VitalVision

Real-Time Validation of Wearable Photoplethysmogram and Electrocardiogram Sensor Data in Constrained Mobile Environments

ETH zürich

Emanuel J. Mairoll

siplab

REGISTER NOW After registration you will be able to apply for this opportunity online.

Android/iOS live interface for visualizing streaming wearable sensor data

Wearables are all the rage. They collect a lot of data, but it's not clear what that data looks like. What do wearables 'see' when we move, when we walk, talk, or jump? What is the depth of our vital parameters that they can observe?

Keywords: App Development, iOS, Android

Description

In this project, we will develop an Android and/or iOS Application that connects to multiple wearable devices, streams the sensor data and live plots it upon the users request. We will interface with wearable devices that are much like the Apple Watch or Fitbit, with sensors for motion tracking, physiological sensing, and activity recognition. The devices have a Bluetooth Low Energy (BLE) interface that we can obtain the data through.

Moreover, for the standard signals such as the Photoplethysmogram (PPG) and Electrocardiogram (ECG), the app should automatically detect signal disruption (e.g., when an electrode falls off) and display a warning message.

Note: Candidates should have experience in app development.

Goal

Develop an App to monitor the measurements of multiple sensing devices.

Contact Details

Manuel Meier, manuel.meier@inf.ethz.ch

Calendar

Earliest start	2023-09-18
Latest end	2024-06-30

Location

Sensing, Interaction & Perception Lab (ETHZ)

Labels

Semester Project

Bachelor Thesis

Master Thesis

ETH Zürich

- Eidgenössische Technische Hochschule Zürich
- Ranked top 10 globally
- Top 1 University in Europe
- ~24 000 students, 500+ labs
- Notable Alumni:
  - Albert Einstein
  - John von Neumann
  - Niklaus Wirth

SIPLAB @ ETHZ

- Sensing, Interaction & Perception Lab
- Research in computational interaction, embodied perception, and mobile health
- Focus areas: mixed reality, machine learning, wearable technology
- Lead: Christian Holz, 20+ researchers
- Collaborations: ETH AI Center, Max Planck ETH Center, Zurich Information Security & Privacy Center

team research publications teaching & theses join us contact

Ultra Inertial Poser
Scalable Motion Capture and Tracking from Sparse Inertial Sensors and Ultra-Wideband Ranging. ACM SIGGRAPH 2024.
details · project page

MARLUI
Multi-Agent Reinforcement Learning for Adaptive Point-and-Click UIs. ACM EICS 2024.
details · project page

Detecting Users' Emotional States during Passive Social Media Use
ACM IMWUT 2024.
details · project page

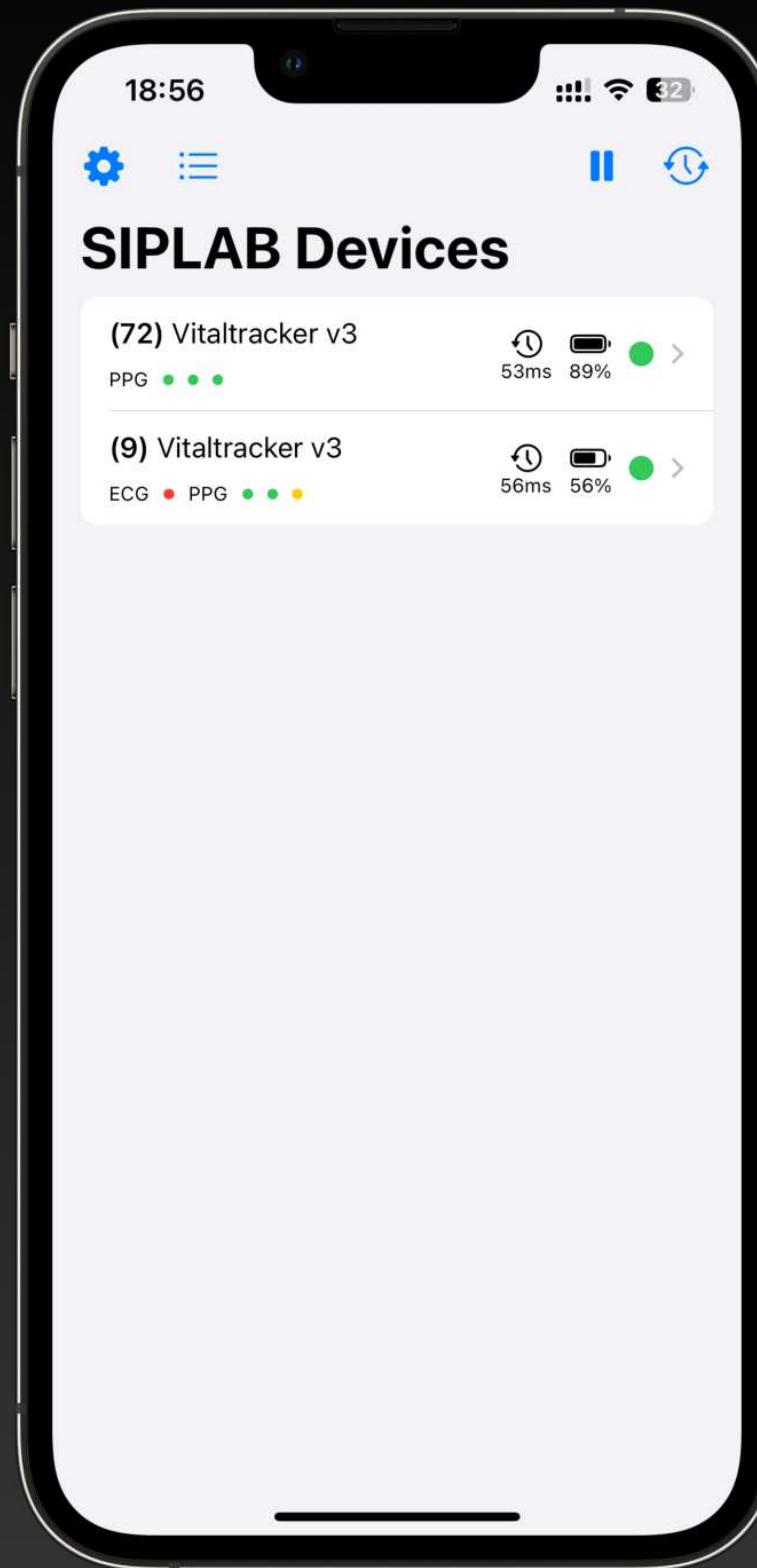
Biomarkers for daily fatigue
Continuous wearable monitoring for fatigue modeling. Cell Press iScience.
details · project page

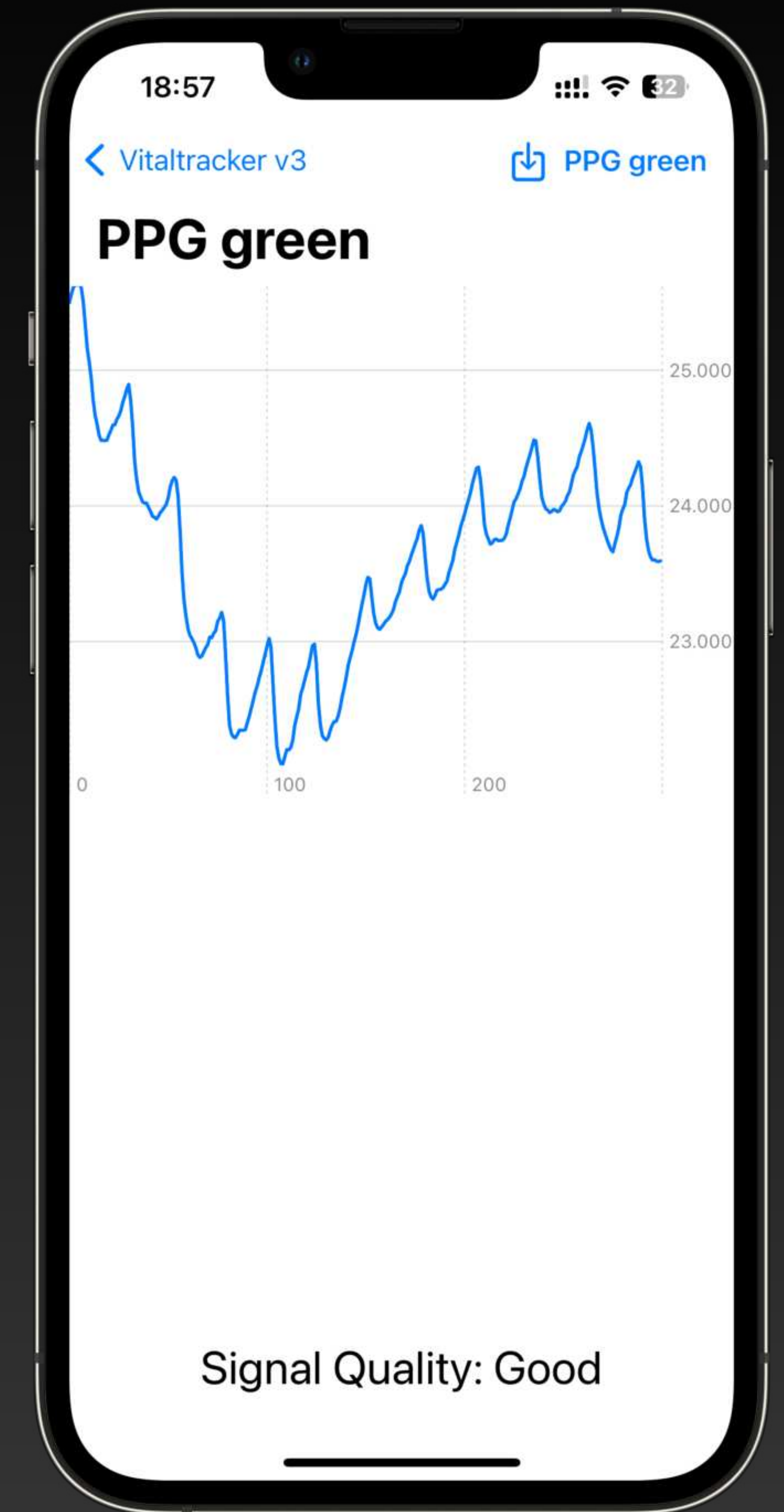
The Problem

- SIPLAB builds wearable sensors similar to Apple Watch and FitBit
- Used to conducts studies such as testing altitude sickness
- Data collection is intrinsically problematic:
  - ECG and PPG Sensors are highly susceptible to disturbances
  - Problems include sensor displacement, motion artifacts etc.
- Data recorded so far is "blind," with no real-time feedback

VitalVision

- Develop an Android/iOS app to visualize and validate live wearable sensor data
- Connects to multiple devices via Bluetooth Low Energy
- Streams and live-plots data
- Handles ECG and PPG signals
- Detects disruptions and shows warnings



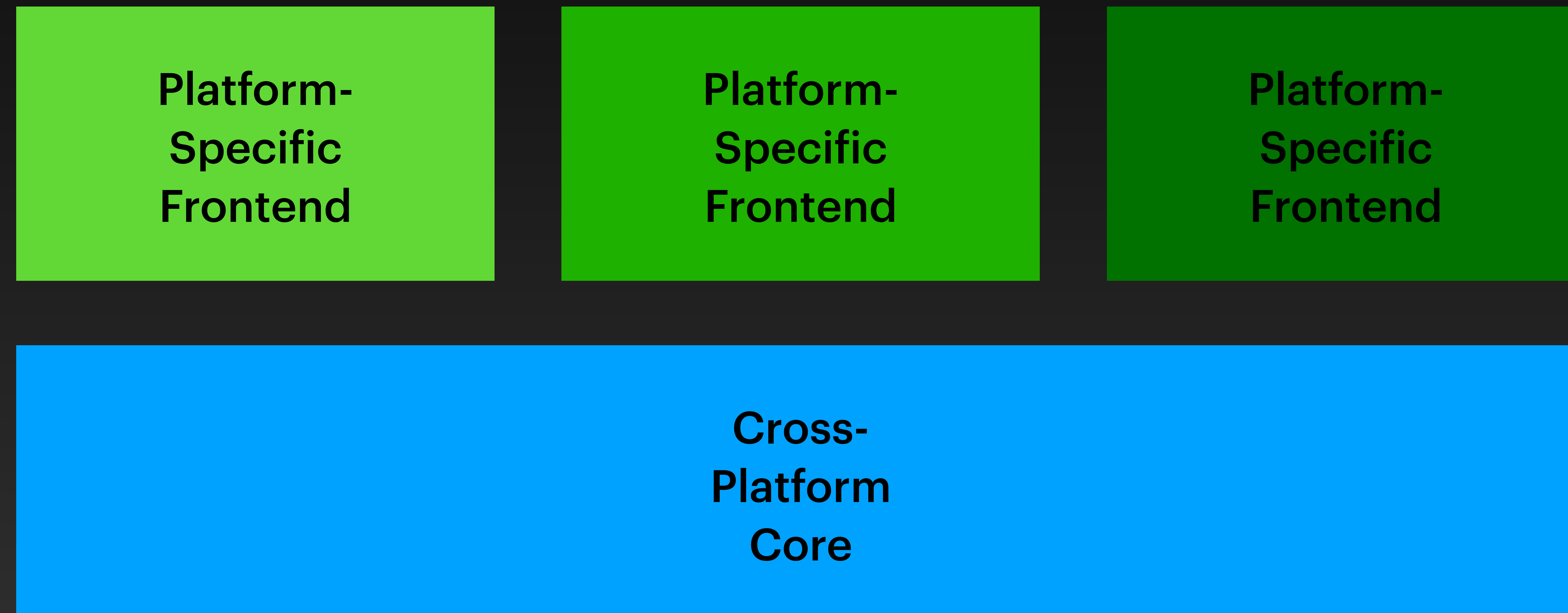


Architecture Time!

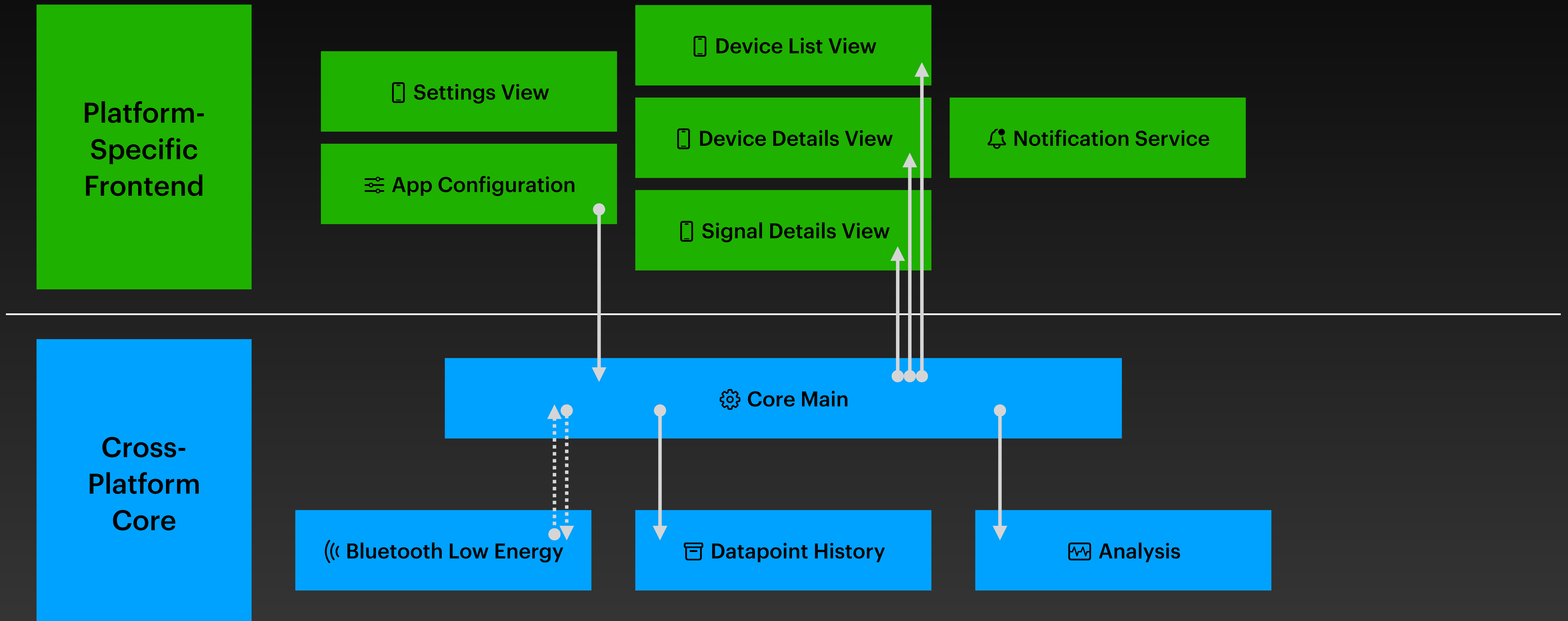
Requirements

- Connect via Bluetooth Low Energy (BLE)
  - Perform data recording setup
  - Stream data in real-time
- Report status of device (e.g., battery level)
  - Visualize sensor data
  - Validate signals efficiently
  - Warn user if problems detected
- Cross-platform compatible (iOS and Android)

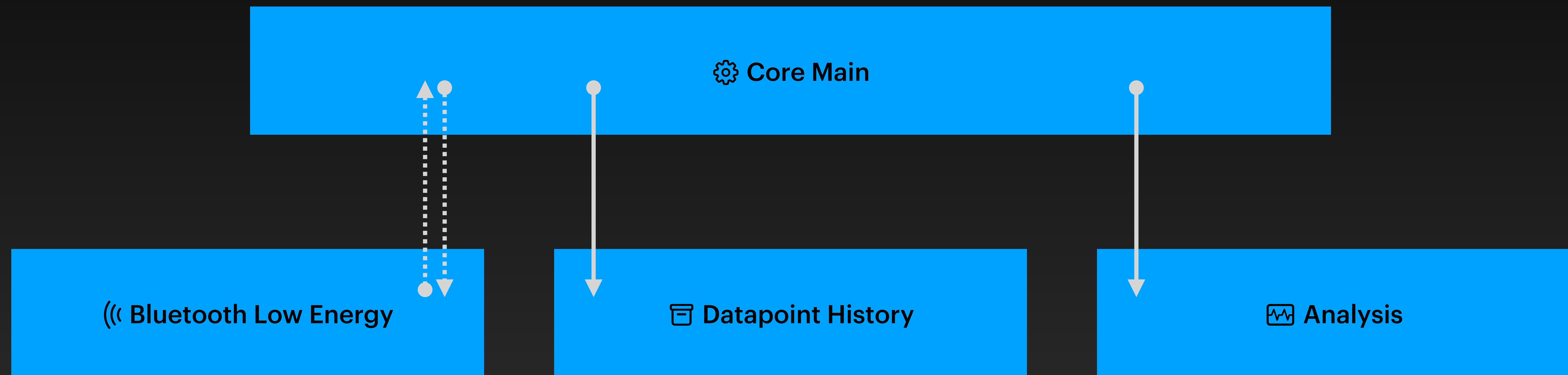
Architecture



Architecture



VitalVision Core



Rust

- modern programming language designed for safety and performance.
- compiles to single binary, cross-platform compatible.
- prevents memory errors without needing a garbage collector.
- has zero-cost abstractions, memory safety, and concurrency support.

- But: Steep learning curve due to ownership and borrowing.

```
fn main() {
    let hello: String = String::from("Hello");
    some_function(hello);
    println!("{}", hello);
}

fn some_function(input: String) {
    println!("{}", input);
}
```

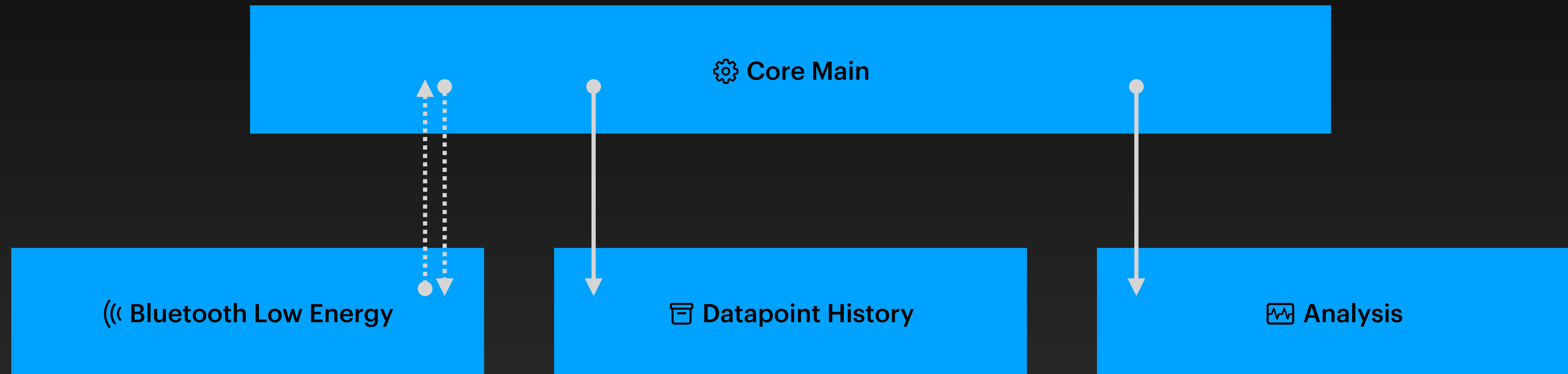
⚠TRADE OFFER⚠

i receive:
Your
mental health

you receive:
Memory safety

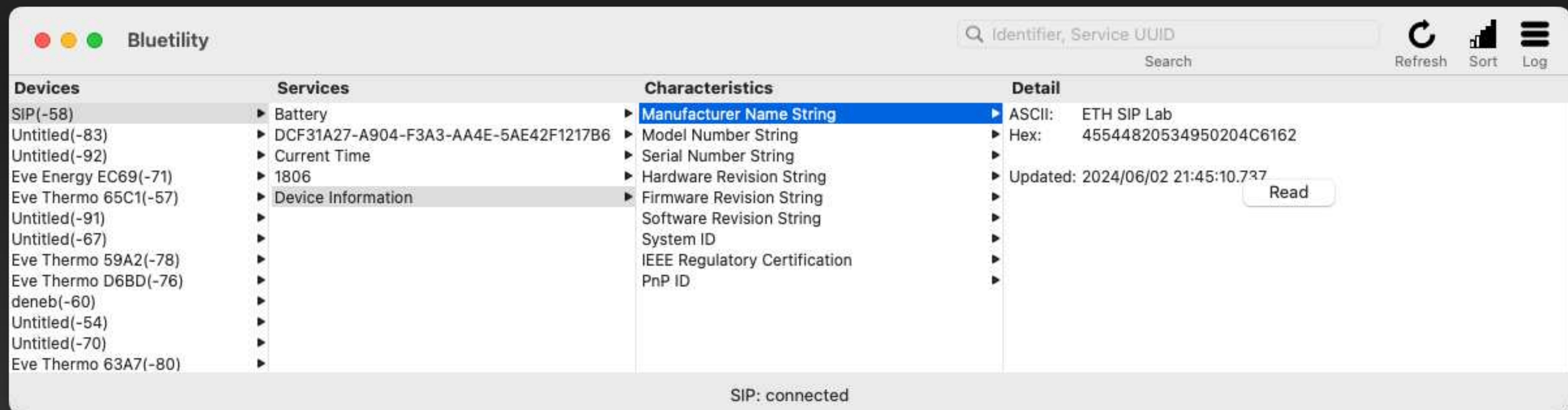
Rust's borrow checker

VitalVision Core



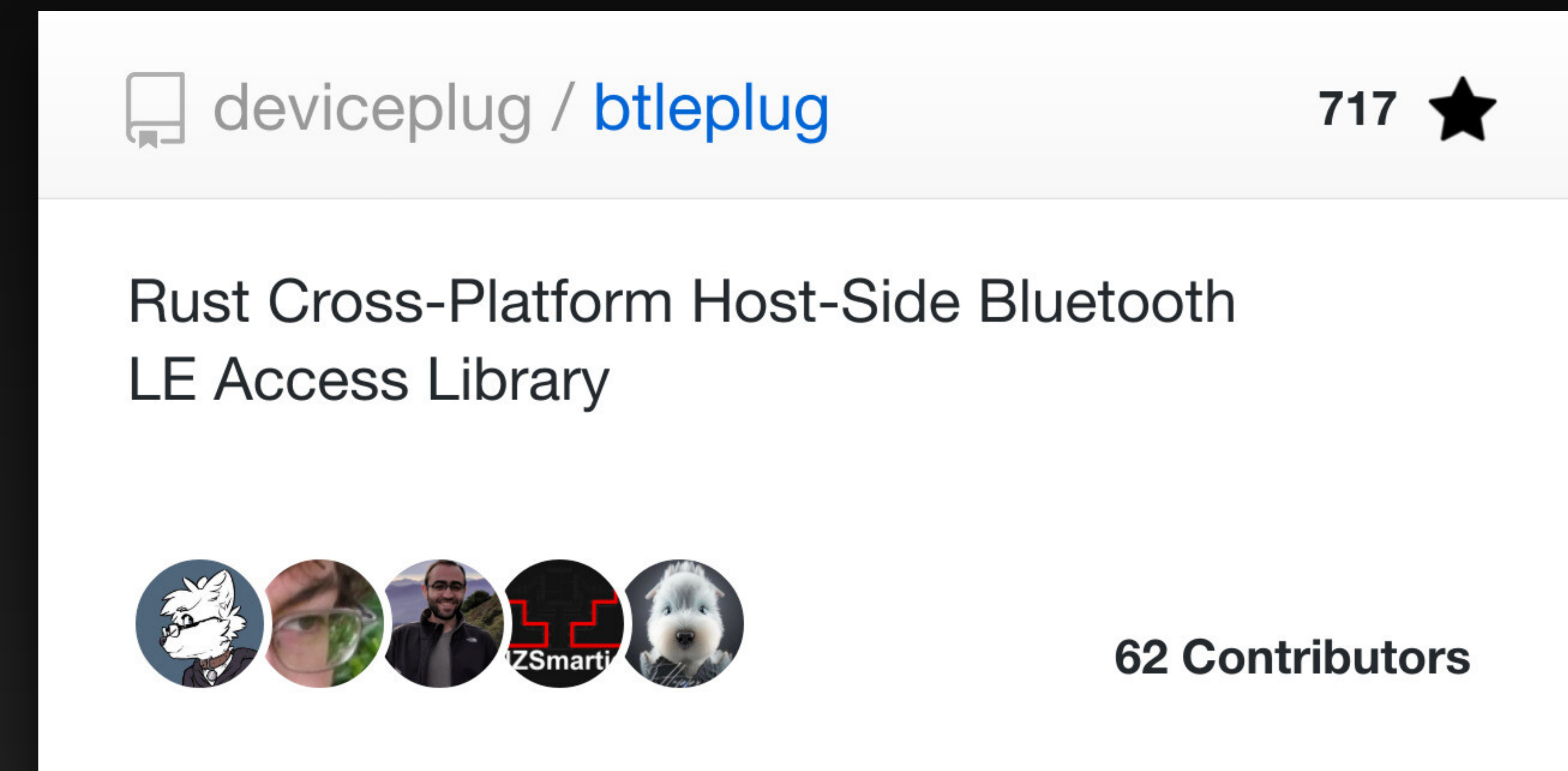
Bluetooth Low Energy

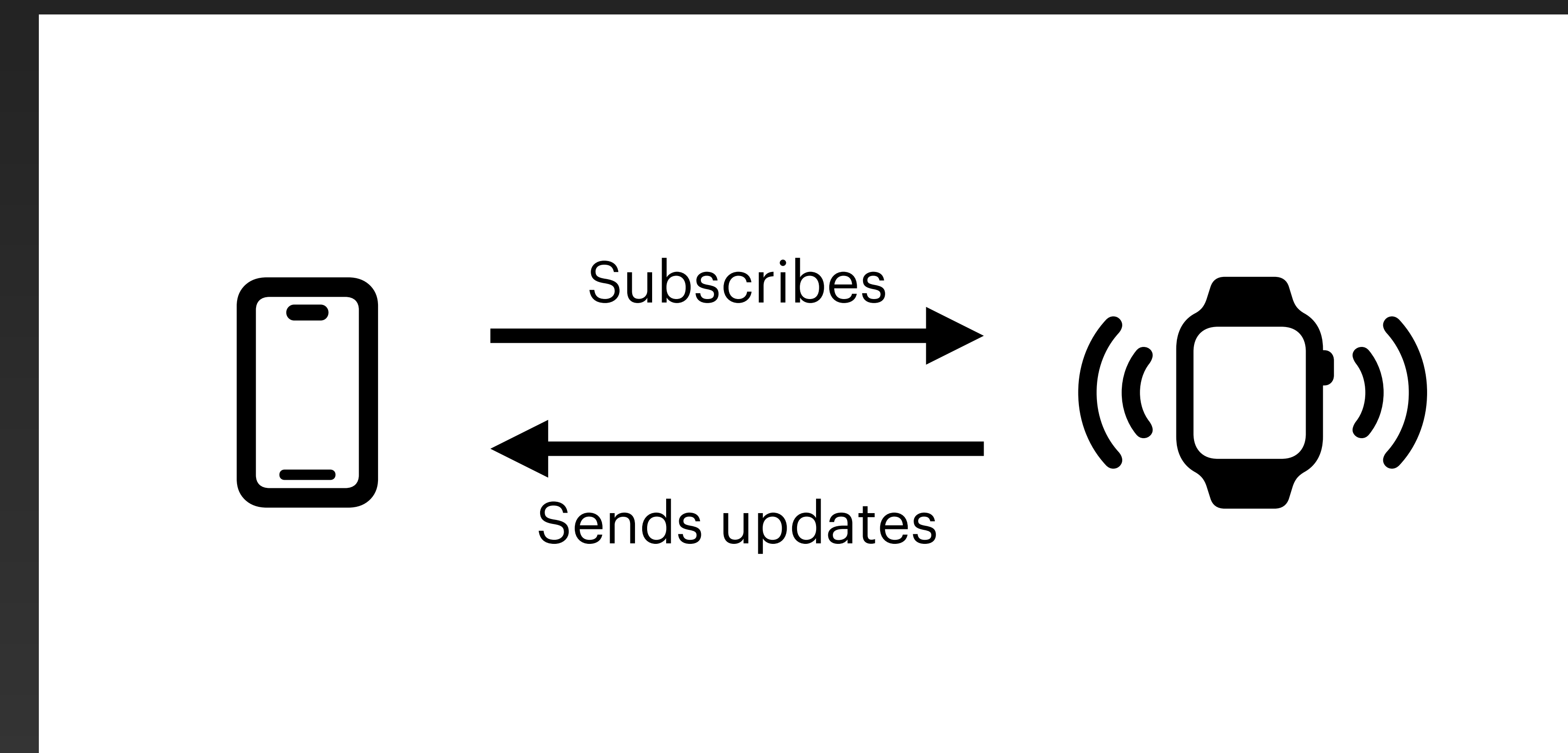
- **Misnomer**: Mainly for structuring communication, not just "low energy"
- **Roles**: Central (controller) and Peripheral (device)
- **Hierarchy**: Device -> Service -> Characteristic
- Characteristic can be read, written, or subscribed to



BLE Component

- Cross-Platform Library: btleplug
- App searches for available SIPLAB devices
- On Connect:
  - Read device information and configuration
  - Set time on device
  - Subscribe to data characteristic
  - Register update handler for incoming data

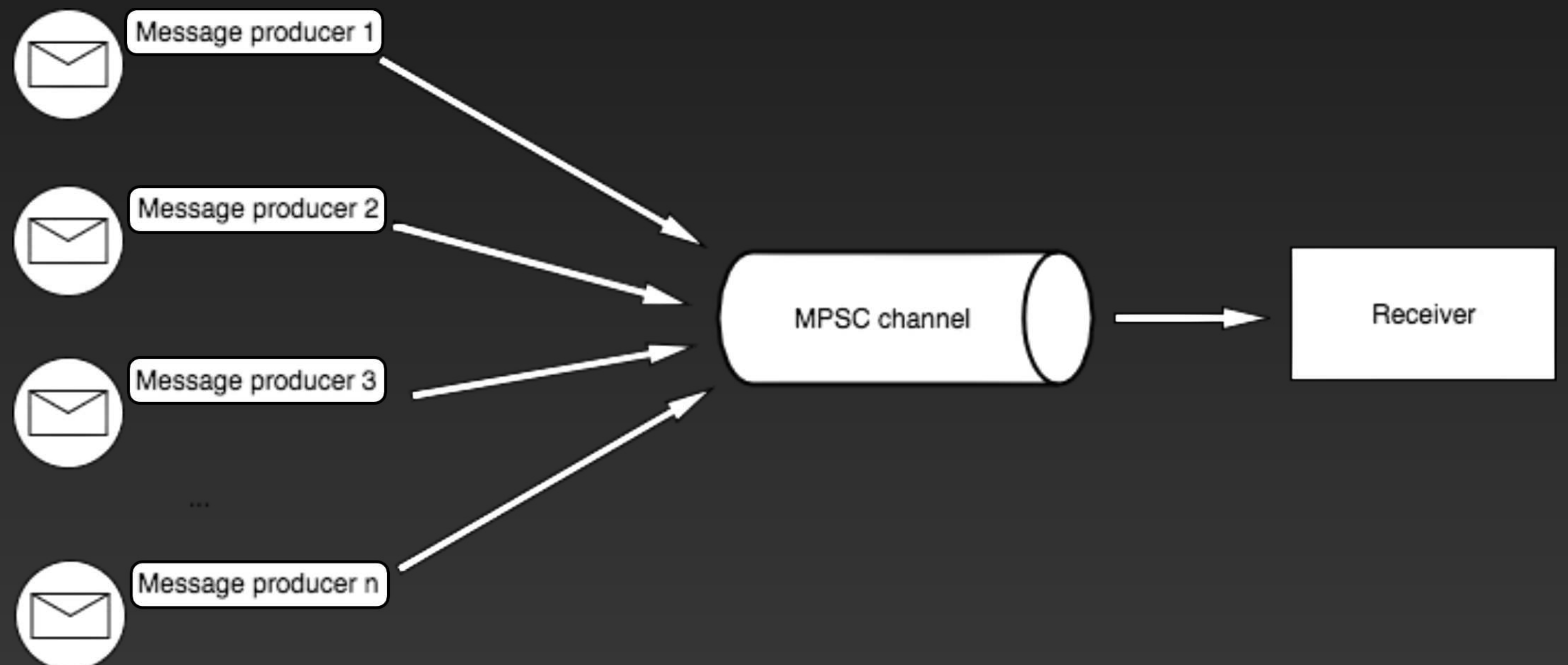




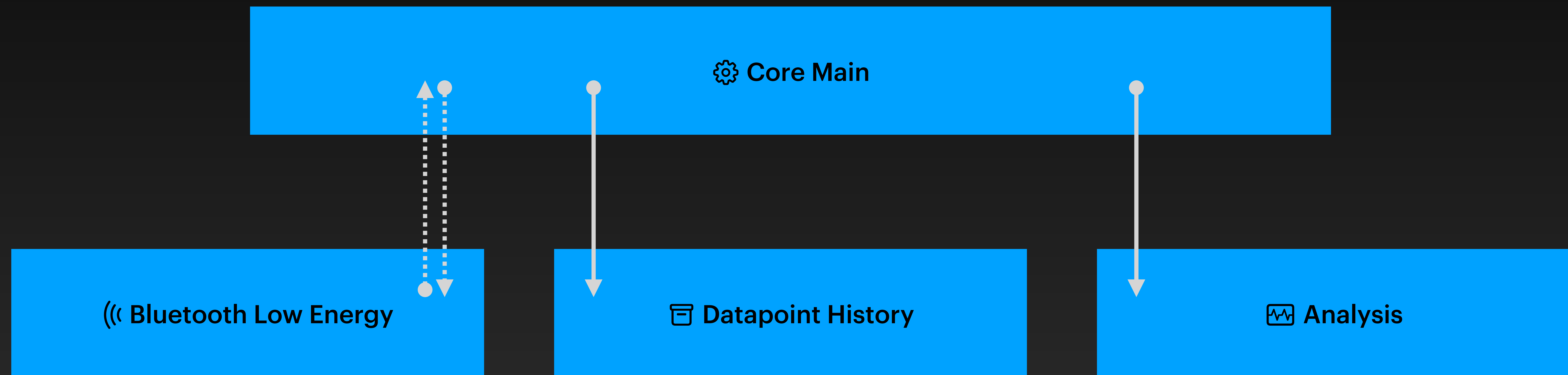
Event-based Architecture

- BLE Component has many different threads
- Event-driven approach for communication between threads of BLE components and the Core Main.
- Events like "DeviceConnected," "BatteryLevelChanged," and "NewDataPoint"
- Tokio and Channels:
  - Tokio: Rust runtime for asynchronous programming
  - Channel: Allows threads to send messages between each other.
  - MPSC (Multi-Producer, Single-Consumer): Type of channel where multiple threads can send messages to a single receiver.

```
use tokio::sync::mpsc;

async fn main() {
    let (sender, mut receiver) = mpsc::channel(100);

    sender.send(42).await.unwrap();

    println!("got message: {}", receiver.recv().await.unwrap());
}
```

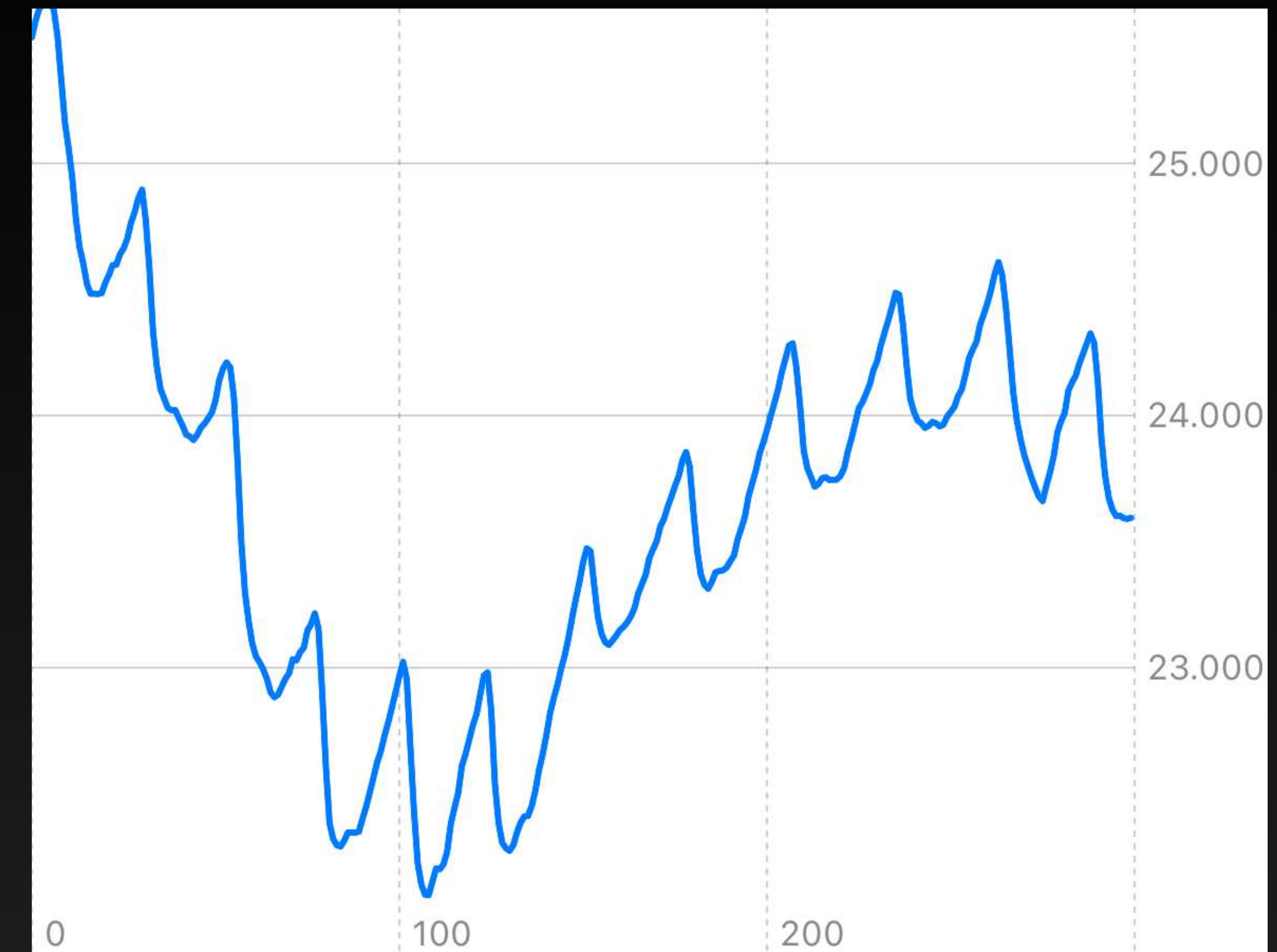


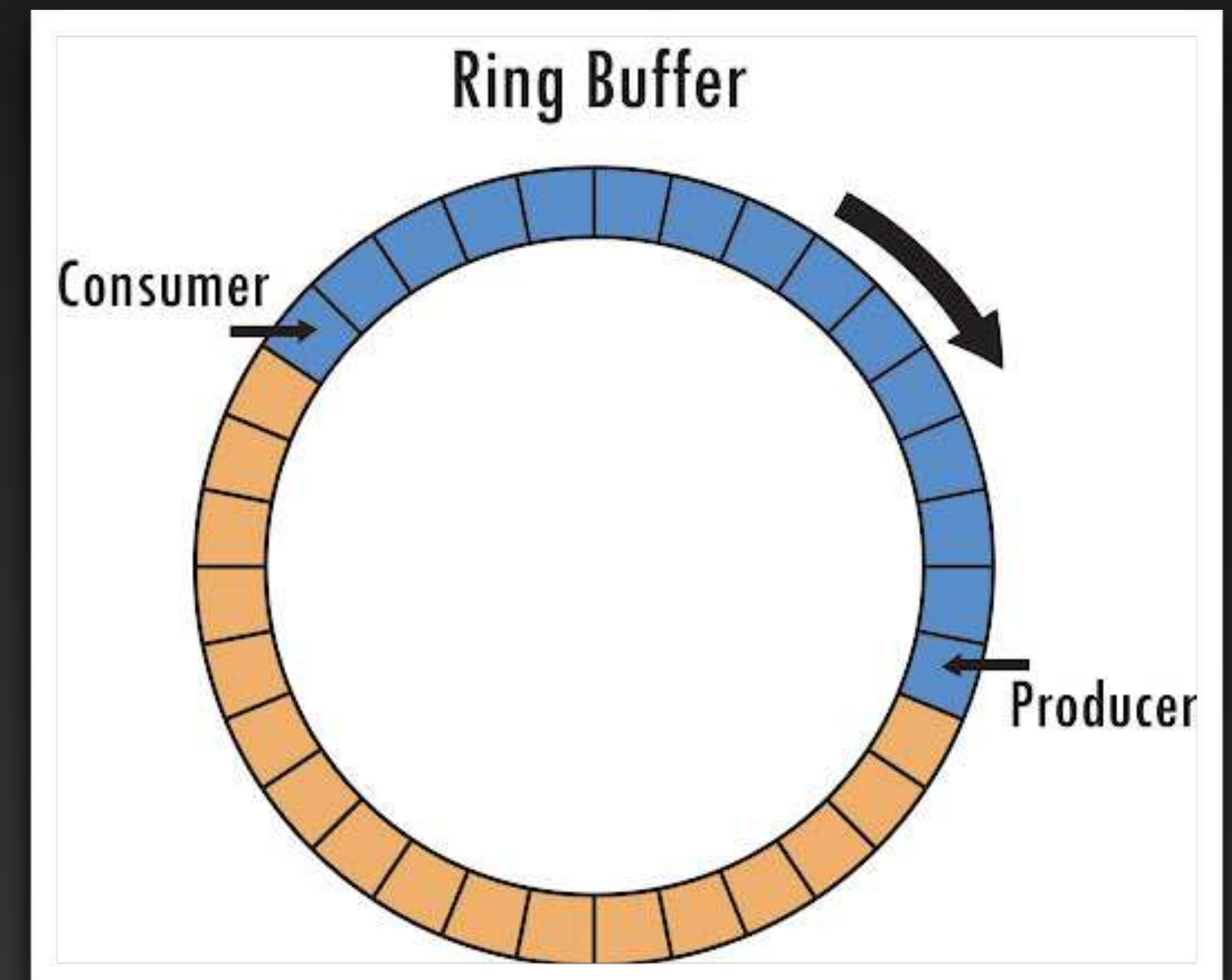
VitalVision Core



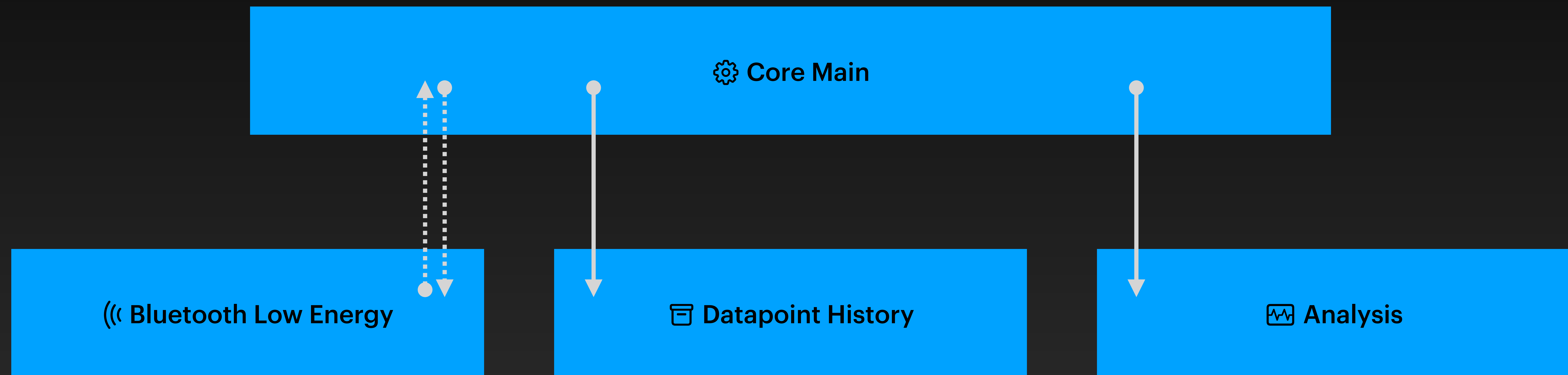
Datapoint History

- New data points received continuously
- Interested in history, eg. last 100 points simultaneously
- Methods:
  - add_datapoint: Adds a new data point
  - get_slice_with_len: Retrieves the last N data points
- Uses a ring buffer to avoid unnecessary copying





VitalVision Core



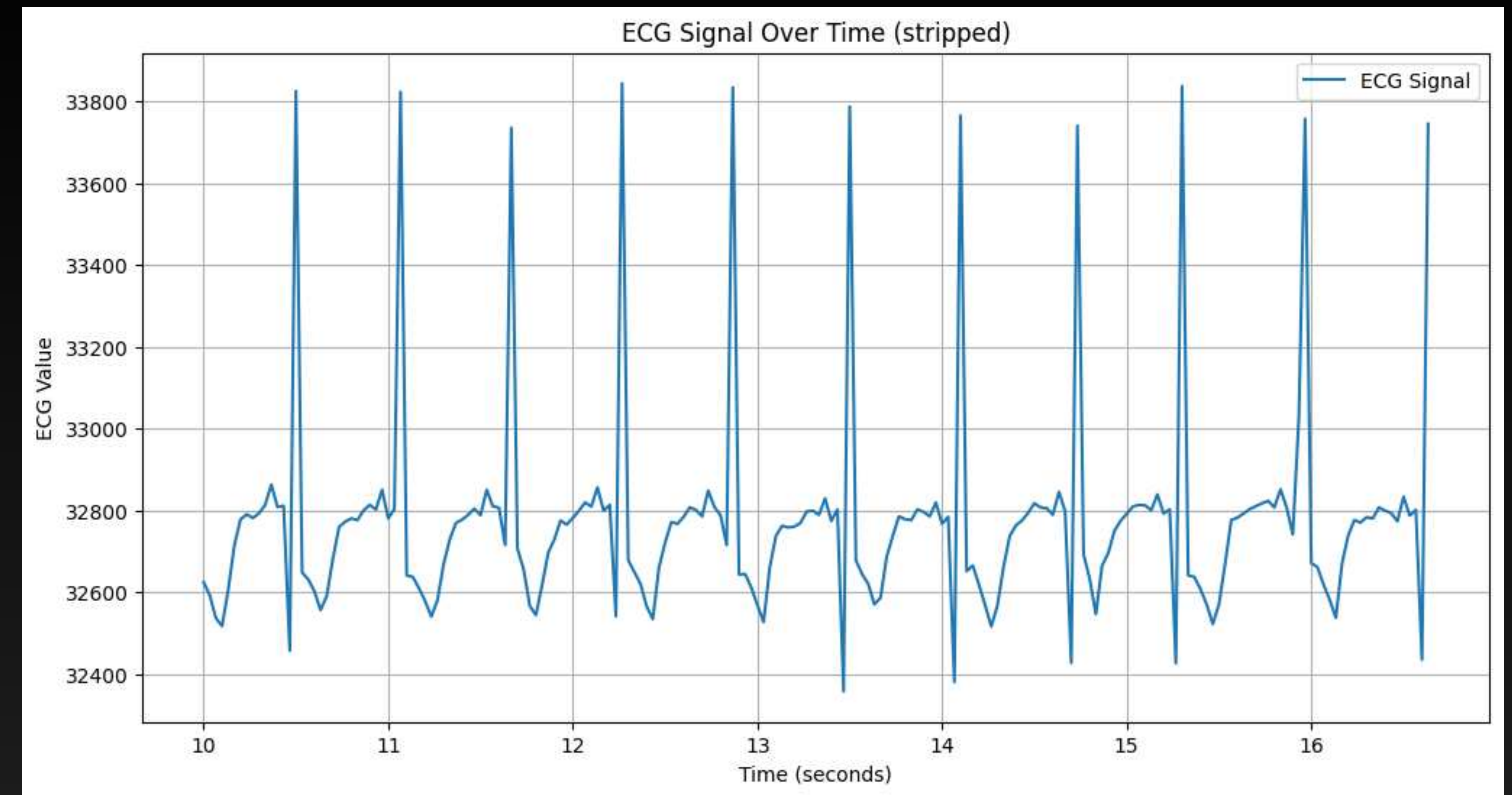
Analysis

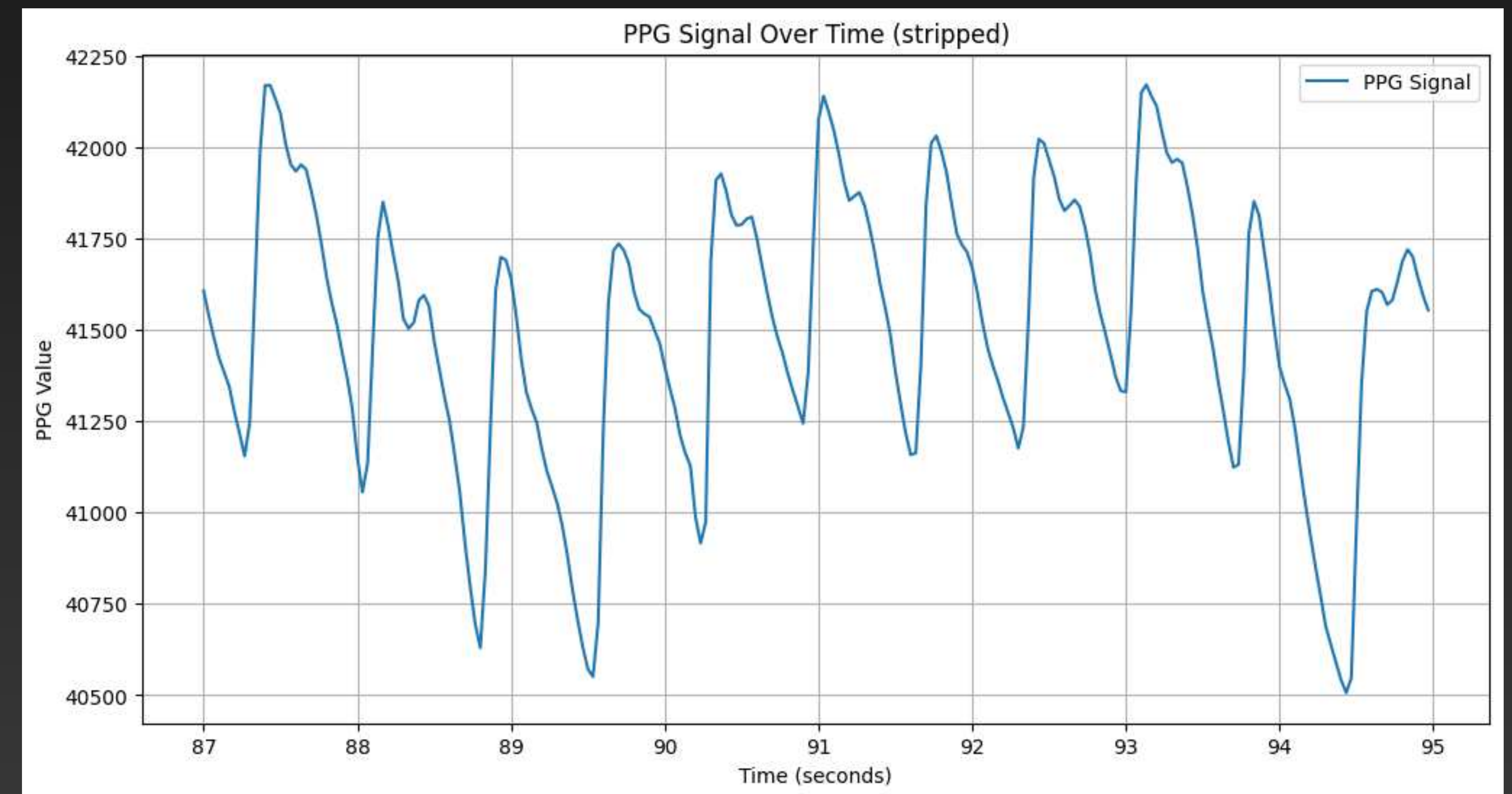
ECG (Electrocardiogram):

- Measures electrical activity of the heart
- Electrodes measure voltage difference for example on the chest
- Characteristic R peaks in the signal
- Sensitive to placement and muscle contractions

PPG (Photoplethysmogram):

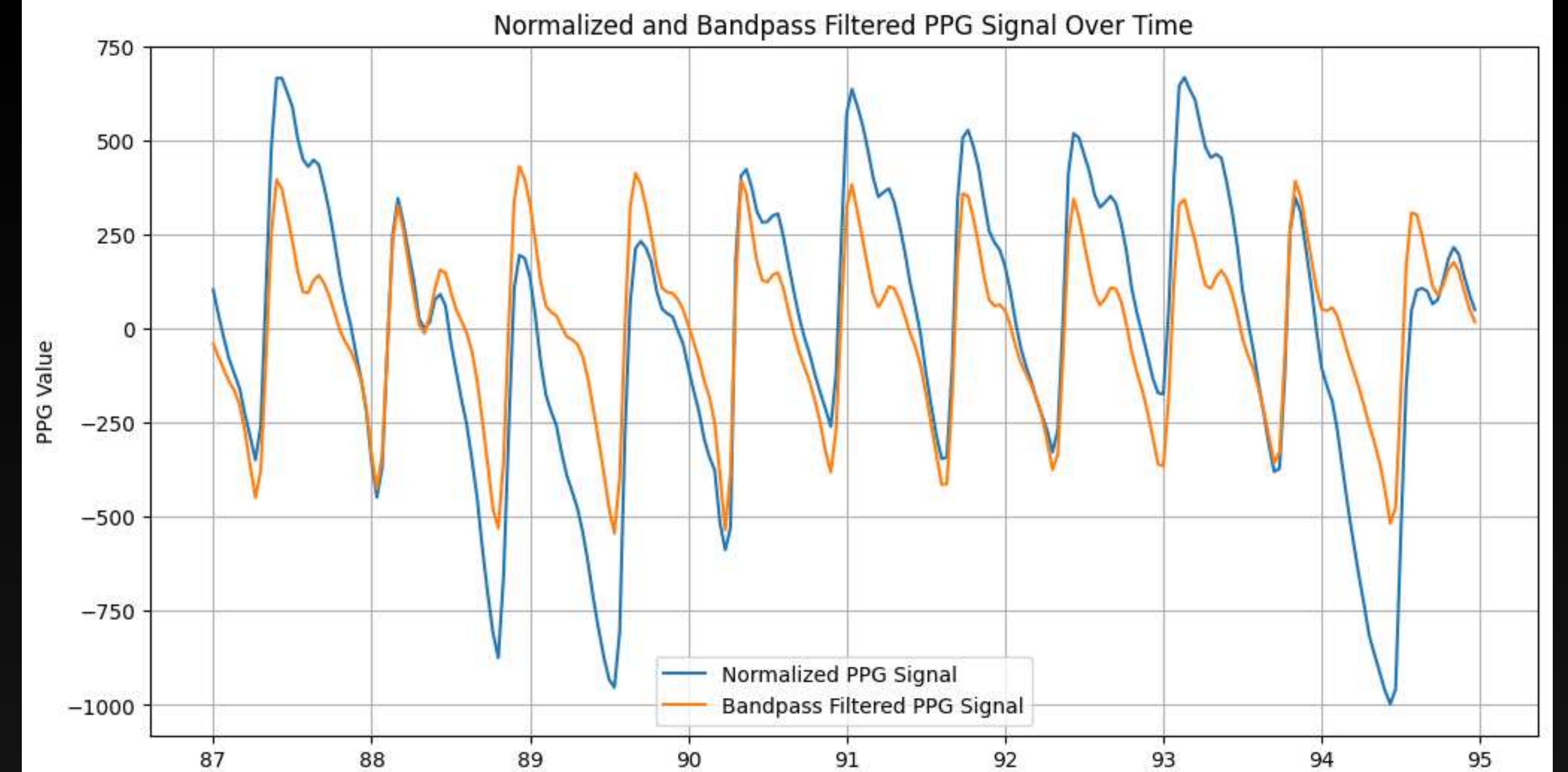
- Measures blood volume changes using light absorption
- Used for pulse oximetry
- Characteristic waveform shape
- Sensitive to sensor placement and motion artifacts

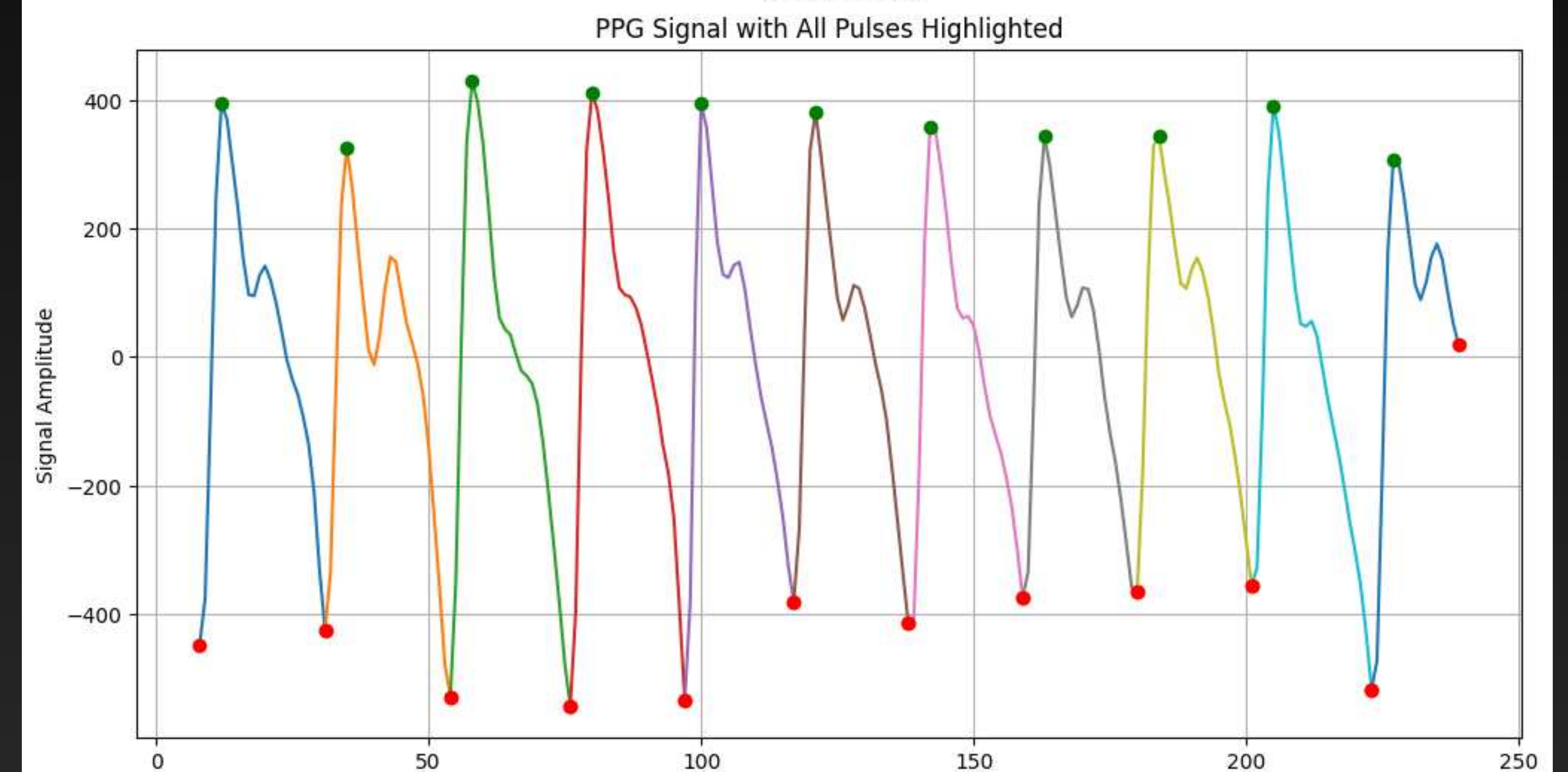


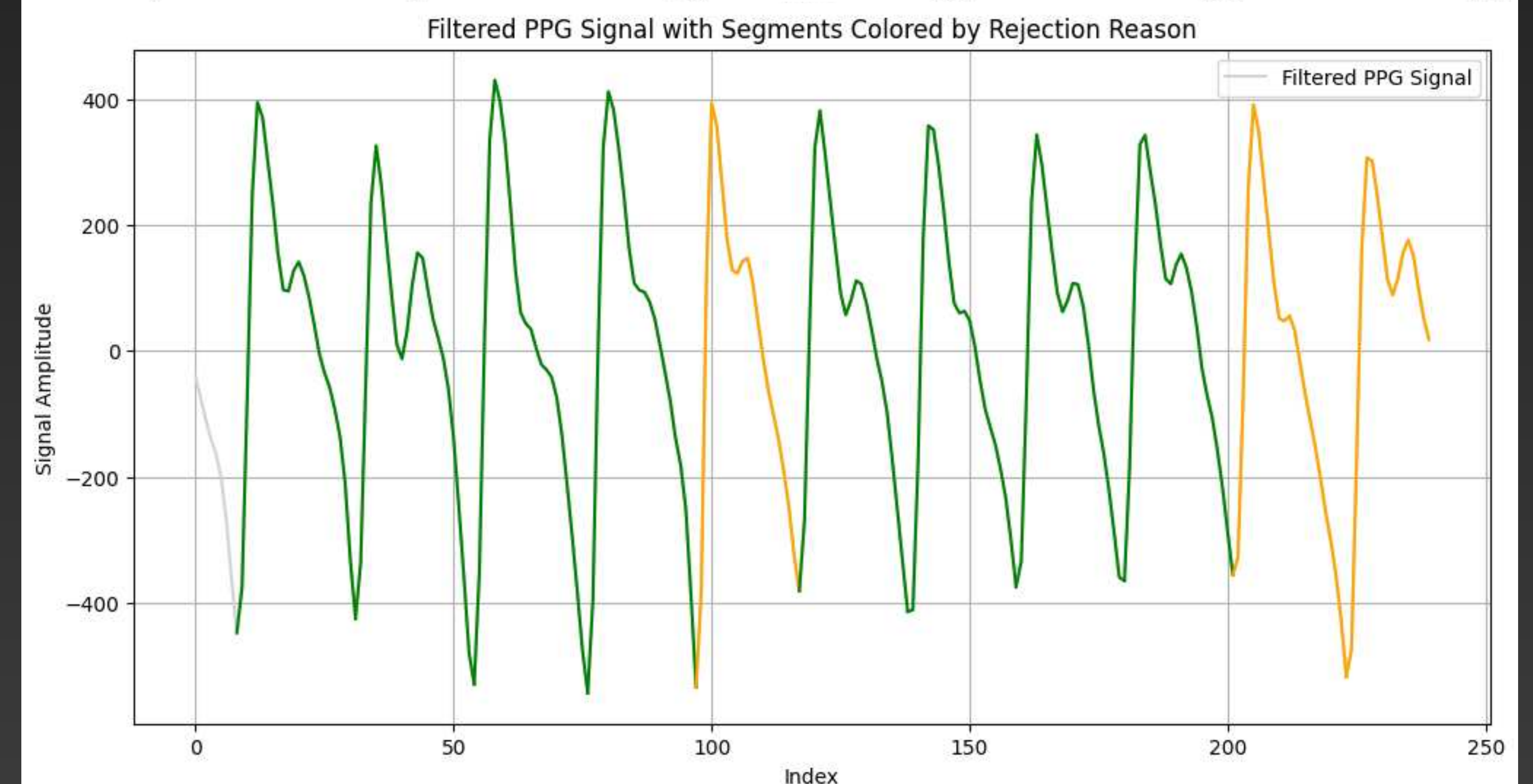


PPG Algorithm

- **Normalization**: Center the signal around zero by subtracting the mean.
- **Filtering**: Apply butterworth bandpass filter to remove noise and artifacts.
- **Pulse Detection**: Identify pulses using the lower envelope of the signal.
- **Validation**: Validate pulses based on amplitude, trough depth differences, and pulse width.
- **Results**: Estimate heart rate and assess signal quality based on valid pulse percentage

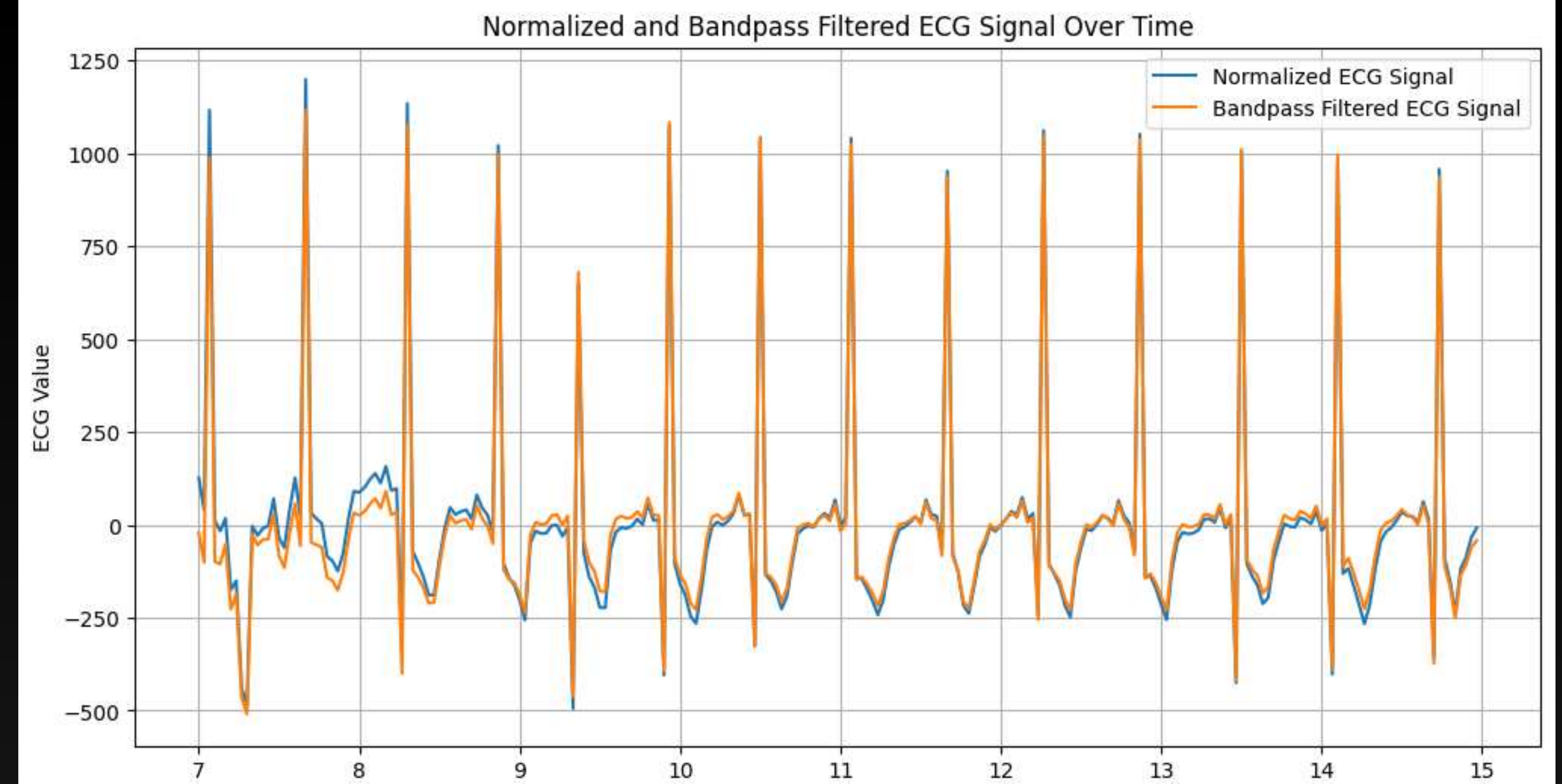


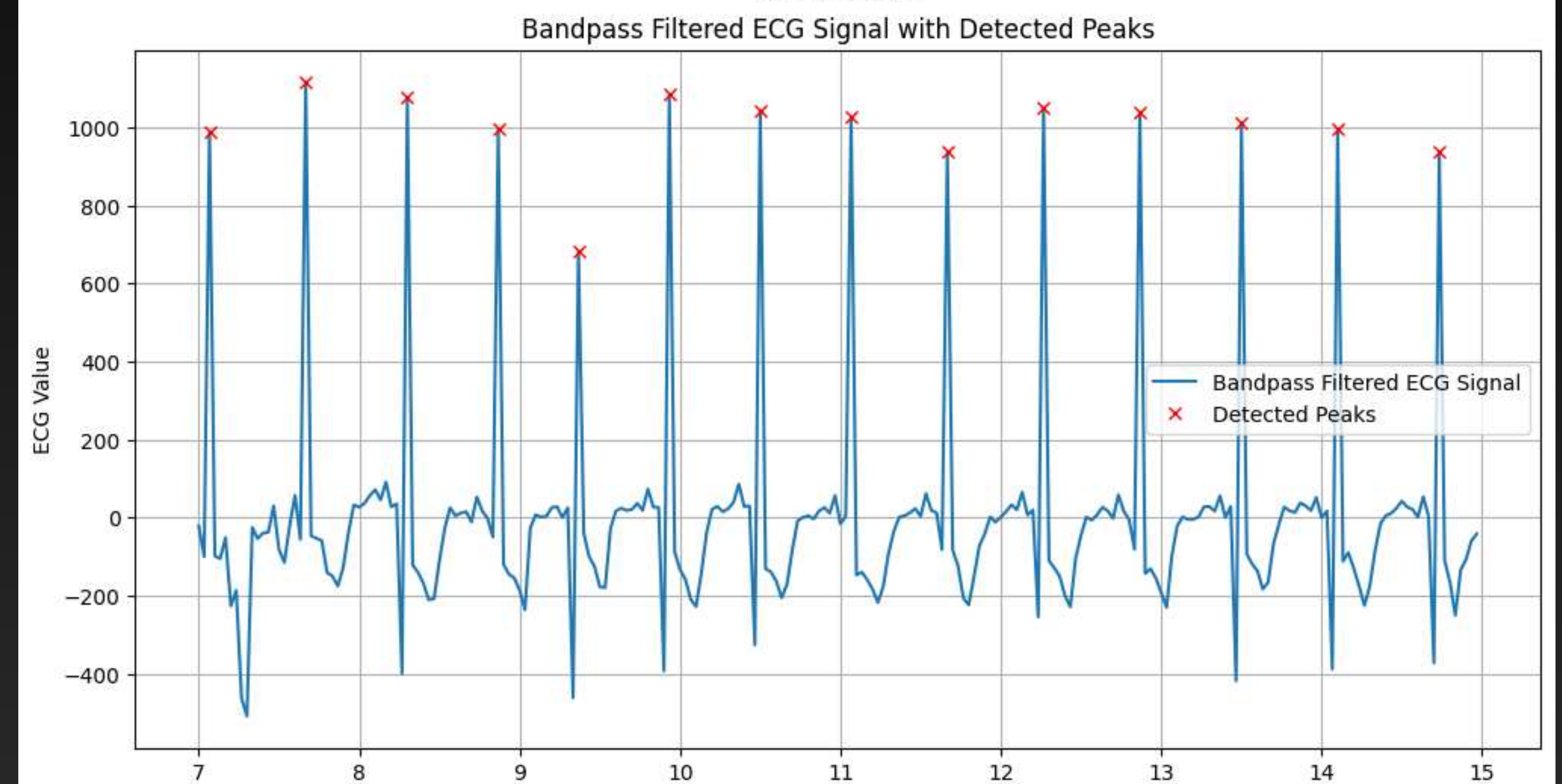


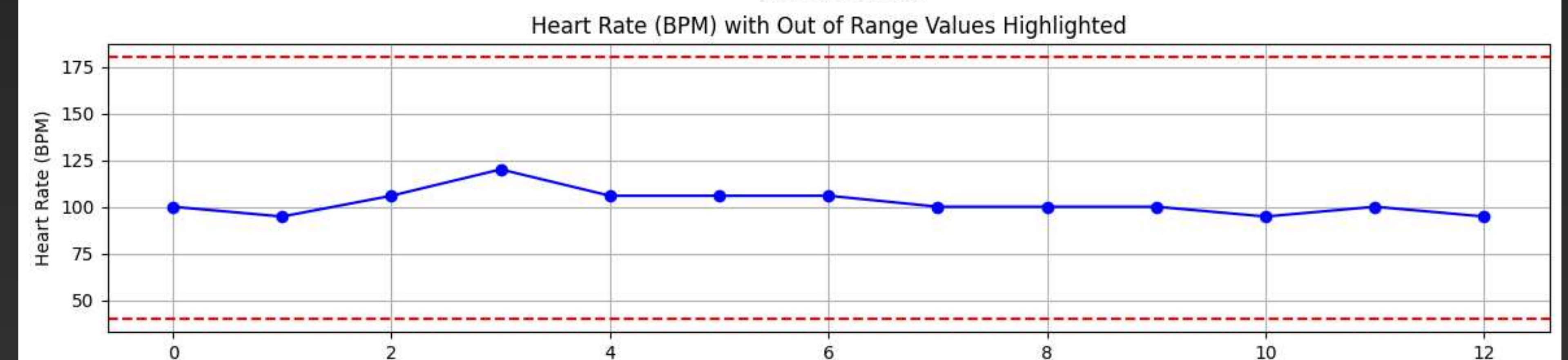


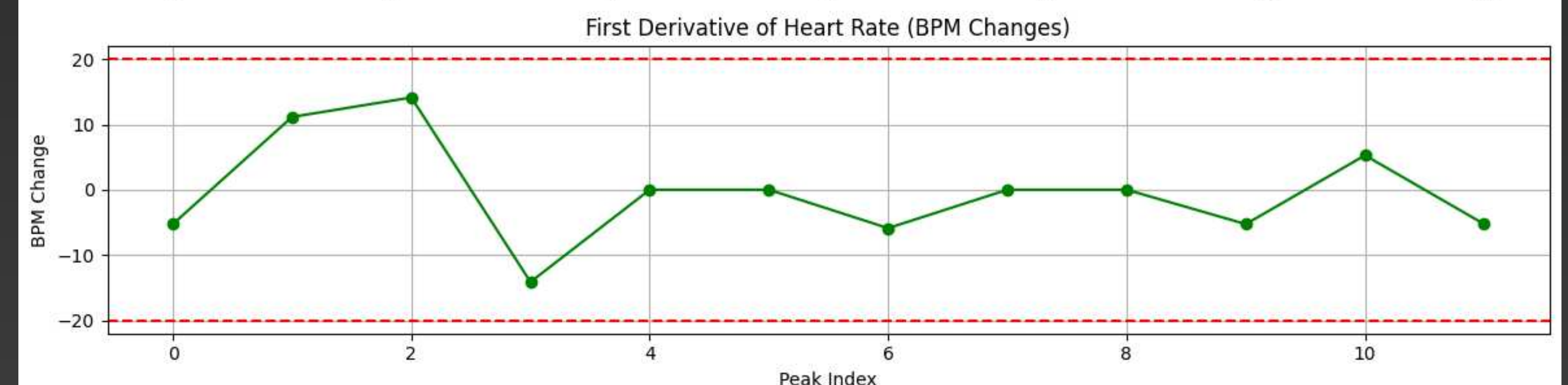
ECG Algorithm

- **Normalization**: Center the signal around zero by subtracting the mean.
- **Filtering**: Apply filters to remove noise and artifacts.
- **Median Absolute Deviation**: Calculate MAD as ground truth "noise level".
- **Peak Detection**: Identify peaks using the `find_peaks` crate, with MAD multiple as the threshold.
- **BPM Analysis and Validation:**
  - Analyze heart rate (HR) and changes in HR
  - Validate based on predefined thresholds
- **Results**: Estimate heart rate and assess signal quality based on valid pulse percentage







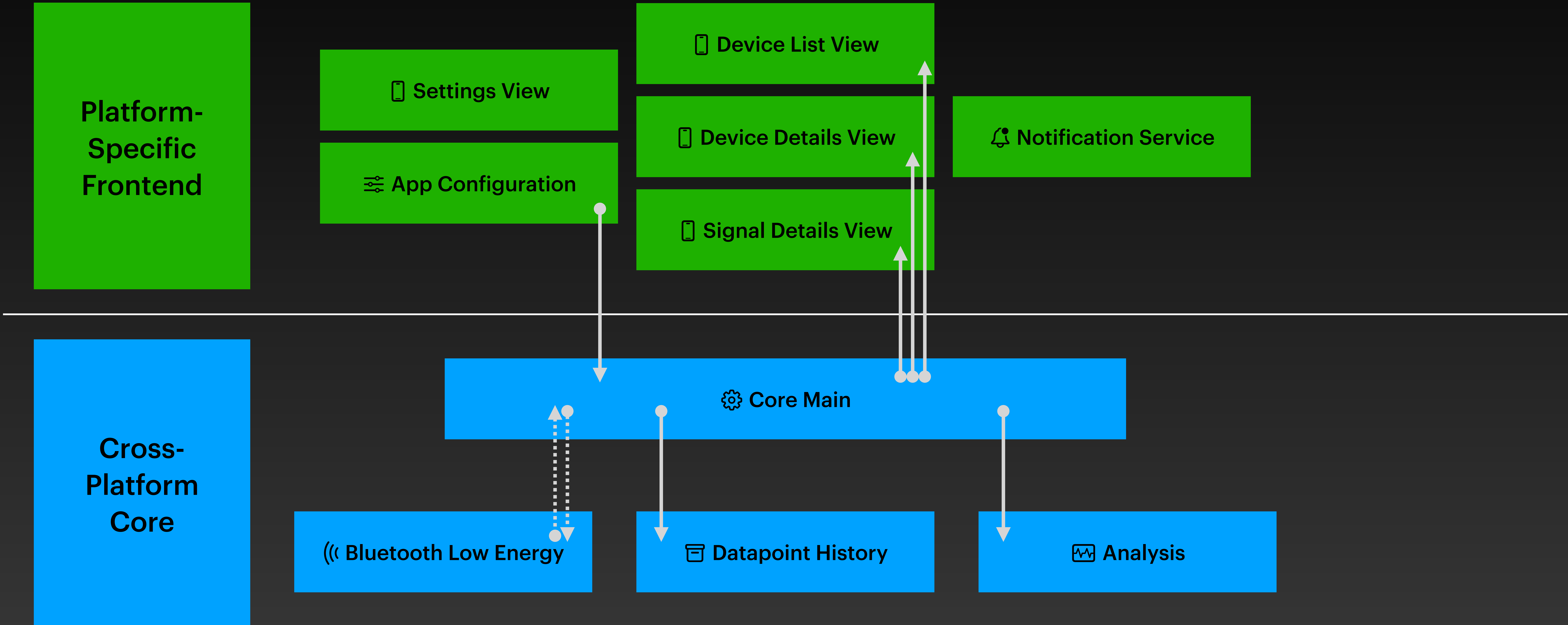


Analysis: From Python to Rust

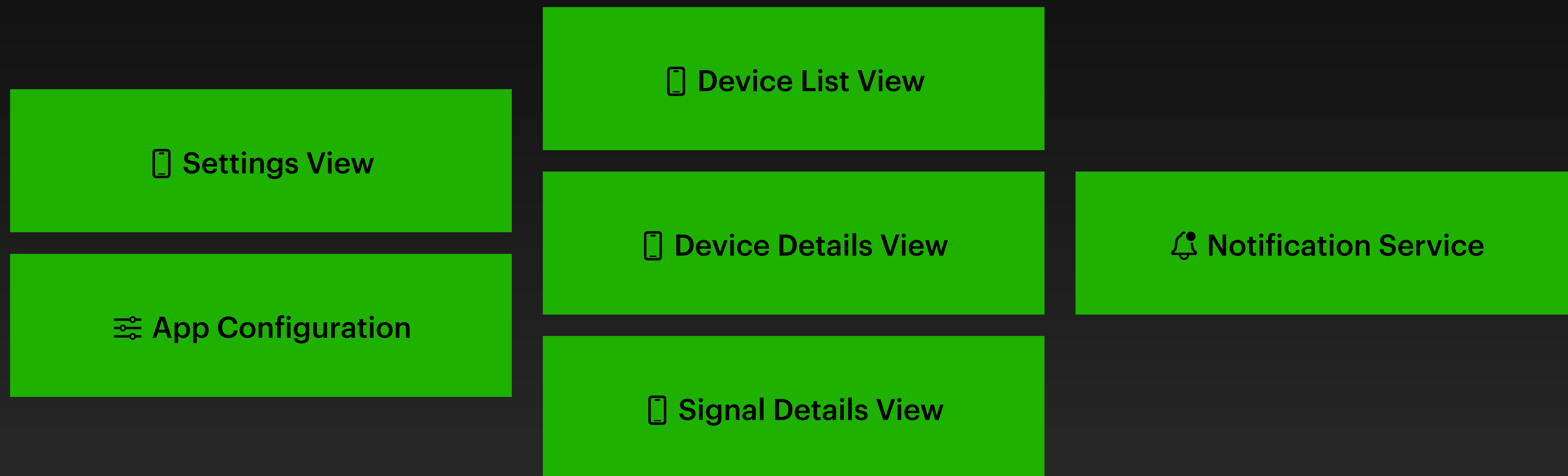
- Developed and parameterized in **Python**, straight forward
- **Filtering**: Uses `scipy` Butterworth filter
- **Peak Detection**: Uses `scipy` `find_peaks` with many params and good documentation
- **Plotting**: Inline in Jupyter notebooks with `matplotlib`

- Ported to **Rust** for performance and compatibility
- **Filtering**: Implemented using `biquad` crate and filter chaining
- **Peak Detection**: Using `find_peaks` reimplementation crate with fewer parameters
- **Plotting**: Hacky implementation using `plotters` crate

Architecture

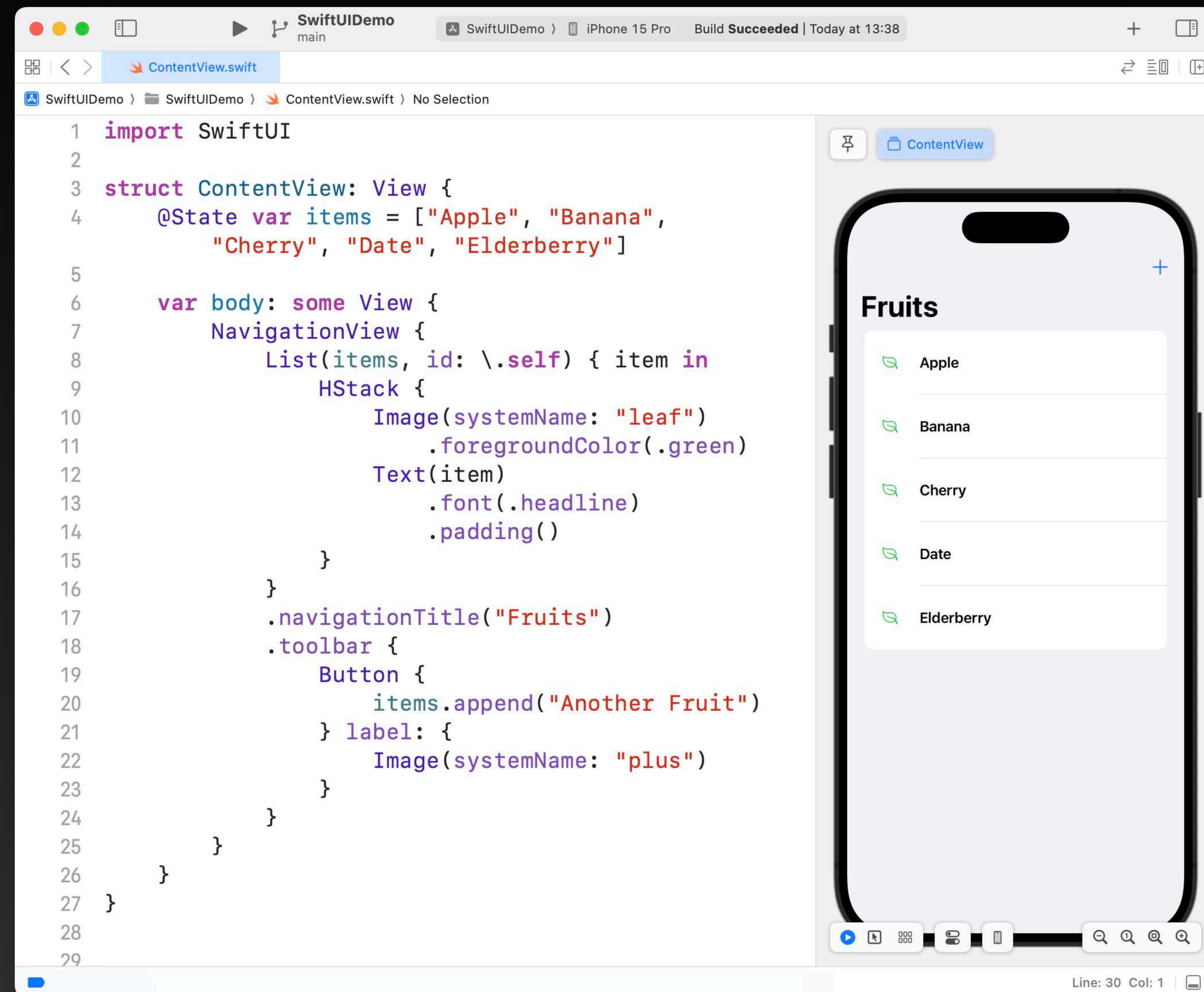


Architecture

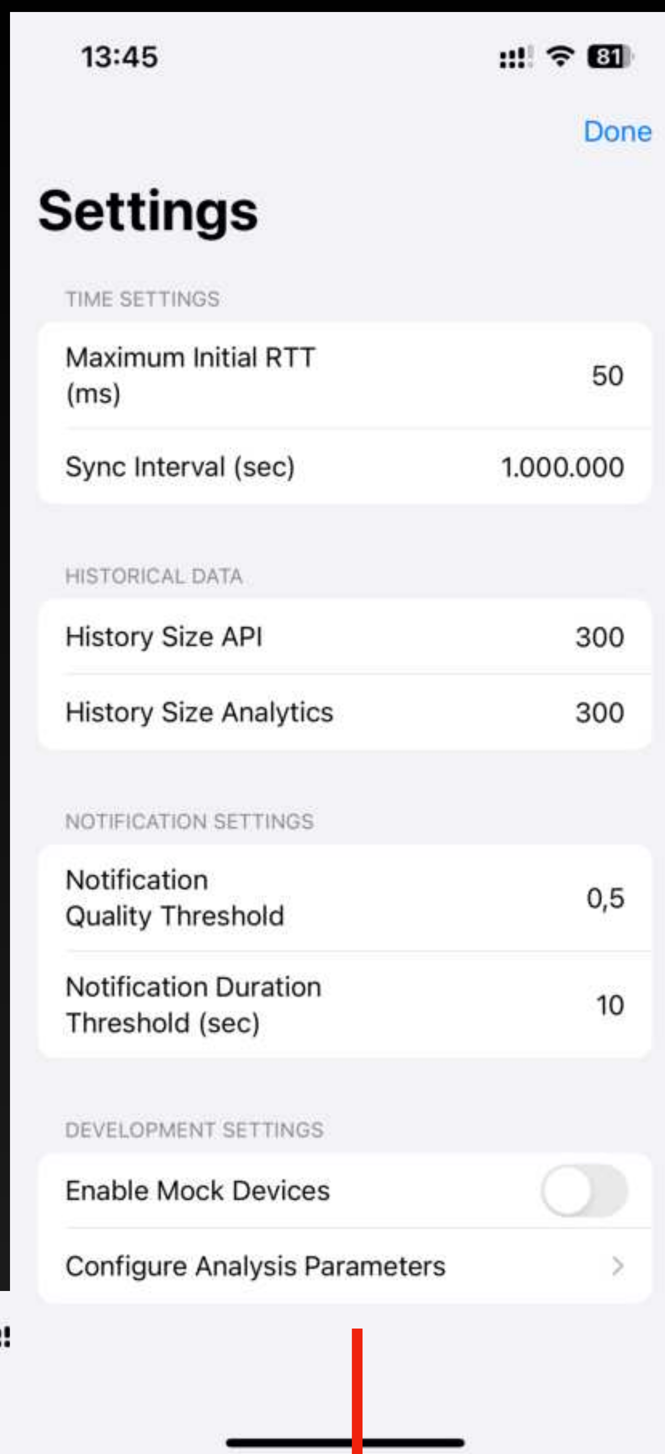


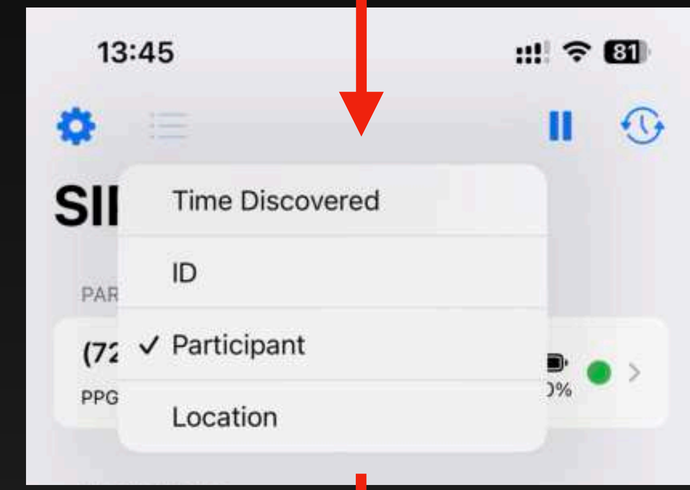
SwiftUI

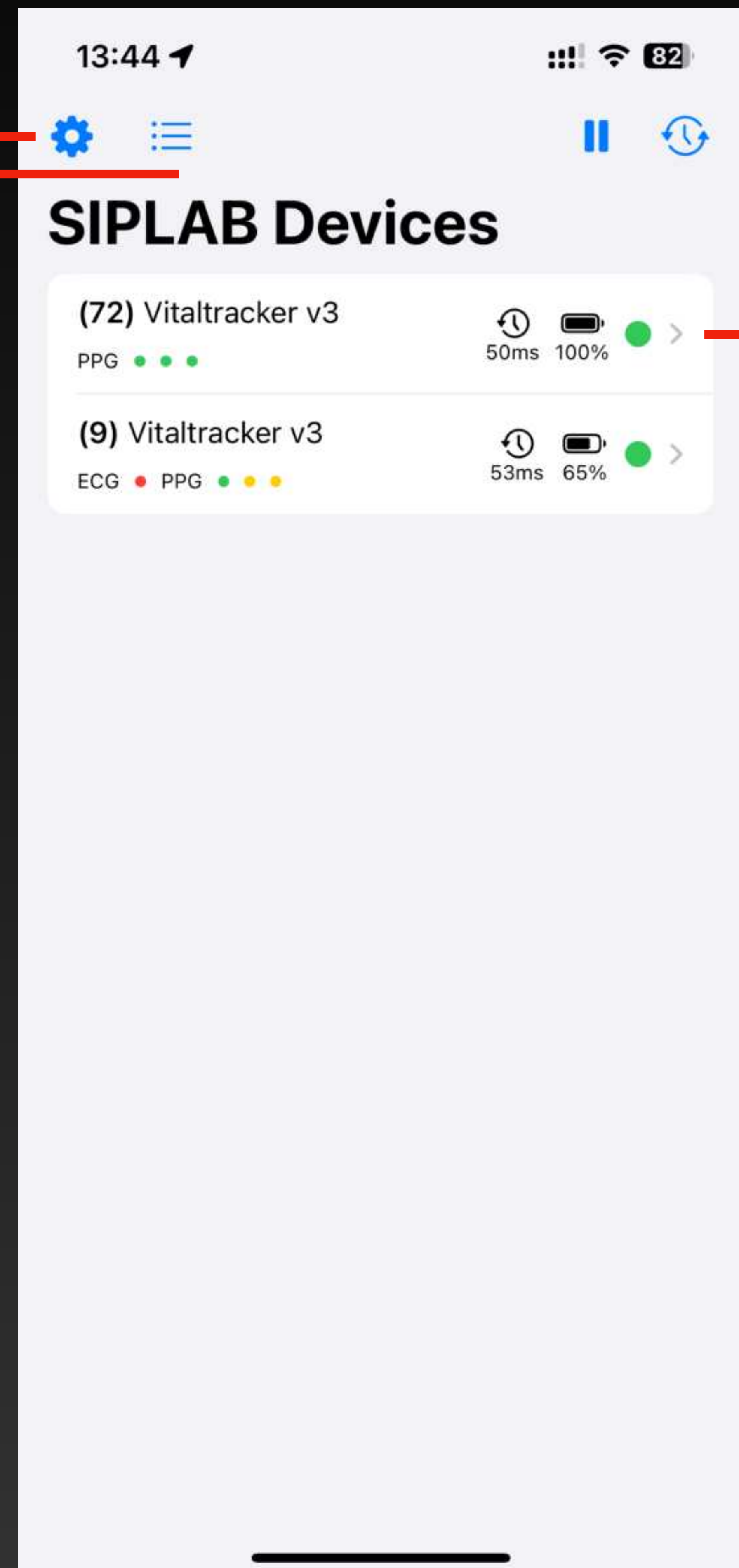
- SwiftUI: Modern UI toolkit by Apple for iOS, iPadOS, macOS
- Declarative Syntax: Write what the UI should do, SwiftUI handles the rest
- Data-Driven: Views update automatically with data changes
- Seamless with Swift, supports dynamic type, dark mode, localization, and accessibility
- Easy to create dynamic interfaces and animations

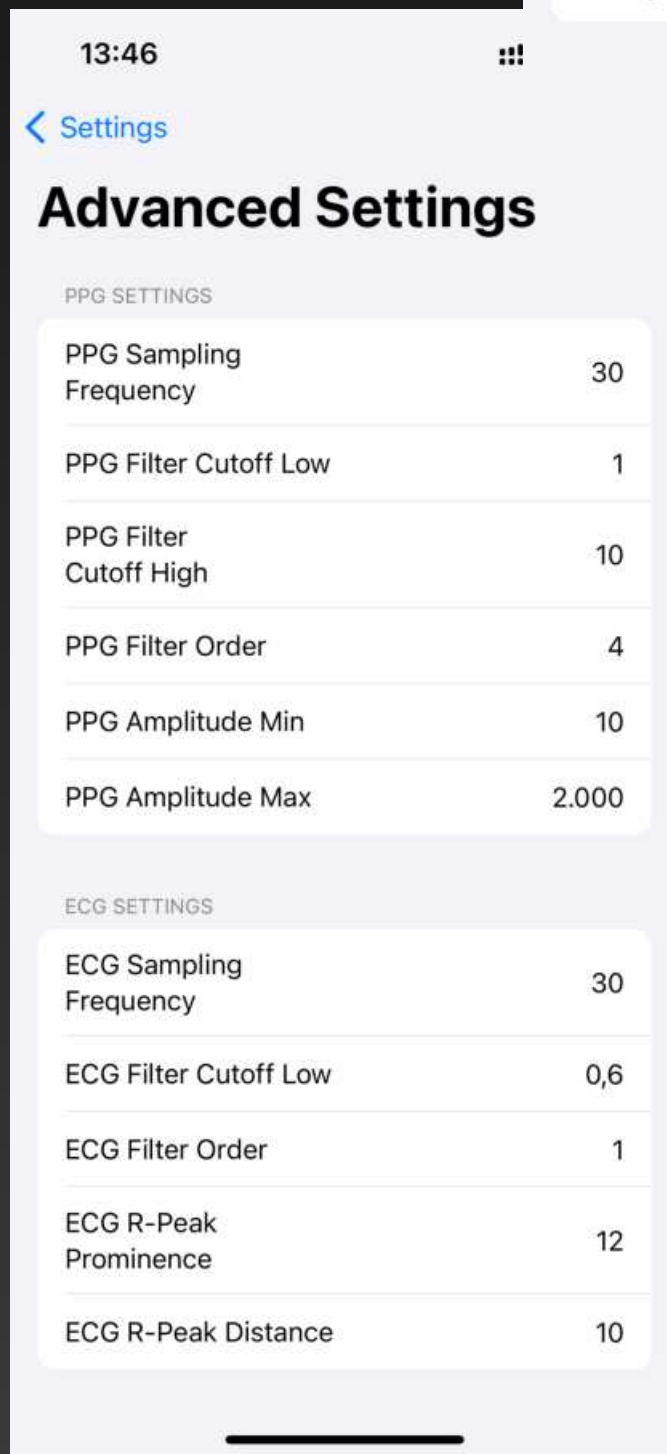


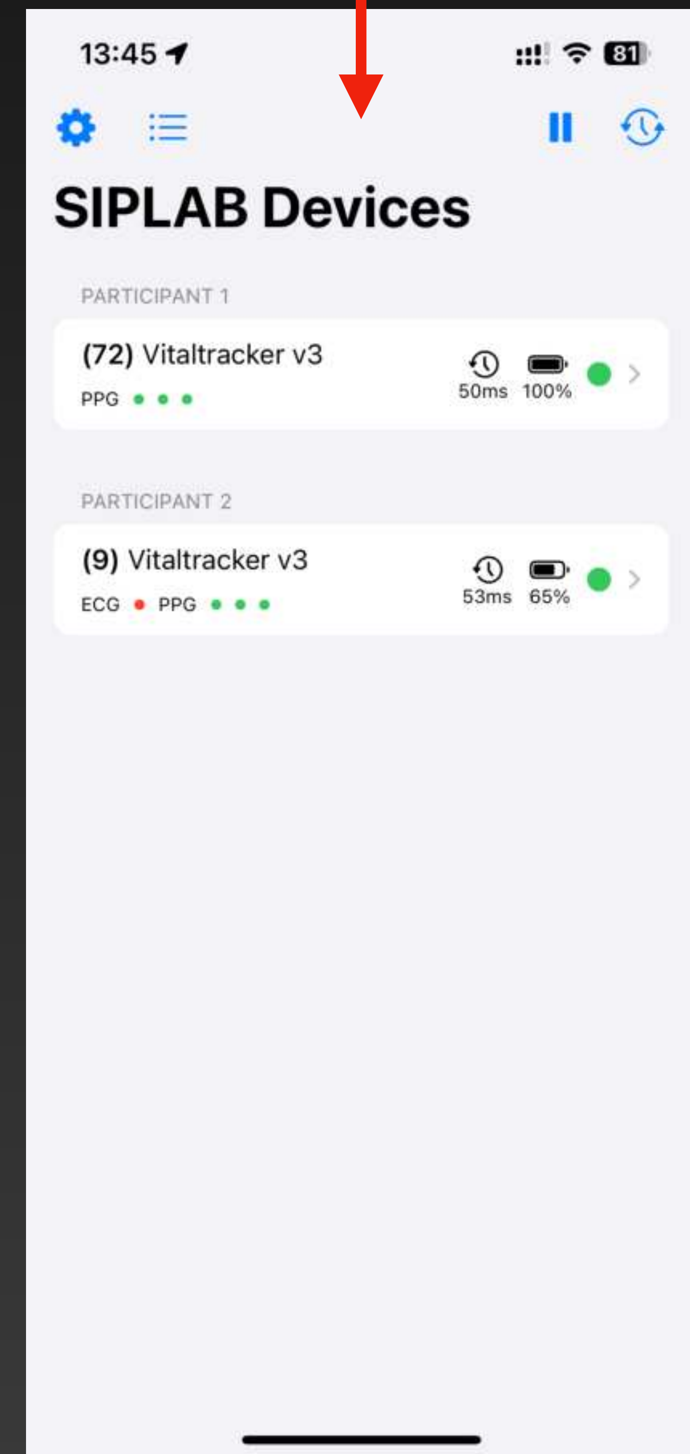
App Navigation

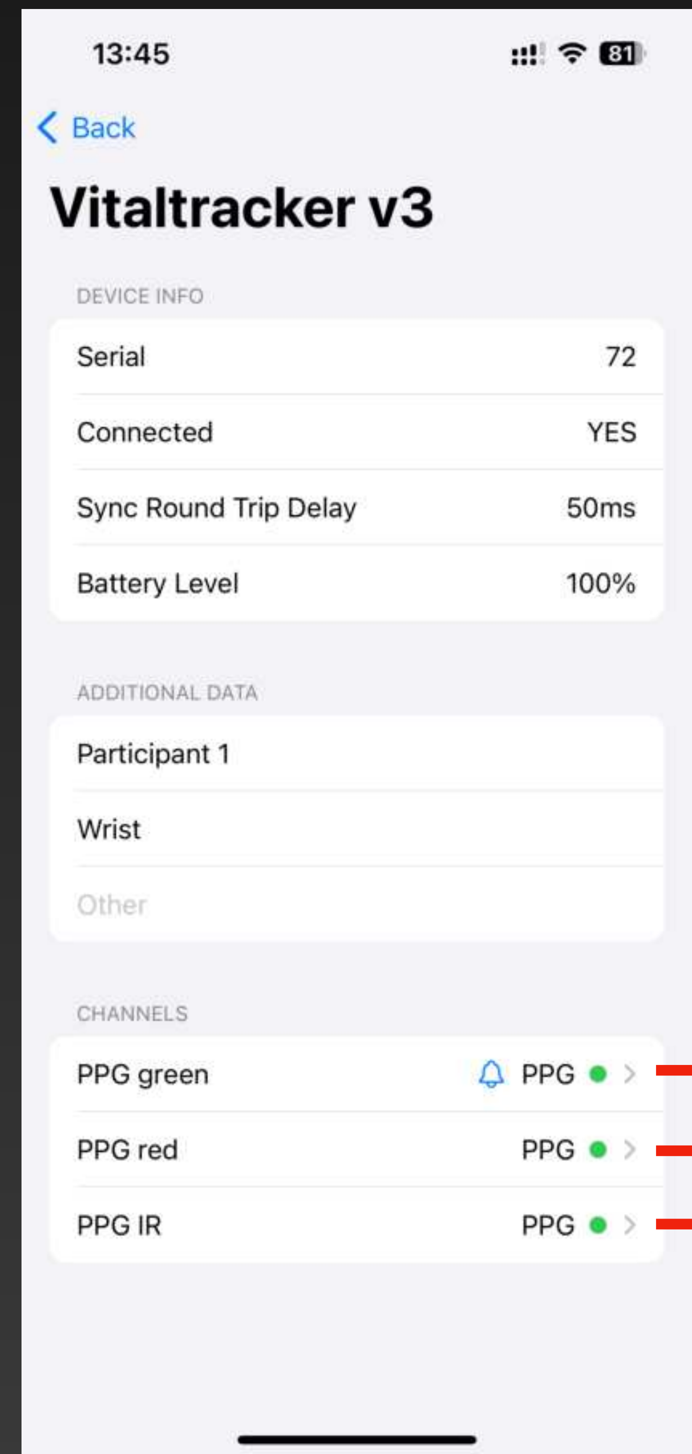




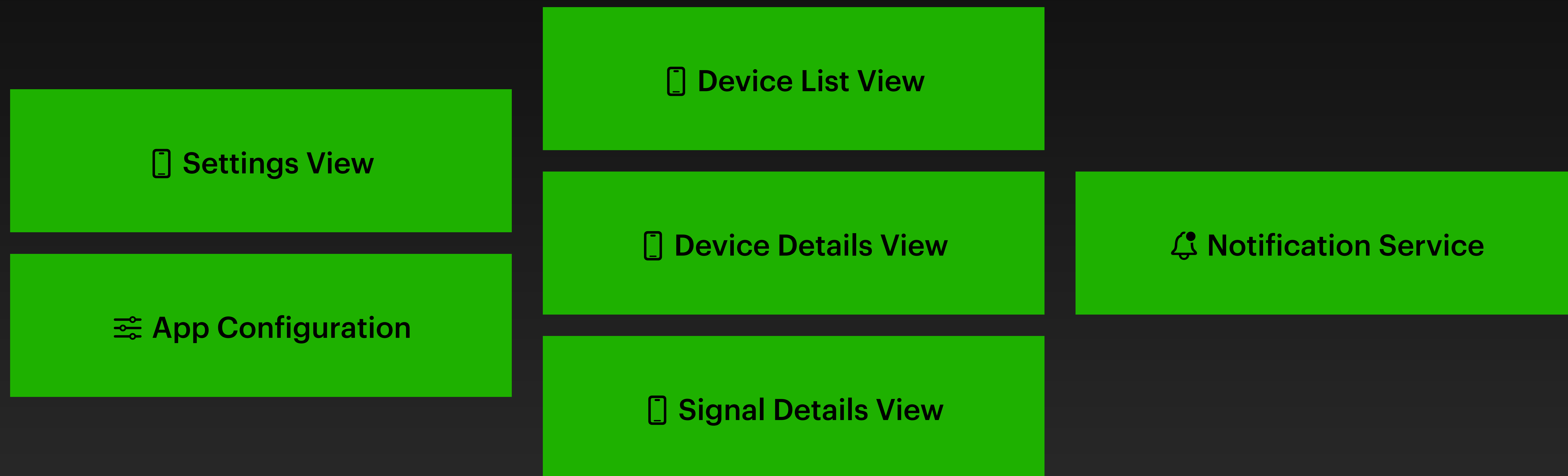






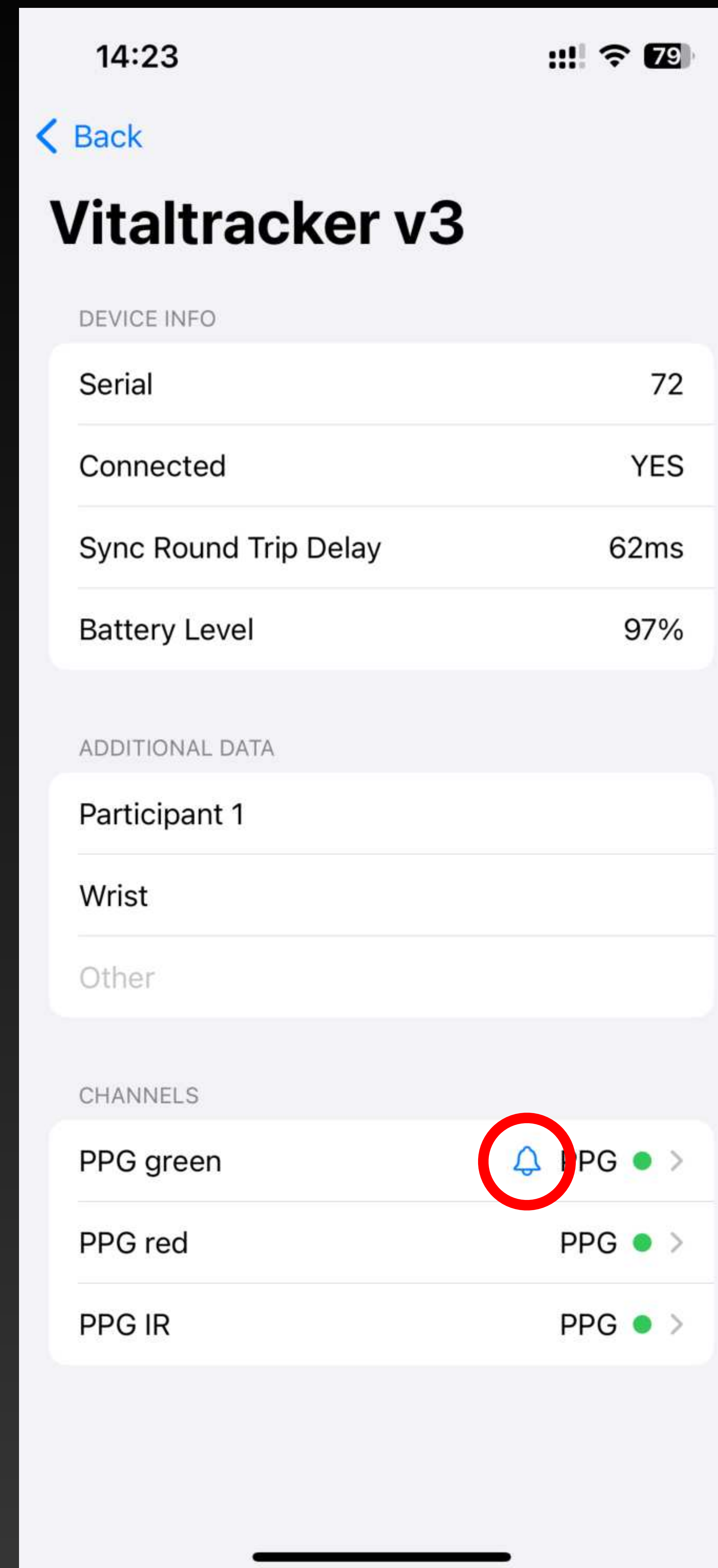


Architecture



Notification Service

- BLE streaming and signal analysis runs continuously, even in the background
- Requires `UIBackgroundModes: bluetooth-central`
- Channels can be marked "watched"
- Notification service subscribed to signal quality reports
- If signal quality falls below threshold for a defined duration, a notification is triggered

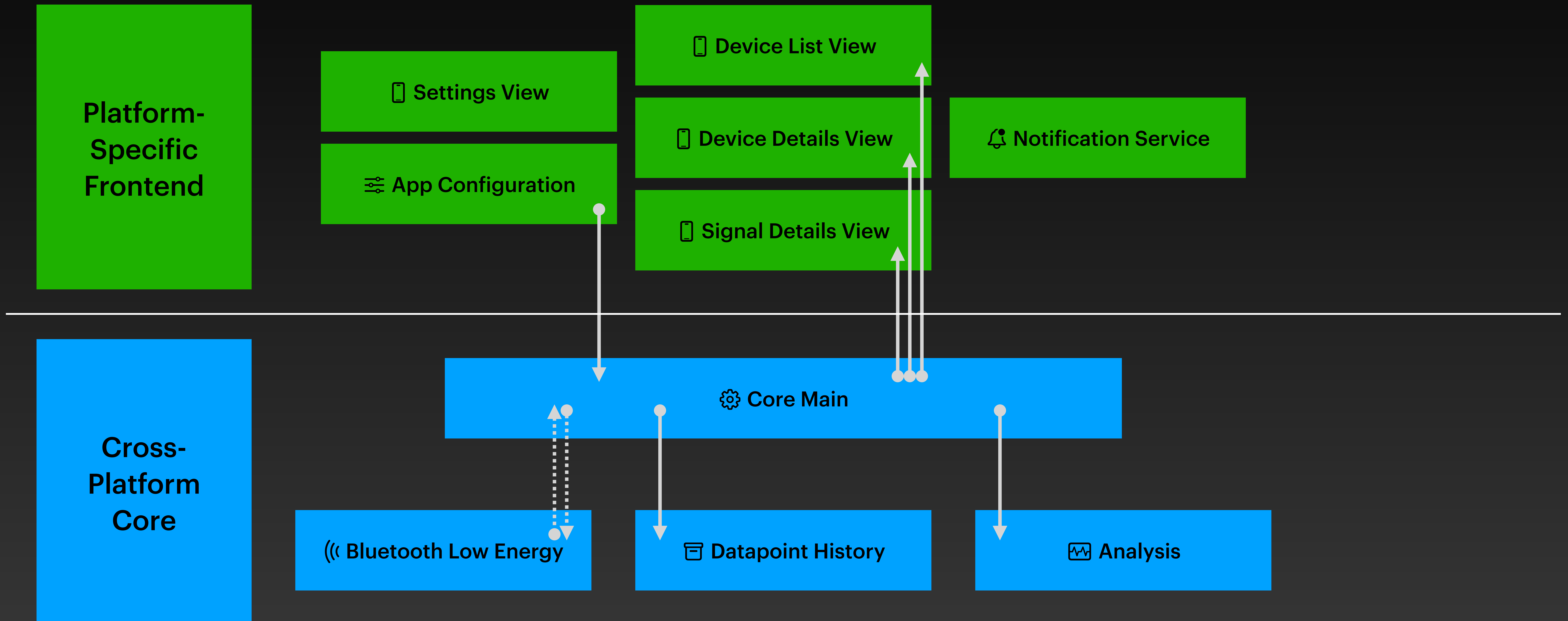


App Configuration

- Uses SwiftUI `@AppStorage`
  - Saves variables in iOS `UserDefaults`
  - Automatically updates views when values change
- AppConfig passed throughout the app, mapped into the core
- All components can access their needed configuration variables dynamically

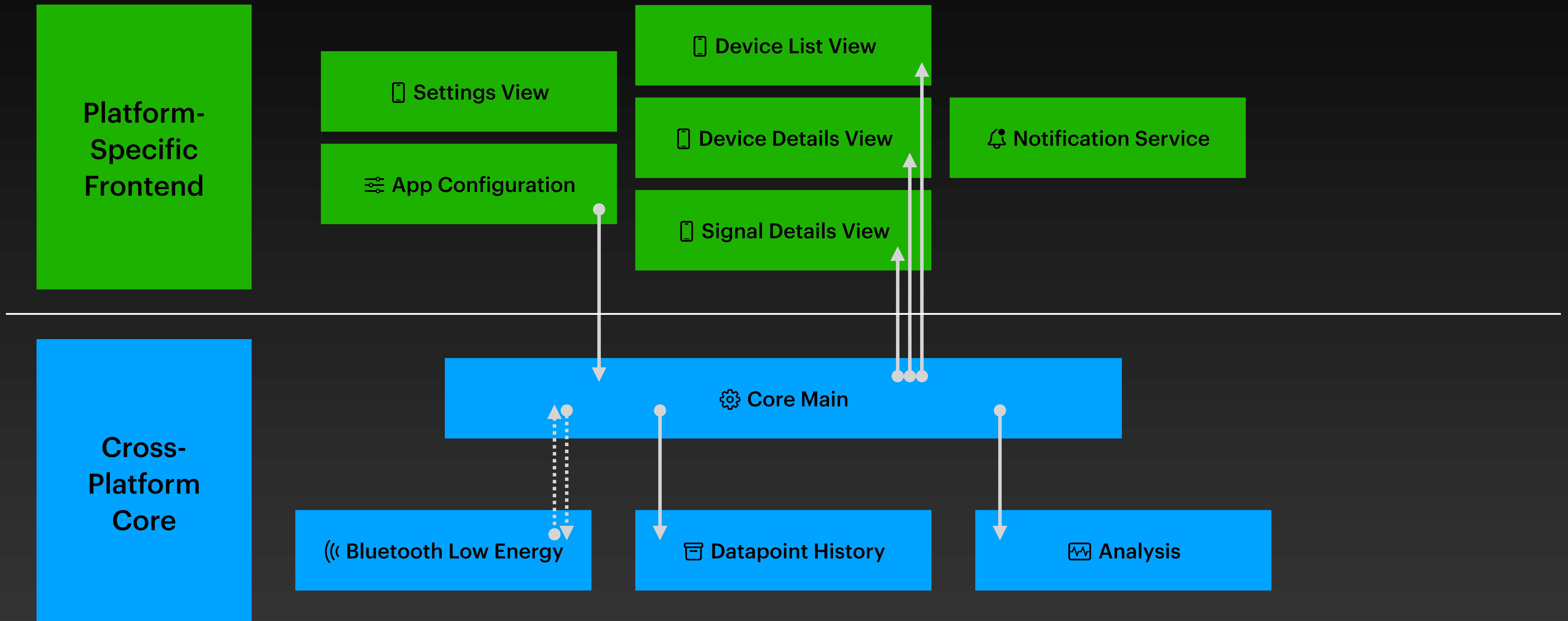
```swift
import SwiftUI

struct AppConfig {
    // Main Settings
    @AppStorage("histSizeApi")
        var histSizeApi: Int = 500
    @AppStorage("histSizeAnalytics")
        var histSizeAnalytics: Int = 500
    @AppStorage("maxInitialRttMs")
        var maxInitialRttMs: Int = 100
    @AppStorage("syncIntervalSec")
        var syncIntervalSec: Int = 60
    @AppStorage("analysisIntervalPoints")
        var analysisIntervalPoints: Int = 60
    @AppStorage("notificationQualityThreshold")
        var notificationQualityThreshold: Double = 0.5
    @AppStorage("notificationDurationThresholdSec")
        var notificationDurationThresholdSec: Int = 300

    // Configurable via Device Detail View
    @AppStorage("additionalDeviceData")
        var additionalData: [String: AdditionalDeviceData] = [:]

    ...
}
```

Architecture



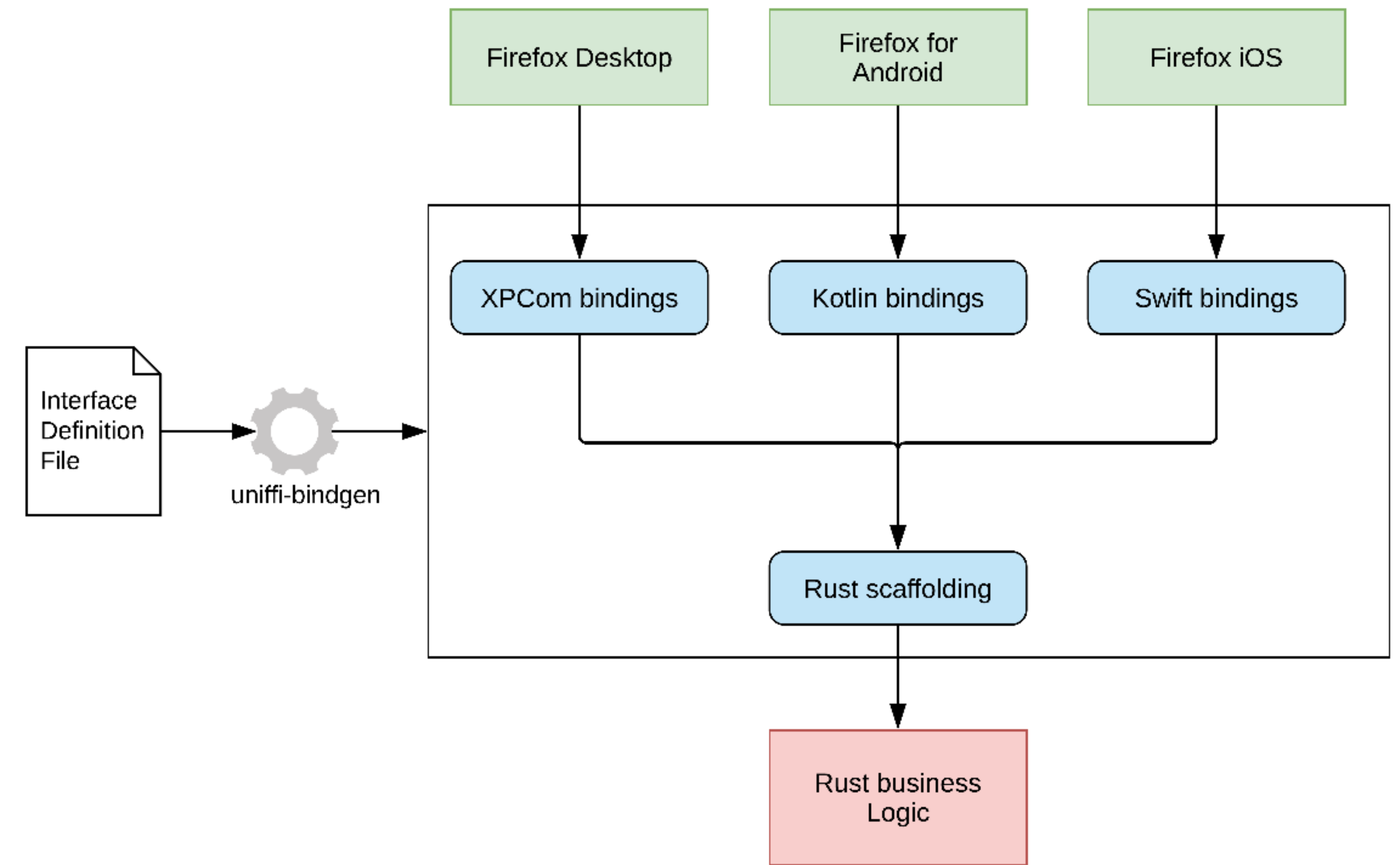
Live Demo

Architecture



uniFFI

- Developed by Mozilla and used extensively within Firefox
- Generates bindings ("translation layer") for Rust libraries to Swift, Kotlin, Python, etc.
- Ensures memory-safe cross-language calls
- Reduces boilerplate and errors in manual bindings




```
interface VVCore {
    constructor(VVCoreConfig config, VVCoreDelegate delegate);

    void start_ble_loop();

    void sync_time();

    void pause();

    void resume();
};
```

Conclusion

Summary:

- Developed an app for real-time visualization of wearable sensor data
- Integrated Rust backend with Swift frontend using uniFFI
- Implemented a nice User Interface, BLE communication, data storage, signal analysis, and notifications
- ~5000 Lines of Code
- Time Spent: too much...

Future Work:

- Develop Android frontend
- Enhance signal analysis algorithms

Lessons learned

- **Very satisfied with the architecture:** uniFFI made integration easy
- **Rust is powerful but challenging:** personal tip, learn Rust, but don't make your bachelor's thesis your first project ;)
- **Prototyping is beneficial:** initial prototype ready within a week, but required continuous improvement through weekly meetings

Q&A

Architecture