ] Sending events to databases like Clickhouse
- [ ] Persisting events to disk for later analysis

Frontend Android App

Motivation

The purpose of the frontend is to provide a **on-device** configuration and visualization interface for the ZIOFA backend.
In situations where there is no external device available for configuring, receiving and visualizing the data, the user should still be able to analyze the behavior of components on the device in real-time.

Requirements

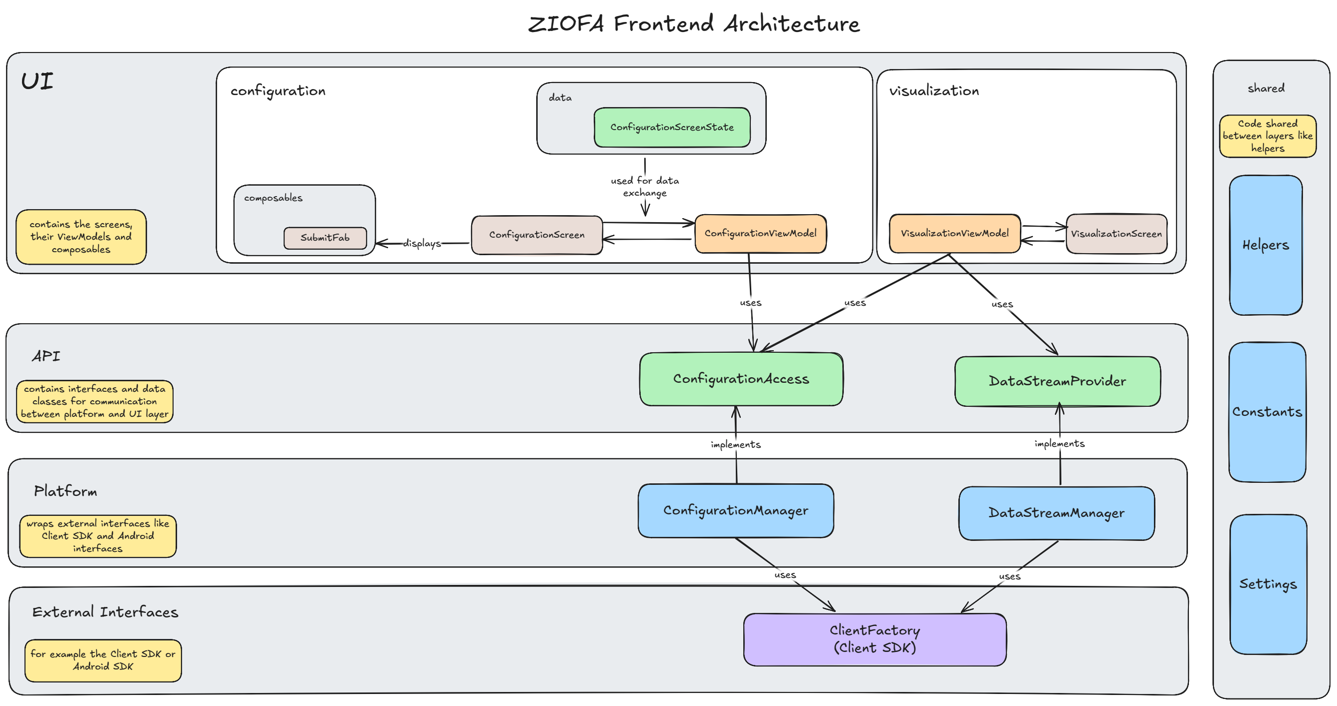
The requirements for the frontend can be found in the requirements doc under / Documentation.

Constraints

- Due to limited hardware resources and the need for non-interference with the behavior of other applications, the frontend should be as resource-efficient as possible. The current implementation is not resource-efficient enough to be used in production environments, but there are ideas for improving the resource usage, which will be discussed later.
- Additionally, while the frontend itself can be run as a non-system app, the backend must always be running in the background as root, meaning the application cannot be used without root access to the device. (for frontend developers only: the mocked Client SDK can be used without root access)
- For some features, like the overlay mode, the frontend requires runtime permissions that are non-requestable on some devices due to missing settings menus. This means that the permissions must either be granted via ADB or the integrator must ensure that the app will be granted the permissions by pre-granting them. For information on how to grant permissions, see the README.

High level app architecture

An overview of the high level architecture of the app can be found in the following diagram.



Runtime view

The following scenario describes the standard use case:

- The user starts the app
- Upon startup, the app performs a sanity check of the backend.
  - If an error occurs the error is shown to the user.
  - While the sanity check is running, the app shows a loading screen.
  - Upon completion, the app transitions to the main menu.
- The user navigates to the configuration menu
  - The app starts polling the process list from the backend.
  - The found processes are combined with data from PackageManager to display icons, app names and process names.
- The user can select a process to configure the backend for.
- The user selects activates or deactivates backend features for the selected process.
  - Upon completing the selection, the user submits the configuration and it is sent to the backend
  - The backend sets and persists the configuration.
  - If an error occurs, the error is shown to the user.
  - Otherwise, the backend starts collecting data for the set configuration.
- The user navigates back to the home screen and then to the visualization screen.
  - App polls the backend for info on running processes and combines this information with the backend configuration applied by the user to display the configured options.
  - The user selects a process, metric and timeframe to visualize.
  - The app subscribes to the events from the backend starts aggregating the data

    - The app subscribes to the events from the backend starts aggregating the data.
    - The aggregation is defined per metric and timeframe
    - The app display the visualization to the user, either as a chart or as an event list.
- The user can select the overlay mode on the visualization screen to launch the overlay.
    - Upon enabling the overlay a seperate background service is started that starts the overlay window and manages its data. The service is stopped again via the app.

Future work

Resource usage

- Aggregation of events should optimally happen in the backend. This would reduce the amount of communication between frontend and backend significantly and improve CPU usage drastically.
- As a first step, the Client SDK directly implement the gRPC communication instead of using the Rust Client SDK, which currently uses a lot of resources for JNI calls.
- Both of the used visualization libraries are not explicitly dedicated to live data visualization. Future work could evaluate more visualization libraries regarding their CPU and memory usage and switch to a more resource efficient solution.

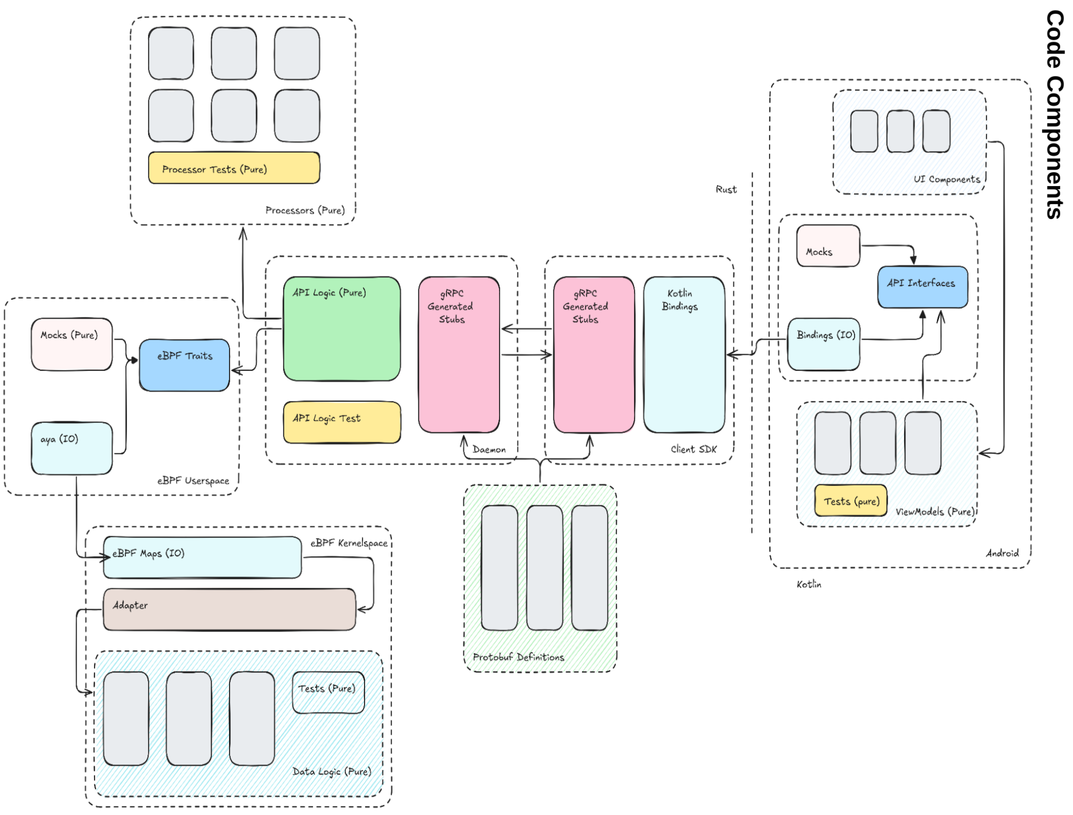
Refactoring

- The current codebase certainly has room for improvement in terms of maintainability. Adding new backend features is fairly easy once the developer is familiar with the classes to change, but it is not trivial and many files need to be changed. Ideally, the mapping between backend features, charts, visualizations and configuration should be consolidated to reduce the required amount of changes.
- The Client SDK is too tightly coupled to the frontend. While only the platform-Layer calls methods from the Client SDK, the upper layers still use some data classes from the Client SDK.
  Changes in the Client SDK often require too many changes in the frontend as well. While this makes sense in terms of reducing the amount of wrappers, it makes it hard for developers not familiar with the frontend to independently work on the Client SDK.

Usability

- Often, there are many clicks required for certain tasks. An example would be configuring and then visualizing a process. The user would have to select the process, select their desired features, click submit, then navigate back to the home screen, go to visualization, select everything again and only then see the visualization. For issues like this, shortcuts could be integrated to f.e. directly navigate to the visualization with the selection applied.

Code Components



Isolation

Isolation has high priority:

- Assigning individual components to individual team members
- Faster build and test cycle
- Easier test design

Abstracting IO

For creating useful and efficient tests, we need to abstract IO operations as much as possible.

Using Rust's traits and Kotlin's interface we can decouple our implementation as much as possible from the real system. This makes deterministic and fast testing possible.

Type safety

Both Rust and Kotlin are type safe languages. For our API we use `grpc` to generate type safe code definitions for both client and server.

safe code definitions for both client and server

gRPC-API service endpoints

InitStream(google.protobuf.Empty) returns (stream events.Event)
Initialize the stream over which all following collected events will be sent. If this gets called multiple times or by different clients, all streams will receive the same data.

ListProcesses(google.protobuf.Empty) returns (processes.ProcessList)
List all processes currently running.

GetConfiguration(google.protobuf.Empty) returns (config.Configuration)
Get the currently set configuration.

SetConfiguration(config.Configuration) returns (google.protobuf.Empty)
Set a new configuration. The old one will be replaced without merging.

IndexSymbols(google.protobuf.Empty) returns (google.protobuf.Empty)
Index all symbols on the whole device into the database. This has to be called once upon startup and every time a package of interest is (re-)installed/updated.

SearchSymbols(symbols.SearchSymbolsRequest) returns (symbols.SearchSymbolsResponse)
Search all symbols for the specified query. Requires an indexing run beforehand.

GetSymbolOffset(symbols.GetSymbolOffsetRequest) returns (symbols.GetSymbolOffsetResponse)
Get the offset of the specified symbol.