somco software

Qt vs Android

Ultimate comparison of Qt & Linux
vs Kotlin & Android stacks

Qt

vs

somco software

Problem statement

Software leaders across industries face a growing challenge: **which technology stack delivers the best balance of performance, scalability, and efficiency** for modern embedded systems?

To provide a clear, data-driven answer, we prepared **a direct comparison between Qt on Linux/Android and Kotlin on Android**. Although the focus is on frameworks, applications rely heavily on their operating environments.

Therefore, we implemented the **same demo application** using both stacks to compare not only coding methodologies but also runtime performance.

This study aims to offer a **practical, unbiased perspective** on how Qt and Kotlin perform under real-world constraints, helping development teams make informed, future-ready technology decisions.

Lukas Kosiński
CEO & Founder
Somco Software (prev. Scythe Studio)

somco software

Demo Application Overview 01

To ensure a fair and practical comparison, **Somco Software** (previously known as Scythe Studio) designed and implemented **the same demo application** using both technology stacks - **Qt on Linux and Kotlin on Android.**

Our team selected a **resource-constrained board** intentionally, reflecting the real-world limitations of embedded systems. We began by designing the **UX/UI in Figma**, then developed two fully functional implementations, each built natively for its respective stack.

Although this benchmark is **sector-agnostic**, we chose an automotive-themed concept, reflecting one of the industries most affected by the Qt vs. Android debate. The demo represents a **digital dashboard for an electric vehicle**, named **Perun** - after the Slavic god of lightning.

The user interface is **visually rich and animation-heavy**, featuring **custom-drawn elements and fluid transitions** to test graphics performance and responsiveness. While this PDF cannot showcase the animations, a demonstration video is available on our **Somco Software YouTube channel.**

Hardware	Qt + Linux stack	Android + Kotlin stack
Toradex Verdin iMX8M Plus, 4GB RAM NXP® i.MX 8M Plus applications processor Quad Cortex-A53 @ 1.8 GHz + Cortex-M7 @ 800 MHz HDMI output (1920×720 @ 60 Hz)	Custom Yocto image (based on Kirkstone) Kernel: 5.15.177 Display stack: Wayland Qt version: 6.8.2	Android 14 (Upside Down Cake) Kernel: 6.1.57 API level: 34

Comparison authors

Somco Software is Embedded Software House with a great expertise in design, GUI, Qt framework and embedded programming including Linux customization. We are proud to hold the title of official Qt Service Partner.

Partners

Qt SERVICE PARTNER

Toradex Swiss. Embedded. Computing.

SoMLabs

Riverdi

Certifications

Clutch

Clutch

Clutch

Clutch

CERTIFIED
ISO 13485:2016
Quality Management for Medical Devices

CERTIFIED
ISO 9001:2015
Quality Management System

Benchmarking Methodology 02

To provide a fair comparison and for the sake of transparency, we explain here what metrics were measured and how we did it. All measurements were taken directly on the target hardware, once running the embedded Linux system and once running the Android system.

Embedded Linux (Qt)

The process began by collecting system information such as **CPU load**, **memory usage**, **storage performance**, **GPU activity**, **power draw**, and **thermal data**. These values were recorded over several minutes while the system was idle to establish a performance baseline. **Boot time** was measured by tracking the duration from power-on to the fully loaded graphical environment.

Afterward, the **test application** was launched in a controlled environment, and its resource usage was sampled at short, regular intervals. Each sample included timestamps, CPU and GPU utilization, memory footprint, and thermal state. All results were saved to structured data files for subsequent analysis.

Android (Kotlin)

For the Android system, the same methodology was applied. System-level metrics were gathered through standard system interfaces and performance counters. Application memory consumption was extracted directly from the **Android diagnostic output**, while **thermal data** came from the onboard temperature sensors.

Boot performance was measured in the same way as for the Qt app.

Frame Rendering

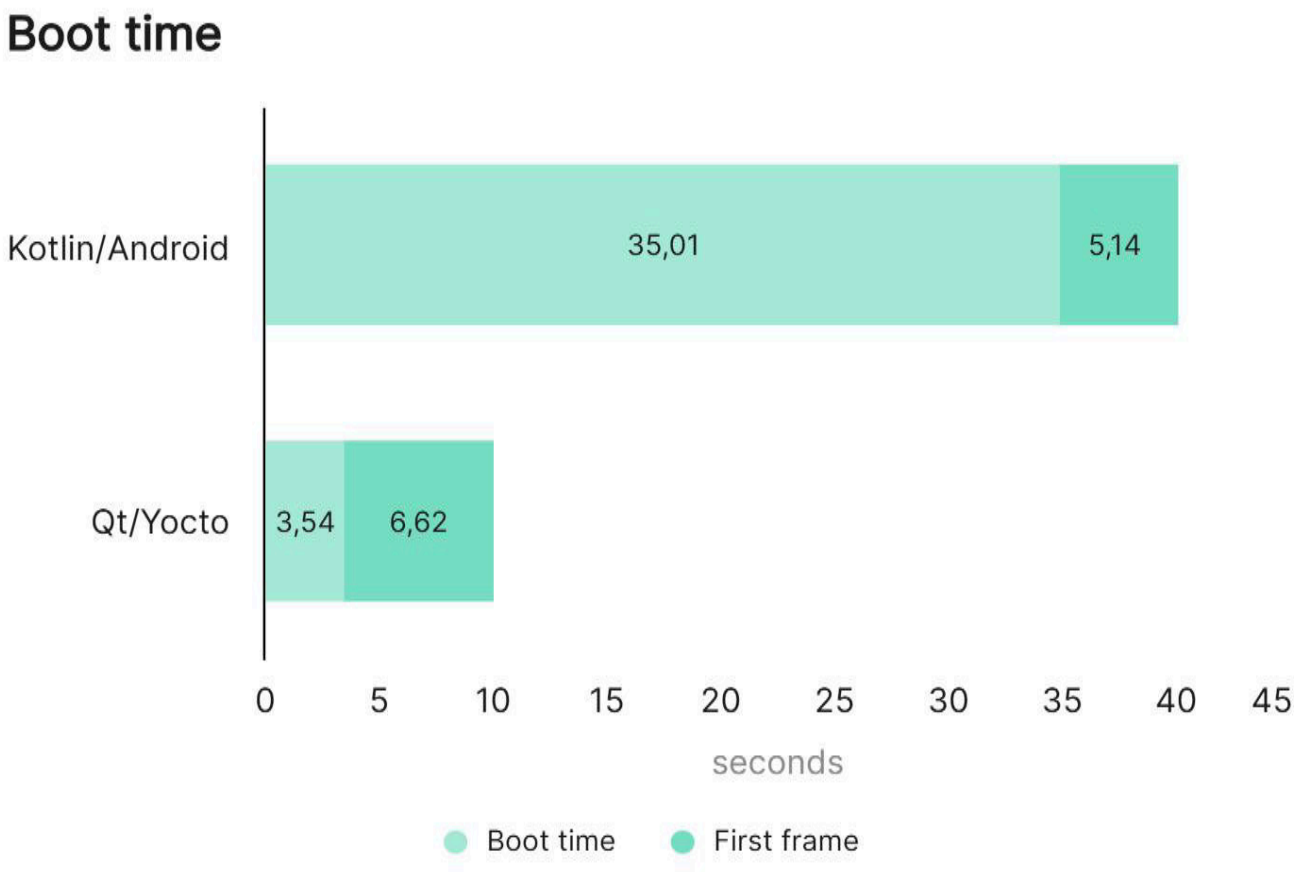
In both environments, **frame rendering performance** was measured within the application working. Metrics included **frame per second rate**.

Each version was left running for several minutes while frame statistics were collected, allowing a direct comparison between the two stacks.

Boot Performance 2.1

Boot time measurements revealed a clear difference between the two environments. The **Yocto-based Linux** system reached the point of displaying the instrument cluster in **just over 10 seconds**, whereas the **Android system** required **around 40 seconds** to reach the same operational state.

In automotive applications, this difference can be meaningful. Instrument clusters are expected to become available almost immediately after the vehicle is powered on, and in many regions, **regulatory guidelines emphasize the prompt display of driver information**. Android's startup process involves initializing a broad range of system components and background services, which contributes to longer boot times. By contrast, **Yocto-based systems can be customized to the specific hardware**, omitting unnecessary services and achieving a more streamlined boot sequence. It also benefits cybersecurity which is a hot topic.

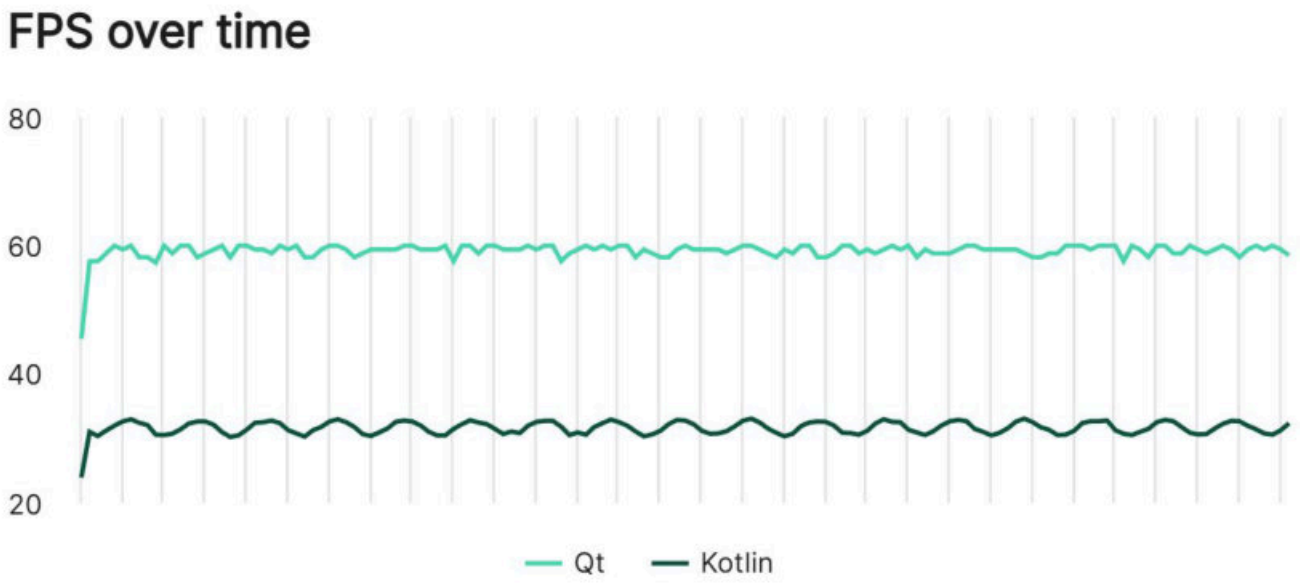


Rendering and responsiveness 2.2

When the systems were running, differences in responsiveness were clear. The Qt/Yocto cluster consistently delivered around **59 frames per second**. This performance was steady and predictable, which is often more valuable than raw frame rates in automotive displays. Smooth gauge sweeps and stable transitions were achieved without major timing fluctuations.

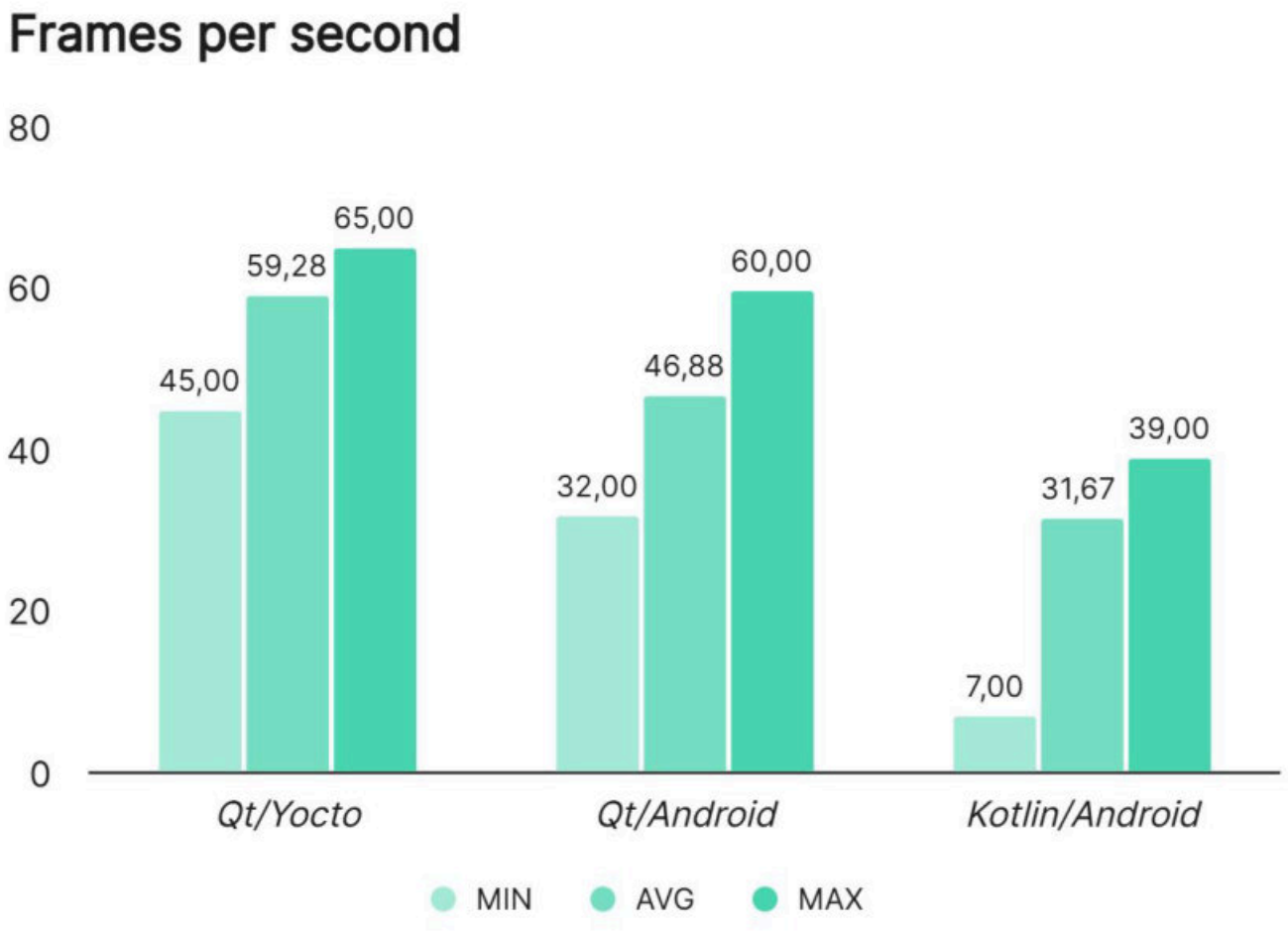
Android, implemented through Jetpack Compose, reached a lower level of responsiveness on the same hardware. In these tests, it **averaged closer to 32 frames per second**. The animations appeared visibly less fluid, and the cluster's visual experience felt less immediate. Our first attempt without any optimizations that require high expertise, it offered only 7-9 FPS.

The graph shows how the FPS rate changed over time for both applications.



For the record, as **Qt framework is cross-platform** we installed it also on the target with Android and results were pretty much the same as on Linux (a bit better on Linux) therefore it's not a system influencing rendering but the framework itself.

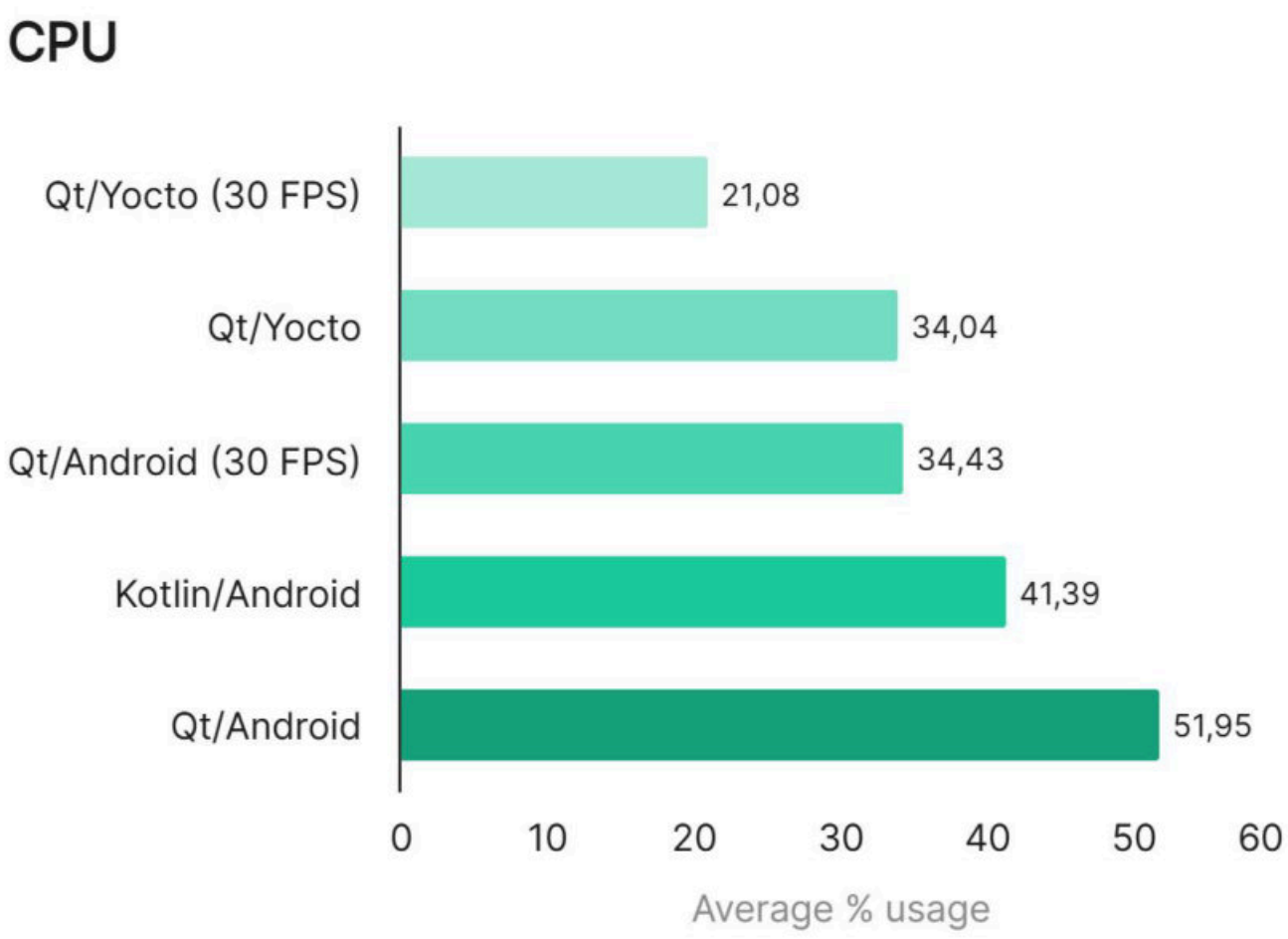
The chart below shows the **minimum**, **average**, and **maximum FPS** achieved by the Qt application on Yocto, the Qt application on Android, and the Kotlin application on Android.



In order to ensure full transparency of the results during testing, we decided to prepare two additional versions of the application in Qt – both for Yocto and Android, where FPS was deliberately limited to 30 FPS to show how Qt uses resources with similar rendering results. Of course, we did not include these statistics for the 30 FPS capped version in the chart above, as the minimum, average, and maximum FPS rates would be 30.

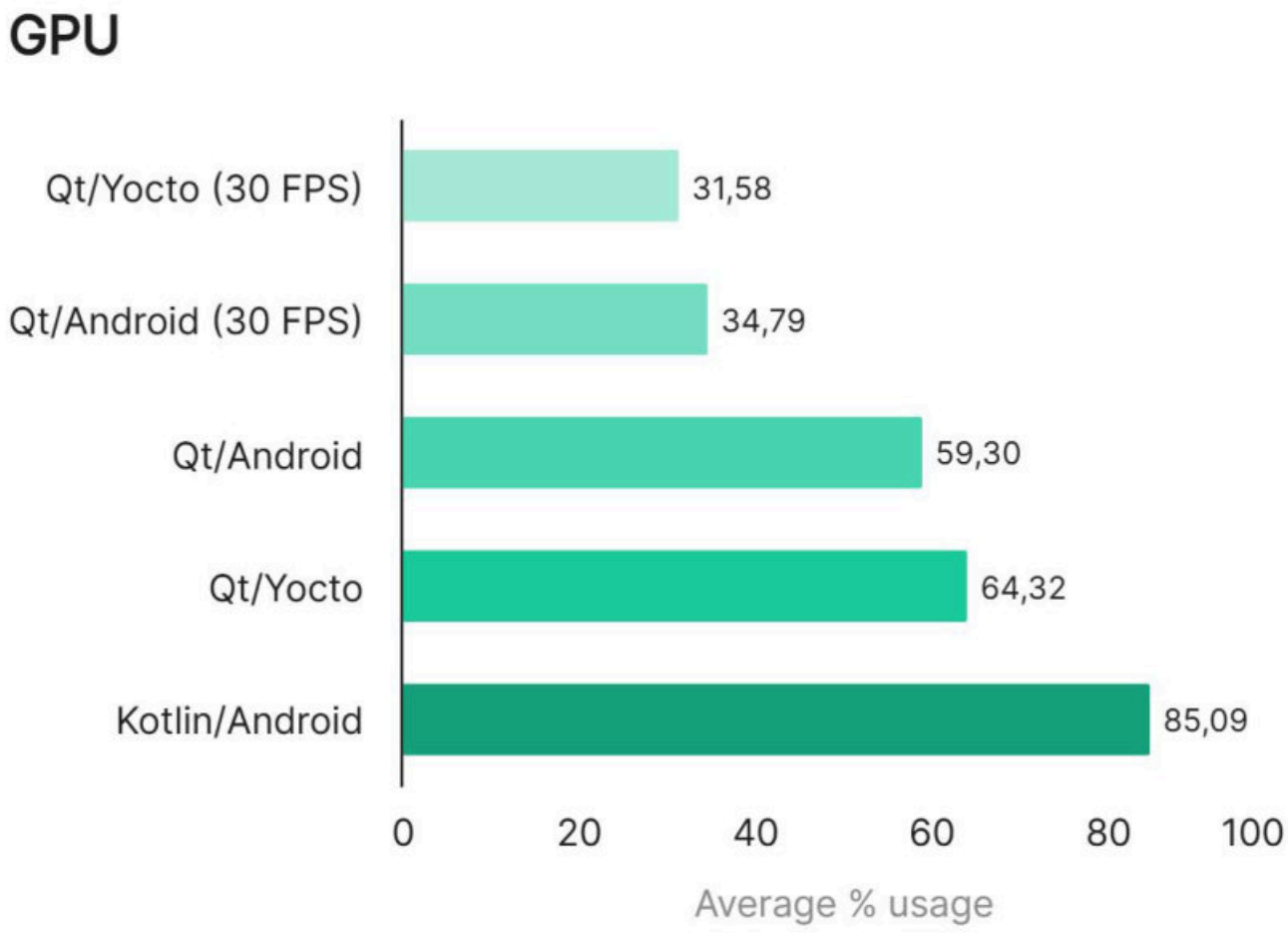
System Resource Utilization 2.3

Beyond what the driver sees, the two stacks placed very different demands on the hardware. Under load, the Qt/Yocto application kept average CPU busy time at around **34 percent**, while Kotlin/Android required **41 percent**. This difference matters because every percentage point of CPU usage in a cluster system is a portion of resources unavailable for other tasks, including safety-critical computations.



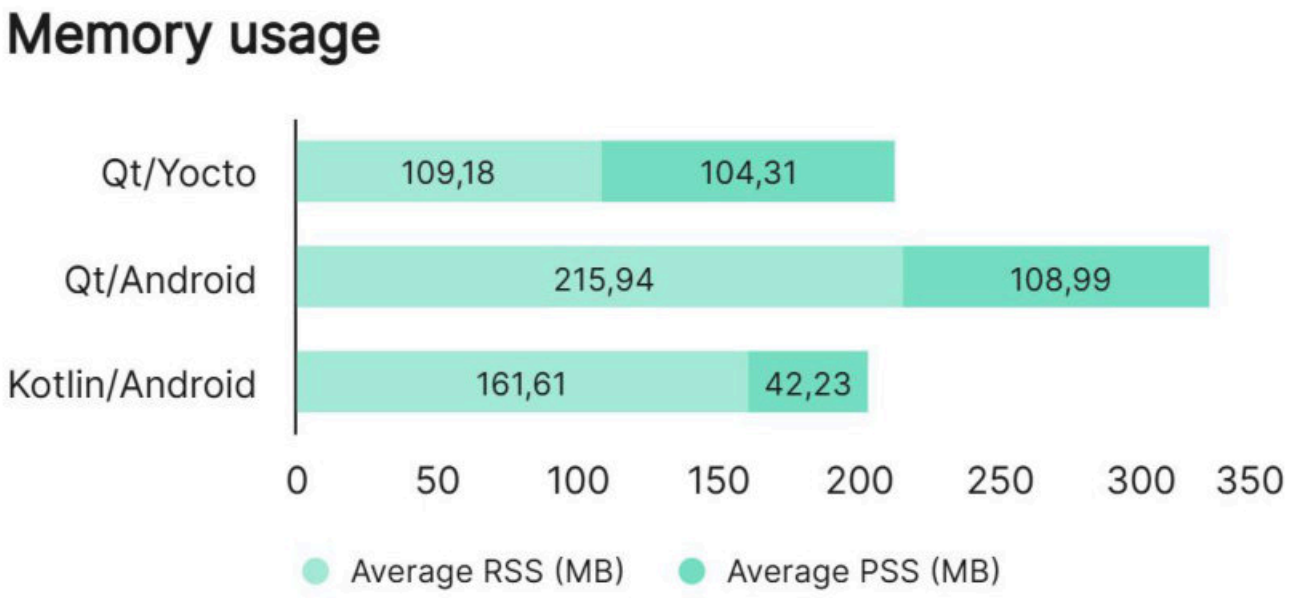
The most notable differences are observed in GPU utilization. As demonstrated in Section 2.2, Rendering and Responsiveness, the Qt/Yocto implementation achieved an average of 59 FPS, whereas the Kotlin/Android implementation reached only 32 FPS.

Despite delivering nearly twice the frame rate, Qt/Yocto exhibited lower GPU utilization by approximately 21 percentage points.



Furthermore, when the Qt application was configured with a 30 FPS limit - bringing its frame rate closer to that of the Kotlin/Android implementation, the disparity in GPU usage became even more pronounced.

Memory consumption also diverged. Qt/Yocto required an average resident memory of just over **109 megabytes**, while Android consumed **161 megabytes**. For a consumer smartphone this would be trivial, but for an embedded system deployed across a fleet of vehicles, every megabyte affects hardware selection, costs, and long-term maintainability.



Today I learned:

RSS (Resident Set Size) represents the total physical memory mapped by a process, counting shared libraries in full, whereas **PSS** (Proportional Set Size) adjusts this by dividing shared memory among all processes that use it, providing a more accurate measure of the process's actual memory cost.

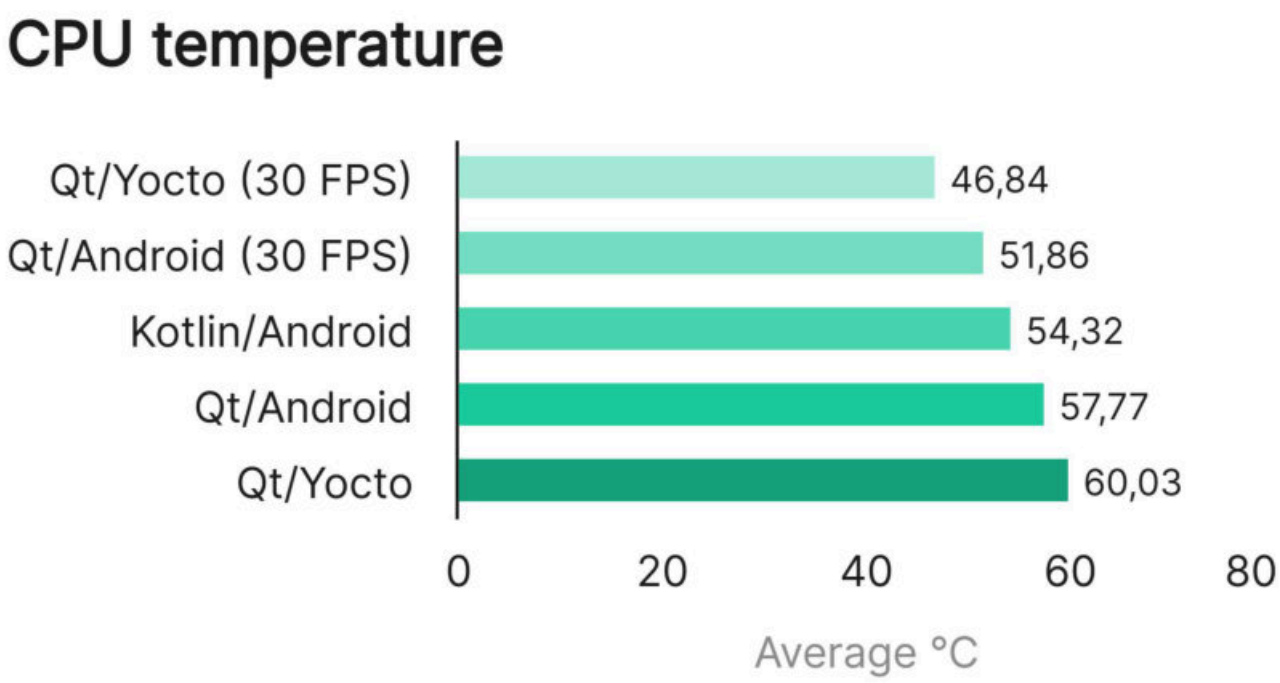
Qt/Android shows a high RSS because it maps a large number of Android system and Qt shared libraries, which are heavily shared with other processes and therefore inflate RSS without significantly increasing PSS.

PSS matters a lot too. Kotlin/Android has low PSS probably, because Android runs a lot of other processes in the background while on Yocto it's barely nothing.

Thermal Characteristics 2.4

Thermal readings showed that the Qt/Yocto application generated slightly higher peak temperatures when there was no FPS cap, with the CPU reaching around 60 degrees Celsius, compared with 54 degrees under Kotlin/Android. Although, Qt deployed on both Linux and Android managed to get better temperatures once we intentionally limited FPS to match maximal rendering rate of Kotlin.

For a system designer, this raises important architectural considerations. Efficient thermal management must balance heat generation, cooling design, and the battery or power budget of the vehicle. Both stacks operated within safe margins, but their different thermal and power behaviors could influence hardware choices and integration strategies.



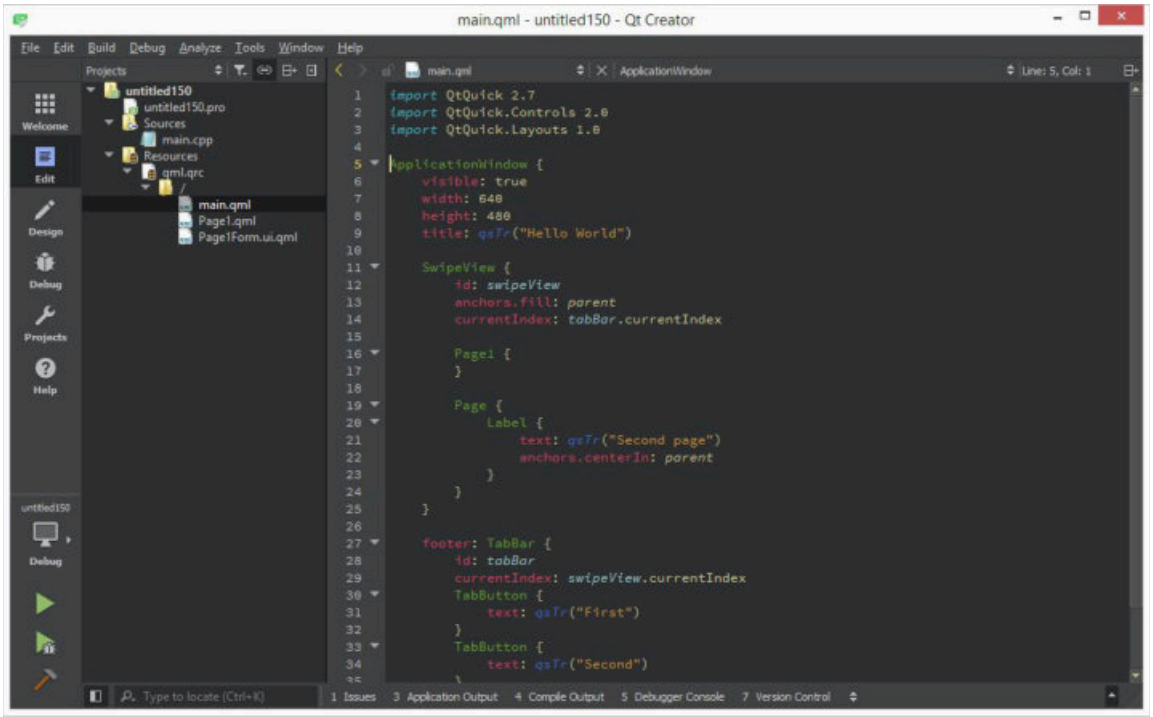
Developer's workflow and experience 03

Qt + Linux

The technical differences extended into the development process itself. Building with Qt on Yocto required establishing a full cross-compilation environment, creating a Yocto image, and carefully configuring the SDK. Although, at Somco configuring custom Linux images is our bread and butter, this process is more complex and demands specialized expertise, but it also grants precise control over what runs on the device. And that as mentioned really matters in the times of SBoMs and all cybersecurity regulations.

Anyway it is important to mention that Qt has ready Linux images for chosen recommended targets available in the Commercial License. Also some hardware vendors like Toradex come up with solutions that make your life a lot easier.

The **separation of declarative QML for the interface and C++ for the logic enforces a structure that feels natural in embedded projects**, where deterministic behavior and resource predictability are central. Implementing visual effects and state machines also proved more straightforward in QML, as the declarative model lends itself well to describing transitions and system states, whereas achieving the same in Kotlin required more manual effort.



Kotlin + Android

Working with Android and Jetpack Compose was almost the opposite. The project could be started in minutes from within Android Studio, with previews, profilers, and debugging tools all available immediately.

This environment accelerates prototyping and lowers the entry barrier for developers already familiar with Android. However, when it came to implementing more advanced visual effects, Jetpack Compose presented several practical challenges. While Compose provides its own set of built-in effects such as blur, shadow, and various graphical layers, these proved less reliable under continuous animation on the tested i.MX hardware. Certain effects exhibited visible glitches or stuttering, particularly when parameters were animated at runtime, which appeared to be linked to recomposition and offscreen rendering overhead within the Compose graphics pipeline.

To address these limitations, custom shaders were developed. Although this approach offered more flexibility and allowed for finer control over visuals, it also introduced inconsistencies across Android API levels and GPU drivers. The same shader did not always render identically on different Android versions. Additionally, shader precision, color space handling, and internal caching behavior could vary subtly, resulting in non-deterministic visual output between devices or builds. In contrast, the Qt rendering path proved far more predictable: once an effect was implemented in QML or GLSL and verified on the target, it behaved consistently across runs, with stable frame timing and visual uniformity.

The separation of declarative QML for the interface and C++ for the logic enforces a structure that feels natural in embedded projects, where deterministic behavior and resource predictability are central. Implementing visual effects and state machines also proved more straightforward in QML, as the declarative model lends itself well to describing transitions and system states, whereas achieving the same in Kotlin required more manual effort.

Comparison of implementation of the same glow effect

04

Although it might be a detail, Qt offers sharper and more attractive visual effects.

Qt

Kotlin

Although the **Perun UI demo** represents the dashboard of a conceptual electric vehicle, this comparison of **Qt and Android technology stacks** remains largely **sector-agnostic**. Both the developer experience and the measured performance indicators influence technology decisions across a wide range of industries. However, certain domain-specific factors can make one stack more suitable than the other.

Sector-specific considerations 05

One of the aspects that should influence your choice is the specific environment of your industry. This comparison isn't just for one use case. It has strong implications across **several industries**.

Automotive

The growing popularity of Android in the automotive industry is largely driven by the emergence of **Android Automotive OS** - a variant of Android specifically designed for in-vehicle infotainment (IVI) and instrument cluster systems. It provides a ready-to-use ecosystem with access to **Google services, voice assistant integration, OTA updates, and a familiar app model**, which accelerates development and leverages existing mobile expertise.

However, this convenience comes at the cost of **reduced control over system components and longer boot times**, which may limit its applicability for real-time or safety-critical features. Qt, by contrast, allows full customization of the software stack and has long been used for **mission-critical HMI applications** where deterministic performance is essential.

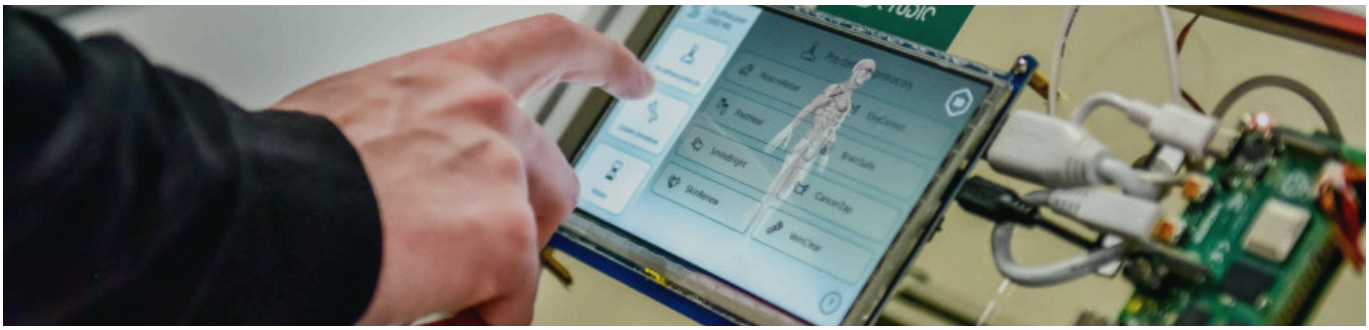
Medical

Many medical systems operate in **kiosk mode**, where a single application runs full-screen, and user access to the rest of the system is restricted for **security, reliability, and regulatory compliance**.

Somco Software encountered such a case, where an existing Android-based medical application had to be **migrated to another operating system**.

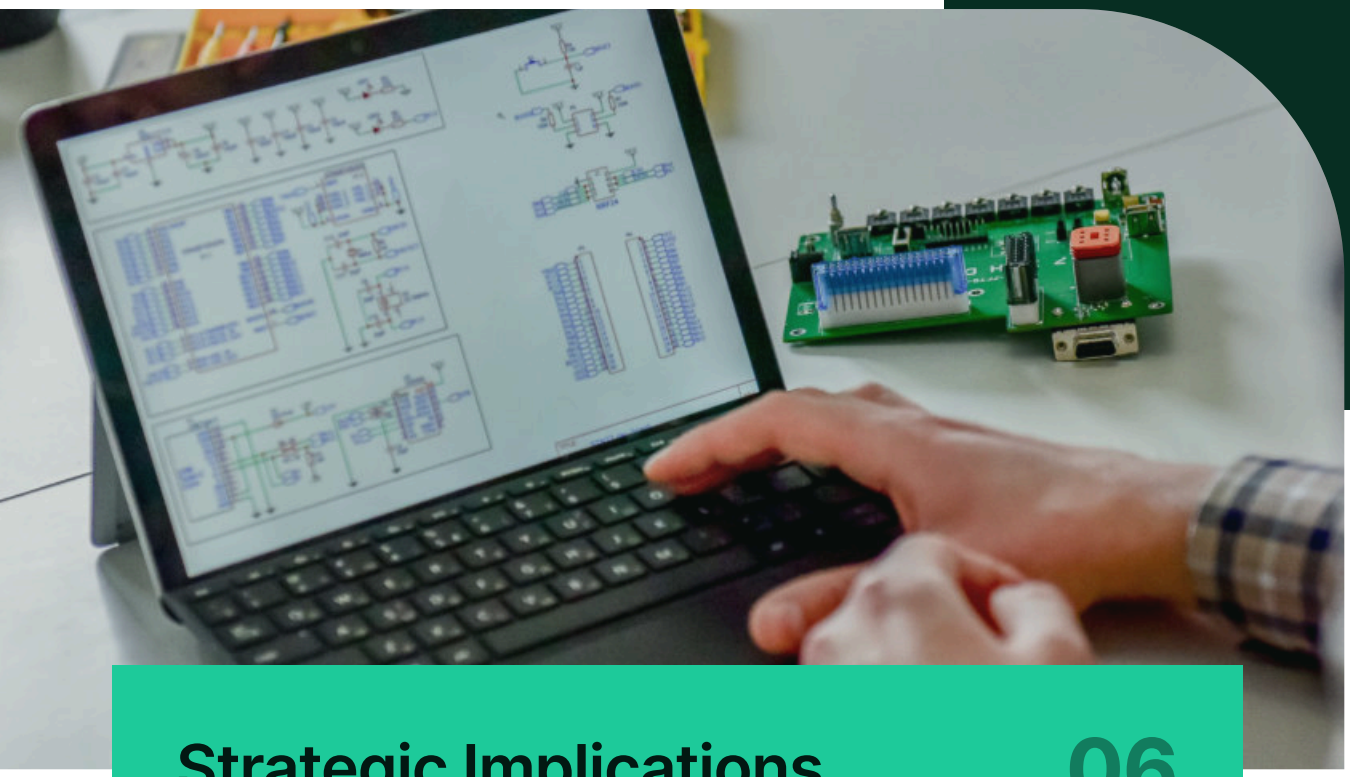
Thanks to Qt's **cross-platform architecture**, this transition was achievable without a complete rewrite - something that would have been significantly more complex using native Java or Kotlin frameworks.

Another key concern in the medical field is **software provenance** and **regulatory documentation**, such as **SBOM (Software Bill of Materials)** and **OTS (Off-the-Shelf Software)** assessments. Qt's monolithic ecosystem - with built-in modules for **3D rendering, networking, database access, and visualization** - minimizes reliance on third-party libraries. This significantly reduces the **SOUP (Software of Unknown Provenance)** assessment scope and overall documentation effort, compared to Android, where developers depend on multiple external components and libraries.



Electronics & IoT

In the broader **Electronics and IoT** domain, the choice of technology stack depends heavily on the **target hardware**. This benchmark showed that Android faced performance challenges even on the **NXP i.MX8 Plus**, which is already considered a high-end embedded processor by industry standards. Qt, on the other hand, scales far beyond that range — supporting **MPUs, MCUs, and even bare-metal environments**. This flexibility makes Qt a practical choice for devices where **footprint, startup time, and real-time responsiveness** are critical.



Strategic Implications 06

The evaluation ultimately highlights two different approaches to the same challenge. Qt/Yocto represents a philosophy of lean, hardware-close design, in which startup speed, resource efficiency, and predictability dominate. It is well-suited to environments where determinism and minimal footprint are key, such as safety-critical displays and long-term embedded deployments. However, Qt often involves commercial licensing fees, which can be significant depending on deployment scale. Organizations must weigh these recurring costs against the benefits of its lightweight footprint and deterministic behavior.

One of the key advantages of **Qt** is its ability to make projects **more flexible and maintainable over time**. Thanks to its **cross-platform architecture**, applications can be adapted to new operating systems or hardware targets with minimal rework, ensuring long-term scalability and investment protection.

Android/Compose, in contrast, represents a philosophy of ecosystem leverage. It excels in productivity, rapid iteration, and integration with the Android Automotive ecosystem. Its trade-off is a heavier runtime and greater demand on hardware resources. On the cost side, Android itself is open source and free of licensing fees, but teams may incur indirect expenses in the form of higher hardware requirements and the need for continuous updates aligned with the Android ecosystem. For teams already aligned with Android as a strategic platform, this may be acceptable or even advantageous, particularly if time-to-market, developer availability, and ecosystem services are more valuable than minimizing license costs or squeezing every cycle from the SoC.

Summary

Category	Qt / Yocto Linux	Kotlin Jetpack Compose / Android
Boot time	~10 seconds	~40 seconds
Frame rate	~59 FPS on average	~31.50 FPS on average
CPU usage	~52%	~41%
GPU usage	~64%	~85%
Memory	~109 MB (RSS)	~161 MB (RSS)
Thermals	~60 °C	~54 °C
Dev setup	Moderate (cross-compilation)	Moderate (Android Studio, built-in)
Hardware compatibility	Wide range of devices supported	Limited to chosen vendors

The two implementations, though built for the same task and on the same hardware, paint very different pictures. Qt on Yocto delivered faster startup, lower CPU utilization, smaller memory footprint, and more consistent rendering. Android with Jetpack Compose offered an easier and faster development experience, but required more resources and produced less fluid runtime behavior on the tested platform.

	Yocto + Qt	Android + Kotlin
Pros	Fast boot	Rich development tools (Android Studio)
	Low GPU usage	Large developer community
	High, stable and consistent FPS	No licensing fees
	Full system control	Easy integration with Android Automotive
	Deterministic behavior	
	Minimal OS overhead	
	Java integration possibility for native features	
Cons	Complex build setup	Long boot time
	Potential Qt licensing costs	Lower frame rate
	More manual integration and debugging effort	Higher GPU and memory usage
		Less system control
		Heavier OS footprint and ecosystem dependency

Verdict

Neither approach can be declared “better” in all cases. **The decision comes down to priorities.**

If the product strategy demands maximum efficiency, rapid boot, and predictable frame timing, **Qt/Yocto provides a natural fit.**

If the strategy instead emphasizes platform maturity, ecosystem alignment, and out-of-the-box readiness with Android Automotive, Jetpack Compose offers clear advantages - while acknowledging that baseline features such as navigation, app store access, and voice assistants come with additional integration and performance costs rather than being “free” parts of AAOS.

somco software

somcosoftware.com