indium
Make Technology Work

Leveraging Flutter for Headless Systems: Empowering User Interfaces in Displayless Environments

A Whitepaper

Table of Contents

1. Executive summary

The demand for headless systems, frequently avoiding conventional displays, has seen a remarkable upswing in our dynamic technological landscape. These systems find application across a wide spectrum, from Internet of Things (IoT) solutions to embedded systems. Yet, an escalating necessity arises to furnish them with a user interface, facilitating seamless interaction and control. This is where Flutter, an open-source UI software development kit (SDK) curated by Google, emerges as a pioneering solution.

This white paper explores the potential of utilizing Flutter as the vanguard for headless systems. It delves into Flutter's pivotal features and advantages, showcasing why it stands out as the paramount choice for crafting immersive interfaces. By harnessing the capabilities of Flutter, we're poised to revolutionize user experiences in displayless environments, setting a new standard for interaction in this rapidly evolving digital era.

2. Introduction to Flutter

Introduced by Google in 2017, Flutter stands as a formidable framework empowering developers to craft applications with a native-like feel, all from a single unified codebase. At its core lies Dart, a contemporary and efficient programming language, that facilitates the creation of applications that compile into native code, ensuring optimal performance. Its distinctive approach to UI development sets Flutter apart, delivering high customizability and an aesthetically captivating user interface, elevating the user experience to unprecedented levels. This fusion of technology and design places Flutter at the forefront of modern app development, redefining what's possible in creating seamless, engaging user interfaces.

3. Challenges & Concerns

Designing user interfaces for headless systems without conventional displays poses a formidable challenge, impeding effective user interaction. The endeavor to retrofit native platforms for these specialized systems demands substantial time and customization efforts. What's required is a versatile and efficient framework tailored precisely to meet these distinctive demands.

Cross-platform compatibility introduces an added layer of complexity, necessitating interfaces that seamlessly adapt to diverse platforms and devices, thereby reducing the maintenance burden. To squarely confront these hurdles, a framework is indispensable; one that furnishes comprehensive tools, boasts an extensive library of UI components, and exhibits exceptional performance. Flutter steps forth as a compelling solution, projecting mobile app-like views onto headless systems, thereby bestowing intuitive and interactive interfaces with impressive efficiency. This innovative approach promises to reshape the landscape of user interaction in headless environments, offering users a seamless and dynamic experience across various platforms.

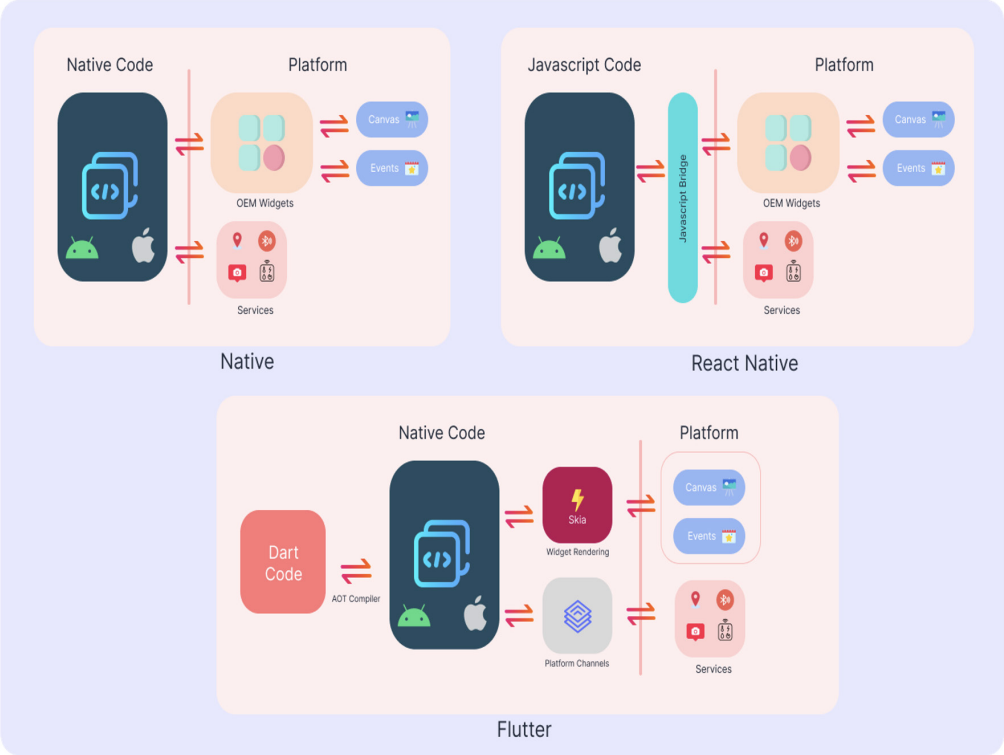
Resource Constraints: Headless systems often operate with limited resources, including processing power and memory. This necessitates an efficient framework that can deliver optimal performance without overtaxing the system's capabilities.

3.1 Accompanying Considerations:

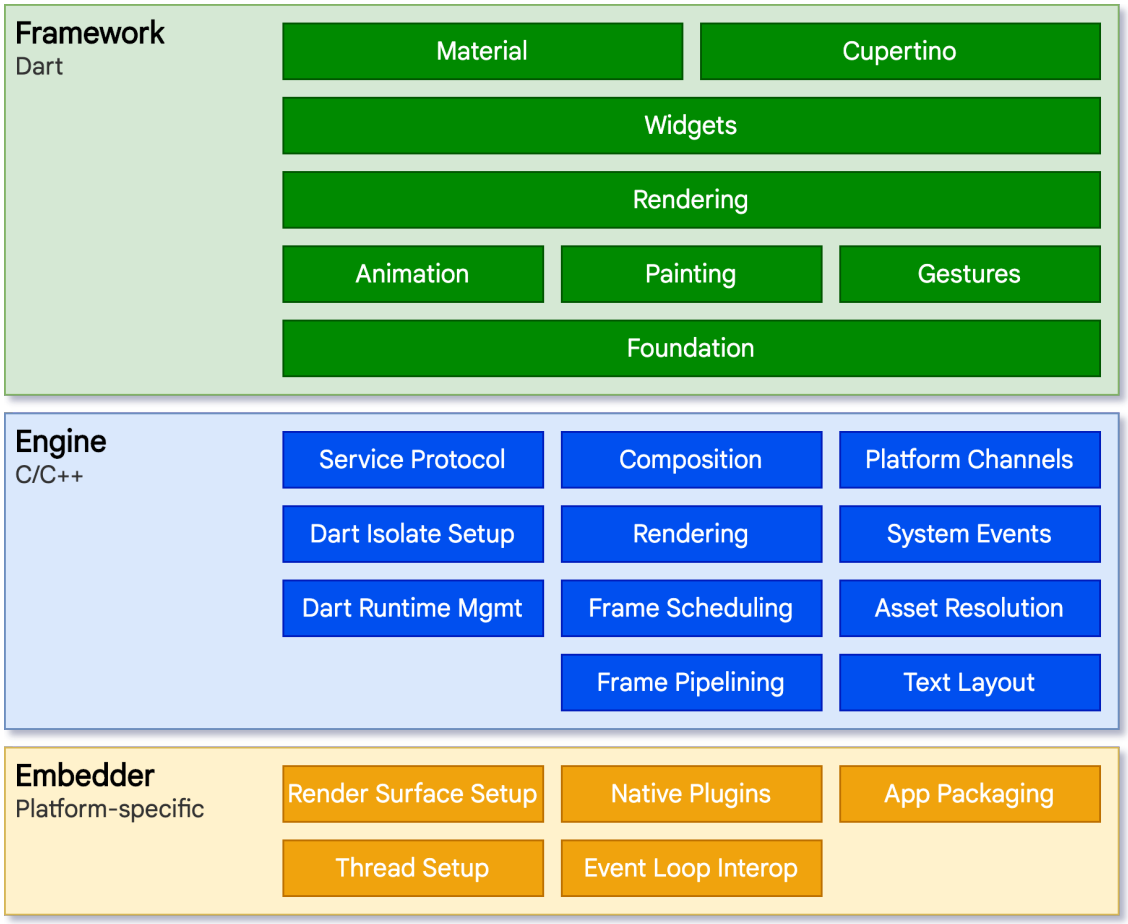
Resource Optimization:	Ensure peak performance within system constraints without overtaxing resources.
Inclusive Design:	Implement accessibility features for diverse users, including those with disabilities.
Real-time Updates:	Enable seamless real-time data synchronization with other devices or servers.
Security Best Practices:	Address unique security concerns with secure communication protocols.
Scalability Assurance:	Adapt to new hardware and platforms, ensuring long-term viability.
Community Support:	Leverage a thriving developer community and extensive documentation.

4. Limitations of The Traditional Approach

When contrasting native apps with cross-platform solutions like React Native and Flutter, it's crucial to delve into their underlying architectures. Native apps on Android and iOS lean on built-in widgets from the respective platforms. When additional functionalities are needed, platform-specific native APIs become essential. React Native employs a bridge to interface with native features while maintaining a uniform interface. This can introduce performance considerations, particularly for intricate UI elements. Flutter boasts its robust rendering engine, Skia. This empowers it to construct the UI independently of the underlying operating system. The result is meticulous control over each pixel, facilitating the creation of highly customizable, visually captivating interfaces. Flutter compiles directly into native ARM (Advanced RISC Machine) code. This circumvents the necessity for a bridge, leading to accelerated performance and responsiveness. This comparison underlines the distinctive approaches and implications of native development and cross-platform solutions, shedding light on the advantages and limitations of each.

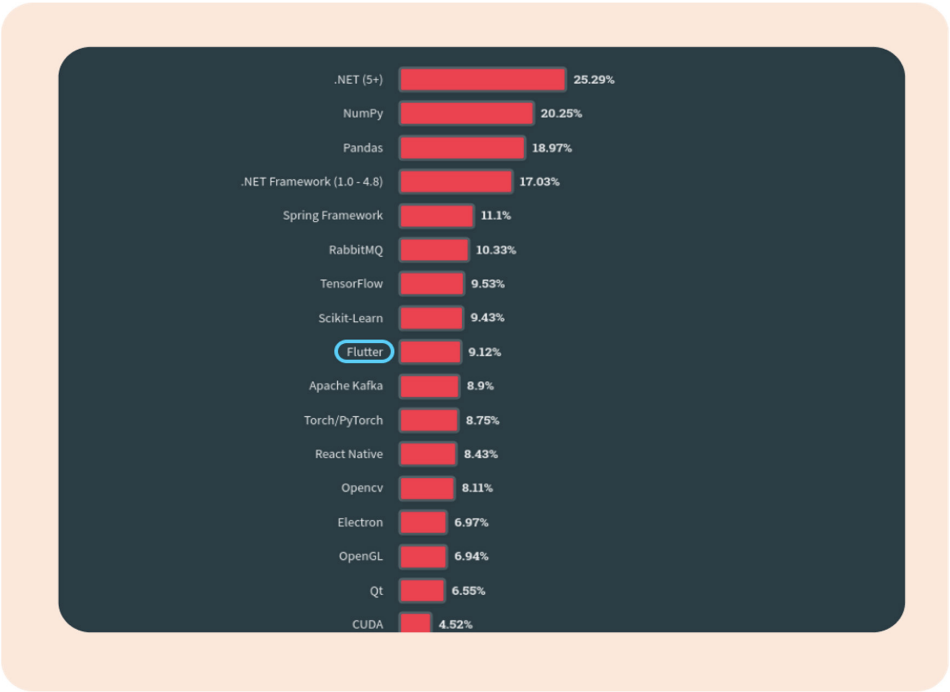


Below is an in-depth exploration of Flutter's low-level architecture, providing a comprehensive understanding of how it operates at a fundamental level.



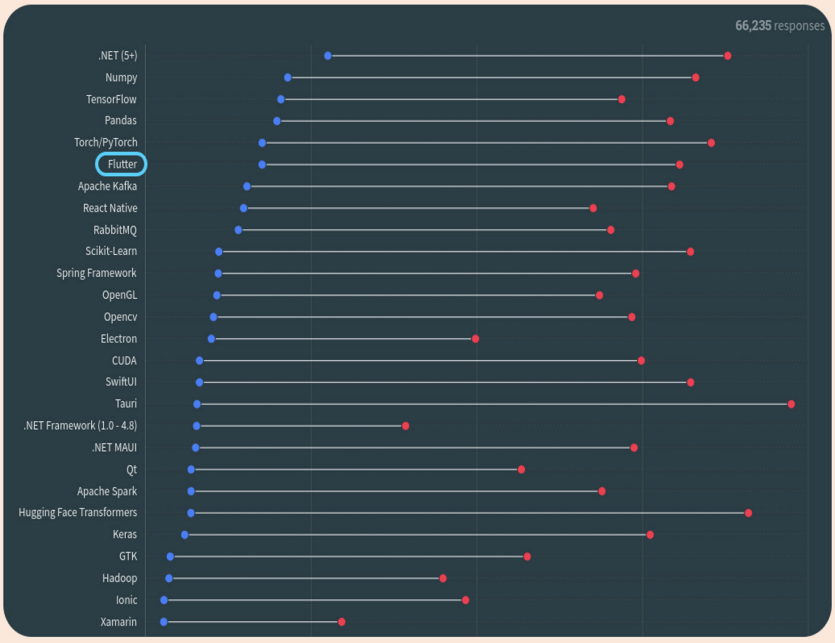
(Note: At the time of publishing this white paper, Flutter's latest rendering engine, Impeller, is still in the beta phase for Android. Therefore, no comments can be made on its performance and capabilities.)

Furthermore, Flutter has gained significant traction and recognition among developers recently. In the latest Stack Overflow Developer Survey, Flutter scored 9.12%, securing the 9th position as one of the most popular technologies. This survey reflects the growing adoption and interest in Flutter as a preferred framework for application development.



Additionally, Flutter stands strong in admiration and desirability, with 14.04% of respondents admiring the framework and an impressive 64.43% expressing their desire to work with Flutter.

These statistics showcase developers' confidence and enthusiasm for Flutter, a robust and reliable framework for building interfaces.



5. Real-world Applications and Diverse Use Cases

5.1 Situation:

An architecture was sought to fulfill several objectives in response to the client's requirements. These encompassed establishing seamless communication between the program and the model, enabling dynamic real-time switching of machine learning models, providing immediate access to sensor readings (CO2, O2, temperature, and humidity), and implementing live security camera functionality. Our proposed solution centered around the versatile Flutter framework, known for crafting intuitive, cross-platform user interfaces.

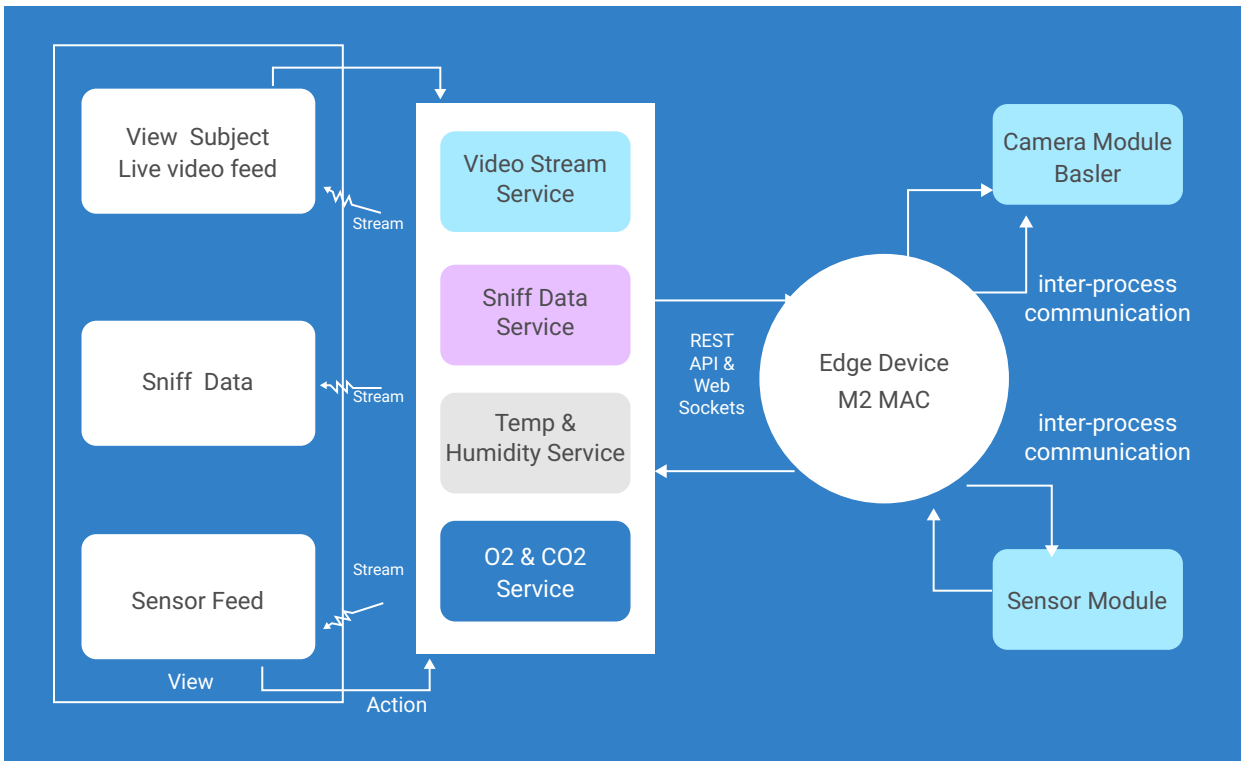
Leveraging Flutter's capabilities allowed us to create a unified, user-friendly interface that seamlessly functions across various devices and operating systems. The framework's hot-reload feature expedited development, enabling rapid iteration and refinement of the user experience.

Choosing Flutter also enabled us to harness its extensive widget library, ensuring our application met our client's technical requirements and delivered a visually appealing and engaging user interface. This technology choice significantly accelerated development, resulting in a more efficient and cost-effective solution.

In the forthcoming sections, we will delve deeper into the technical intricacies of our Flutter-based architecture, shedding light on how it enabled us to efficiently achieve the client's objectives while maintaining a user-centric approach throughout the project.

5.2 Comprehensive Project Exposition

5.2.1 Process Diagram



5.2.2 Section 1: Problem Formulation

Every successful app project commences with a precisely defined problem. Here, we address the challenge of olfaction detection and detail our approach to resolving it. This involved breaking it down into sub-problems, extensive research for solutions, and carefully selecting the appropriate technology stack. The app's tasks encompass displaying continuous smell data, streaming a real-time camera feed, and facilitating model selection. Before embarking on these tasks, it was imperative to establish direct communication with the edge device (SBC – Single Board Computers). Given the app's real-time demands, any delay surpassing 2 seconds was deemed unacceptable, necessitating the exclusion of an intermediate router.

5.2.3 Section 2: The Approach

To tackle the challenges in Section 1, we adopted a strategic approach to establish efficient communication between our application and the edge device, enabling real-time task execution.

Initially, we considered two potential solutions for linking the edge device to the mobile application: utilizing the mobile device as a hotspot and connecting the edge device to it, or using the Edge Device's (Mac's) hotspot and connecting the mobile device to it. To access any service or application on a machine, we required two key elements: the **IPv4 (RFC 791)** address and the port number. While we possessed the latter (port number) while managing both the front and back end, obtaining the IP address presented a challenge. Android (API >29) restricts retrieving the device's IP connected to its own hotspot.

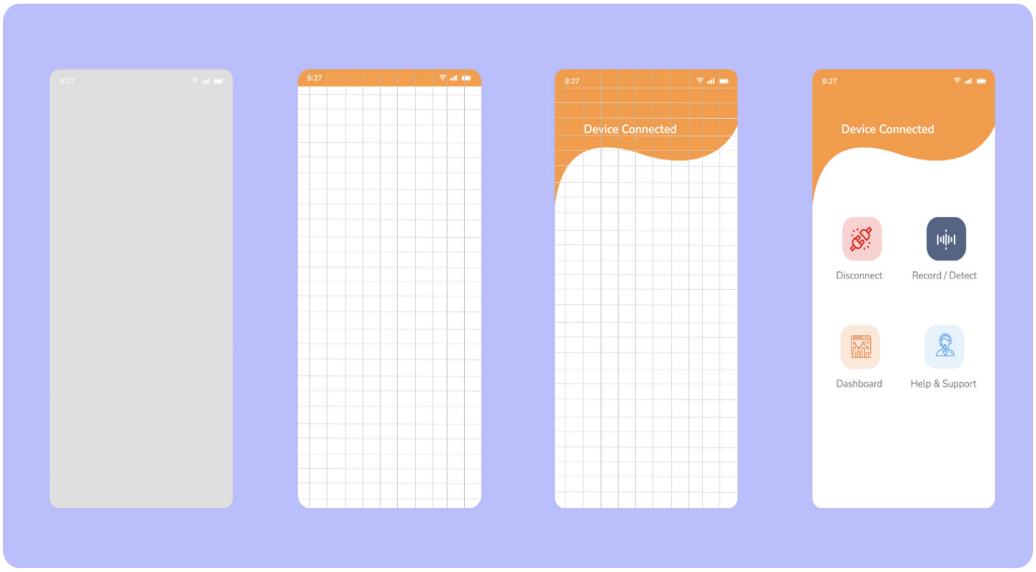
Transitioning to the second solution initially seemed straightforward. However, we encountered a connection issue when attempting to connect to the server using a dummy app. After nearly an hour of investigation, it became apparent that the **IPv4 (RFC 791)** address we attempted to reach belonged to a different network. This underscores a fundamental networking concept— a device can have multiple IP addresses for each connected network. This prompted us to quest to ascertain the server's IP from the app's perspective. Fortunately, we were able to accomplish this without significant difficulty.

5.2.4 Section 3: Software Development

Flutter offers a rich repository of built-in widgets encompassing Layout, Material Design (Google's Material Design), Cupertino (Apple's iOS design guidelines), Media, Animation, and more. When skillfully combined, these widgets empower the creation of visually striking UIs. Allow us to showcase the formidable capabilities of Flutter's widget system through the illustration of a UI element.

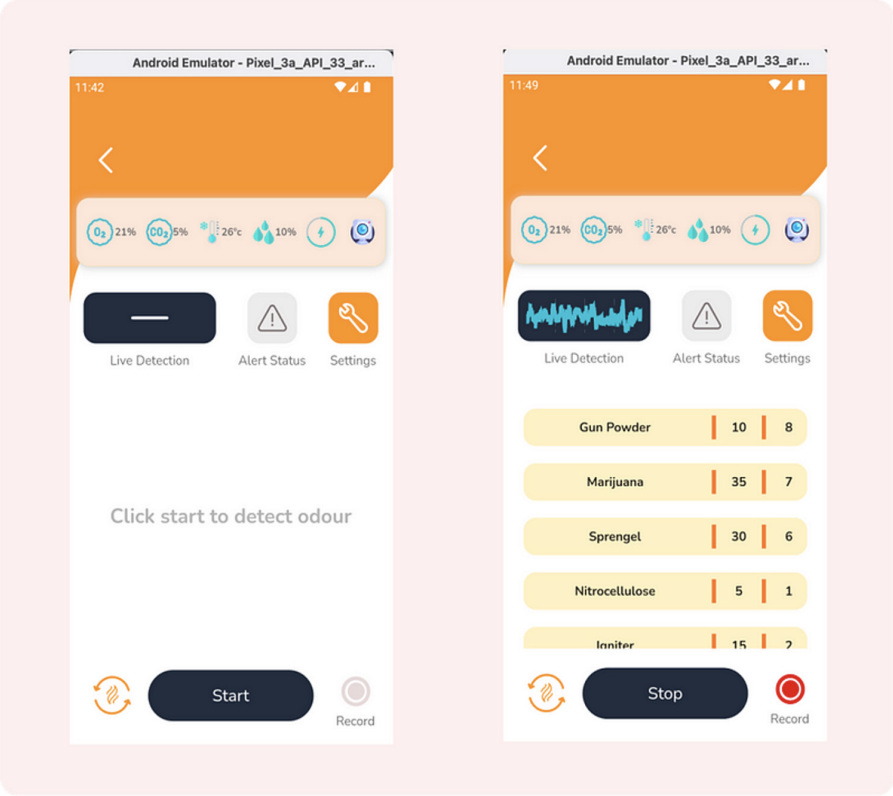
Within this specific screen, a wavy background is featured. Notably, this isn't an image, but a design meticulously crafted on a canvas using code. This exemplifies the exceptional versatility and precise control achievable with Flutter widgets. This level of customization ensures that the UI aligns seamlessly with the project's unique requirements and vision.

With the UI in place, we had two critical components at our disposal. Firstly, the data stream from the Olfactometer needed to be presented as a **sniff signal,** updating every 2 seconds. Secondly, we integrated **a live video feed from the camera** into the interface. These elements formed our interactive application's backbone, providing users with real-time insights and visual feedback.



5.2.4.1 Sniff Signal

The data stream from the Olfactometer was harnessed through Open Ephys, a robust open-source software. This data was multidimensional, encompassing channels ranging from 0 to approximately 5K, each with a capacity of 1kHz. To manage this intricate data stream, we opted for Web Sockets (RFC 6455) as they provided a more fitting solution compared to the conventional HTTP (V2) (RFC 7540) protocol. This choice ensured efficient and seamless data flow handling, a pivotal aspect in delivering real-time updates for the sniff signal.



5.2.4.2 Video Broadcast

The approach to video broadcast was distinctly unique in our case. Instead of a conventional camera, we utilized a specialized scientific-grade camera called **Basler.** This camera boasts various features and settings that surpass those of standard cameras. Noteworthy attributes include high-speed data transfer, precise color reproduction, low-light sensitivity, and advanced image processing capabilities. Additionally, the camera can capture images and videos at exceptionally high resolutions, rendering it ideal for scientific research and industrial applications where accuracy and precision are paramount.

The **Basler** camera provided a high-quality video feed to achieve our project objectives. However, it's worth mentioning that the specific camera procured by our client did not support chunk data transmission, a requirement for protocols like **RTSP (RFC 2336)** or similar alternatives. Consequently, we leveraged **Web Sockets (RFC 6455)** again, enabling the transmission of images over the network at an impressive speed of approximately 90 frames per second. This adaptation ensured seamless integration and real-time video feed streaming within our application.

5.2.4.3 Rendering Real-time Data and Video Streams: Integration and Implementation

In Flutter, there exist multiple methodologies for presenting data on a screen, including the use of setState or StreamBuilder. The choice is contingent on the specific use case and the application's requirements.

The setState method, while simpler, is well-suited for small-scale updates. However, upon invocation, it triggers a complete rebuild of the widget tree, which can be resource-intensive and potentially impact performance.

On the contrary, StreamBuilder proves more adept at handling real-time updates and accommodating large-scale data rendering. It actively listens to a data stream and selectively rebuilds only the affected portion of the widget tree. This approach alleviates strain on system resources and enhances overall performance. For our project, we strategically implemented StreamBuilder to manage changes efficiently. For instance, the Graph and Video Streaming components are rendered separately from the overall screen. Model selection and sensor fine-tuning are seamlessly handled through standard REST APIs, ensuring an optimized and responsive user experience.

5.2.5 Challenges

Throughout the development journey, we encountered several hurdles, primarily centered around the task of streaming data from a socket connection to multiple screens. This arose from the fact that the stream broadcast feature, as outlined in the Flutter documentation, didn't function as anticipated. To surmount this obstacle, I turned to leveraging **BehaviorSubject** from **RX_dart**. Those familiar with the RX library understand that it can be intricate and entail verbose syntax. Fortunately, my prior experience with Angular applications facilitated a smoother transition. Additionally, given that in my design the full-screen operated as a new widget being pushed onto the call stack, I opted to implement the service as a **Singleton**, ensuring seamless data flow.

Furthermore, we encountered some minor challenges regarding the integration of custom **SVG icons** into the project. Specifically, I discovered that there was no native way to directly incorporate **SVG icons** (relying on a package felt less than ideal). Instead, I had to convert them into a font format and subsequently import them into the project. While not a major setback, it did introduce some inconvenience and added supplementary steps to the development process. This experience serves as a valuable reminder of the significance of attending to even the minutest details, as they can significantly impact the overall developer experience.

5.2.6 Testing & Build Generation

An unexpected challenge surfaced during our rigorous testing phase: while video streaming appeared seamless in the emulator, the app's connection with the edge device would abruptly terminate. Identifying the root cause proved elusive initially. However, further investigation uncovered a bottleneck in the underlying network connection (IEEE 802.11g), resulting in sporadic terminations. To surmount this issue, we strategically shifted to a different band (IEEE 802.11ac @48Mhz/80Mhz), which led to a markedly smoother operation and facilitated uninterrupted development.

Our primary focus during development was on Android, with future plans to expand to iOS. For Android, we leveraged a feature known as "split APKs," enabling the creation of separate APKs tailored to different CPU architectures (e.g., ARM or x86). This optimization significantly reduced the overall application size by excluding unused binaries. However, in the case of Flutter, the APK size is also influenced by the number of packages incorporated. To maintain a compact app size (approximately 11MB), we predominantly crafted our own classes and integrated only essential packages or those too time-intensive to develop from scratch. This strategic approach ensured that the app remained lean without compromising functionality or performance.

5.3 Expanding Possibilities: Tailoring Solutions for Diverse Projects

By employing a similar strategic approach, we have the capability to adeptly meet the requirements of related projects that exhibit similar functionality or entail modifications that build upon the existing features. This method enables us to efficiently cater to the needs of analogous endeavors, all while leveraging the solid foundation already in place.

Embracing this approach not only streamlines the development process but also optimizes resource utilization. It capitalizes on existing solutions, fostering efficiency, and promoting uniformity across projects that share common functionalities or seek incremental enhancements. This systematic approach ensures that each project benefits from a wealth of accumulated expertise and established best practices.

5.3.1 Crop Monitoring System Leveraging a robust and versatile framework, we have the capability to design visually engaging and interactive user interfaces precisely tailored for crop monitoring systems. The application interface will seamlessly present real-time sensor data in an intuitive and user-friendly manner. This cross-platform solution empowers clients to conveniently access crucial insights, enabling data-driven decisions to optimize irrigation schedules, monitor crop health, and ultimately maximize overall yield.

5.3.2 Cattle Monitoring System The system benefits from a dynamic and responsive user interface, offering real-time tracking of cattle health, location updates, and behavioral analysis. This comprehensive solution equips clients with effortless access to vital livestock information across various devices, streamlining herd management and facilitating well-informed decision-making.

5.3.3 Energy Consumption Monitoring Dashboard The energy consumption monitoring dashboard boasts a visually appealing and user-friendly interface, purpose-built to meticulously track and analyze real-time energy usage data. This interactive platform empowers users to establish efficiency goals, receive tailored recommendations, and actively foster a more sustainable future. With its cross-platform capabilities, the dashboard ensures accessibility across various devices, delivering a seamless and engaging user experience.

6. Quantifiable Benefits

Efficient Development: Flutter's cross-platform capability allows developers to work from a single codebase, significantly reducing the time and effort required for development compared to maintaining separate codebases for different platforms.

Seamless Adaptation: Flutter's extensive UI component library offers pre-designed and customizable widgets. This enables interfaces to adapt to various headless systems seamlessly, ensuring functionality and user experience remain intact.

High Performance: Powered by Flutter's robust rendering engine, Skia (now Impeller), developers gain precise control over each pixel on the screen. This translates to swift and responsive interfaces, even on headless systems with limited computational resources, resulting in an elevated user experience.

7. Conclusion

This white paper has delved into the extensive capabilities of Flutter, a versatile cross-platform framework, for the development of applications in headless systems. We provided an overview of its introduction, rendering engine, and highlighted its popularity. Additionally, we showcased practical applications such as Crop Monitoring, Cattle Monitoring, and Energy Consumption Dashboard.

The successful implementation of the Olfactory Detection App using Flutter serves as a testament to the effectiveness of our systematic approach. However, Flutter harbors immense potential beyond these specific cases across diverse industries and environments. For instance, our app can be instrumental in identifying various odors in critical locations like hospitals and airports. Using Flutter, this application can enhance the overall experience and ensure a safer and more secure environment for everyone within such facilities.

With its visually appealing user interfaces, cross-platform adaptability, and continual advancement, Flutter stands at the forefront of innovation in headless system applications. We trust that this white paper imparts valuable insights and serves as a guiding light for maximizing the potential of Flutter within the realm of headless systems.

8. References

1. Flutter Architectural Overview. Retrieved from: https://docs.flutter.dev/resources/architectural-overview.

2. Stack Overflow Developer Survey 2023 - Most Popular Technologies. Retrieved rom: https://survey.stackoverflow.co/2023/#technology-most-popular-technologies.

3. Stack Overflow Developer Survey 2023 - Admired and Desired Technologies. Retrieved from: https://survey.stackoverflow.co/2023/#technology-admired-and-desired.

4. Flutter Architecture https://www.javatpoint.com/flutter-architecture#:~:text=The%20topmost%20layer%20is%20the,everything%20in%20the%20Flutter%20app

5. Build apps for any screen: https://flutter.dev/

6. What is Flutter App Development and How Can It Benefit Your Business?

7. https://www.thedroidsonroids.com/blog/what-is-flutter-app-development

USA
Cupertino | Princeton
Toll-free: +1-888-207-5969

INDIA
Chennai | Bengaluru | Mumbai | Hyderabad
Toll-free: 1800-123-1191

UK
London
Ph: +44 1420 300014

SINGAPORE
Singapore
Ph: +65 6812 7888

www.indiumsoftware.com

For Sales Inquiries
sales@indiumsoftware.com

For General Inquiries
info@indiumsoftware.com