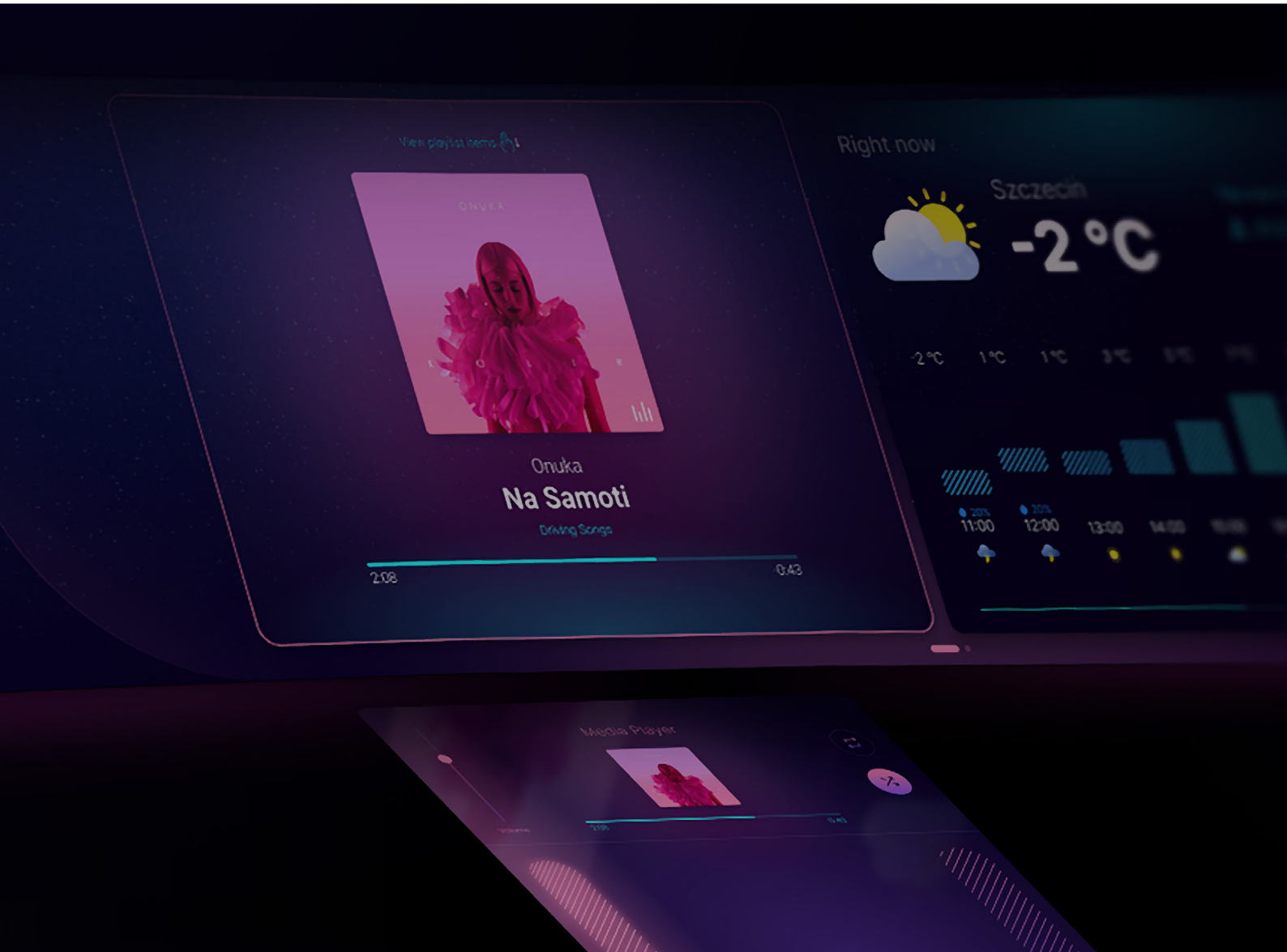
Qt for MCUs vs LVGL

in the embedded GUI market

Spyrosoft S.A.
www.spyro-soft.com

Mikalai Arapau
mikalai.arapau@gmail.com

May 2026 update



Executive Summary

As an HMI leader in the embedded industry, we looked at Qt for MCUs—a commercial, full-stack GUI framework from The Qt Company that brings the familiar Qt Quick (QML) and C++ paradigm to microcontrollers—and LVGL (Light and Versatile Graphics Library)—a widely adopted open-source C graphics library focused on lightweight footprint, portability, and flexibility.

Qt for MCUs leverages an optimized graphics runtime (Qt Quick Ultralite) to deliver smartphone-like user interfaces on embedded devices, with integrated tools for design, development, and testing. LVGL, in contrast, is an open-source C library centered on widgets and 2D drawing that can run on very resource-constrained MCUs (even with under 100 MHz CPU, ~50 KB of RAM). Unlike Qt for MCUs' comprehensive ecosystem, LVGL is purely a GUI library—developers integrate it with their own platform code and other libraries for non-UI functionality.

> Qt for MCUs vs LVGL: both are technically strong embedded GUI options. In our study, Qt for MCUs cut development time by ~30% thanks to its tools, while LVGL wins on runtime licensing flexibility.

We have conducted an independent study where both frameworks were used to implement the same washing machine GUI on two MCU platforms (NXP i.MX RT1064 and ST STM32U5). The results showed that Qt for MCUs reduced development time by roughly 30% compared to LVGL. This productivity gain stemmed largely from Qt's integrated end-to-end tooling (e.g. the Figma-to-Qt design import and Qt's unified Qt Creator IDE) which streamlined designer-developer collaboration.

At runtime, the results were highly configuration- and platform-dependent. On RT1064, Qt for MCUs delivered the strongest sustained frame rate and lower measured runtime RAM/flash footprint. On STM32U5, both Qt for MCUs and LVGL@60FPS achieved true 60 FPS-class smoothness, but Qt did so at much lower CPU load while using more heap and a larger footprint due to its app's FPS-oriented adjustments. The key differentiators therefore lie not only in raw runtime metrics, but also in development efficiency, workflow integration, tuning flexibility, and long-term capabilities.

Qt for MCUs offers faster time-to-market, an all-in-one toolchain, and optional certified components for safety-relevant domains such as automotive and medical devices, at the cost of commercial licensing. LVGL's core runtime library remains MIT-licensed and can be used without runtime license fees, while the newer professional LVGL Pro tooling introduces paid options for commercial teams.

Both have thriving user communities and successful products, so the choice depends on your project's particular requirements and constraints.

Contents

Study Methodology and Test Scenario 5
- Study Goal and Scope 5
- Test Application: Washing Machine HMI 5
- Toolchains and Workflows Compared 6
- Hardware Platforms and Configuration 6
- Measured Outputs 7
- Why This Methodology Matters 7

Design-to-Code Workflow Comparison 8
- Figma to Qt: Automated QML Generation 8
- Figma to LVGL: Styles Export and Manual Rebuilding 10
- Implications for Design Fidelity and Iteration 12

Development Environment and Deployment 12
- Qt for MCUs: Integrated IDE and Toolchain Support 12
- LVGL: Vendor IDE and Multi-Tool Workflow 13
- Vendor Tools and Low-Level Integration 14

Testing and Quality Assurance 15
- Qt for MCUs: Squish for MCUs – Automated UI Testing 15
- LVGL: Unity/Ceedling and Simulator-Based Testing 16
- Testing Philosophy: Black-Box vs White-Box 17

Performance Benchmarks 18
- Metrics collected 18
- RT1064 footprint 19
- RT1064 runtime performance 19
- Conclusions 22

Source Code Footprint 23

Development Productivity and Time-to-Completion 24
- Development Effort Breakdown 24
- Economic Implications 25
- When Does a 30% Gain Matter Most? 26

Strategic Ecosystem Advantages of Qt 27
- One Framework from MCUs to PCs 27
- Tooling and Productivity Ecosystem 28
- Long-Term Support and Stability 28

Frameworks Positioning: When to Choose One Over Another 30

Choose Qt for MCUs if... 30
Choose LVGL if... 31
Selection Criteria Summary 32

Architecture and Full-Stack Considerations 33
Qt for MCUs as a Platform Layer 33
LVGL as a GUI Library 33
Integration of Other Software Components 34
Pros and Cons – Architectural Trade-offs 34

Lifecycle Maintenance and Stability 35
Qt's Release and Support Model 35
LVGL's Community-Driven Evolution 35
Implications for a product team 36
Commercial Support vs Community Support 36

Security, Safety, and Robustness 36
Qt for MCUs – Security and Safety Posture 37
LVGL – Community-Driven Security and Safety 37
Comparative View for Safety/Security 38

Hybrid Local and Remote UIs 39
Common Patterns for Local + Remote UX 39
Qt-Centric Architectures 39
LVGL-Centric Architectures 40
Communication & Separation 40
Headless or multi-screen considerations 40

AI-Assisted Development and Agentic Workflows 41
Qt for MCUs 41
LVGL 41
What to expect 42

Conclusion 43

References 44

About Spyrosoft 45

About The Author 45

Study Methodology and Test Scenario

To provide an objective comparison, we performed a side-by-side implementation of an identical GUI using Qt for MCUs and LVGL. This section outlines how the study was conducted and why its findings are relevant for real-world embedded GUI development.

Study Goal and Scope

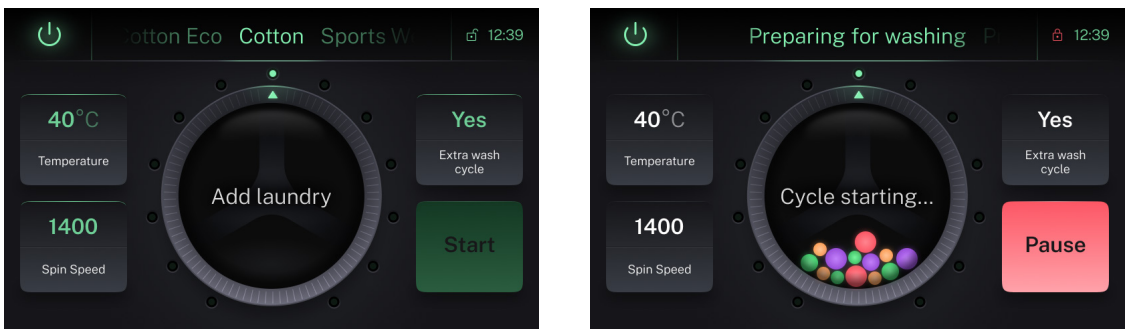
The goal was to compare end-to-end development of an embedded HMI using Qt for MCUs vs. using LVGL, focusing on both developer effort and runtime performance. Rather than isolated benchmarks, the study covered the full workflow from design to deployment – starting with a high-fidelity UI design, moving through code implementation, and ending with the application running on actual hardware. Key aspects measured included:

- **Development effort:** Hours spent in each phase (design import, UI building, hardware bring-up, performance tuning).
- **Runtime metrics:** Memory usage (RAM/flash), frame rate, and CPU utilization on the target MCUs.
- **Workflow characteristics:** The tools and steps required to go from design files to a running app, and the collaboration between designers, developers, and testers.

Crucially, the comparison used one realistic application rather than artificial micro-benchmarks. This ensured the results reflect typical challenges of developing a modern appliance GUI, instead of focusing on any single optimized metric.

Test Application: Washing Machine HMI

The chosen demo is a modern washing machine GUI featuring multiple screens, smooth animations, and touch-based navigation – representative of the “smart appliance” interfaces increasingly common in consumer and industrial devices. Our UX designer created the UI in Figma with smartphone-like visual quality (rich graphics, dynamic transitions) to serve as the common design input for both frameworks. By using the exact same Figma design for Qt for MCUs and LVGL, the study ensured a fair comparison: both implementations had to meet the same visual and interactive requirements without one being inherently simpler than the other. This also tested how well each framework could realize a pixel-perfect design created by a designer, and how much manual tweaking was needed to match the intended look and feel.



Washing machine UI

Toolchains and Workflows Compared

For each framework, the latest available tools were used to translate the Figma design into an embedded application:

- **Qt for MCUs workflow:** For this evaluation, our team used a beta version of Qt's Figma-to-Qt bridge tooling to transform the Figma design into Qt UI assets/code for subsequent development. The resulting project was then continued in Qt Creator (Qt's IDE), where developers added application logic and integrated MCU-specific services using C++ and Qt Quick Ultralite, while preserving the structure generated from the design. Qt's test automation tool Squish for MCUs was used for GUI testing on the target hardware. In summary, the Qt workflow was highly automated, beginning with a design-to-code step and continuing through implementation and testing within Qt's toolchain. At the time of publication, Qt had released Figma to Qt 1.0, the official successor to Qt Bridge for Figma.

- **LVGL workflow:** We tried a combination of the Figma to LVGL plugin and the new LVGL Editor (part of LVGL's tooling ecosystem) to implement the same design. The Figma to LVGL plugin exported style information – colors, fonts, and basic widget styles – from the Figma file. However, unlike Qt's plugin, it did not export full layout or interaction code. We had to manually rebuild each screen and widget layout. We used the LVGL Editor's XML-based UI composer to arrange components and screens and apply the exported styles, visually previewing the result. Once each screen was reconstructed in the editor, C code could be generated from the XML and integrated into a project. Development then proceeded in a traditional MCU IDE (for example, STM32CubeIDE or MCUXpresso) to write the application logic and handle hardware initialization. In summary, the LVGL workflow involved multiple tools – a style exporter, the LVGL Pro Editor for UI assembly, and a separate IDE for coding – with significant manual effort to translate the design into code.

Despite the introduction of LVGL's new editor (which improves productivity compared to purely hand-coding LVGL), the contrast was clear: Qt's approach provided an immediate running prototype from the design, whereas LVGL's approach required reimplementing the design by hand. This difference in automation level is a major theme in the results, affecting development time and team collaboration.

Hardware Platforms and Configuration

Two popular MCU development boards were selected as test platforms to cover a range of typical use cases:

- **NXP i.MX RT1064-EVK:** A crossover MCU board featuring a 600 MHz Arm Cortex-M7 processor, 4 MB on-chip flash, and an on-chip LCD controller with 2D graphics accelerator (PXP). This represents a high-performance MCU scenario, often used in industrial HMI panels or higher-end appliances where rich graphics are needed. The board was configured with an external display (LCD) and used double buffering (two framebuffers) to ensure smooth animations in both frameworks.

- **STMicroelectronics STM32U5G9J-DK2:** A development kit for the STM32U5 series, featuring an ultra-low-power 160 MHz Arm Cortex-M33 MCU with 3 MB SRAM and 4 MB embedded flash, plus advanced power-saving features. For graphics, it supports an LCD interface (LTDC) and multiple accelerators, including the NeoChrom (2.5D) accelerator /

NeoChromVG GPU, Chrom-ART (DMA2D), and Chrom-GRC™ (GFXMMU). This represents a more power/battery-sensitive device class (e.g., wearables or IoT) where efficiency is critical. Double buffering was enabled on this board.

Using these two boards allowed the study to observe each framework in different conditions – one with more performance headroom (i.MX RT) and one more constrained and low-power (STM32U5). Both frameworks were optimized for each board, making use of available graphics accelerators, tuning memory placement, etc.

Measured Outputs

The study tracked a set of concrete metrics on each platform:

- **Development effort:** Time spent in each development phase – from importing the design to building out the UI, bringing up on hardware, and final performance tuning. This revealed how much developer time each framework required to reach a fully working product.

- **Memory footprint:** The total RAM usage of the application at runtime, and the total flash memory (program storage) used by the compiled firmware, for each framework.

- **Performance:** The frame rate (frames per second) achieved during typical UI animations, both maximum and minimum observed, as well as the peak CPU utilization during heavy UI activity.

- **Workflow notes:** Qualitative observations on tool usage, debugging difficulty, and any friction points encountered were also recorded, to supplement the raw numbers with context.

By collecting both quantitative data and qualitative developer feedback, the study not only compared "speed and size" of Qt vs LVGL, but also captured the developer experience differences.

Why This Methodology Matters

This comparative approach mirrors real product development far more closely than isolated benchmarks:

- Both frameworks started from the same design and had to achieve the same functionality, which controls for differences in scope or ambition. It's a like-for-like comparison in terms of what was built.

- The process spanned the entire development lifecycle – design handoff, coding, testing, deployment – giving a holistic view. In commercial projects, time lost or saved in handoff or testing can be just as critical as raw performance.

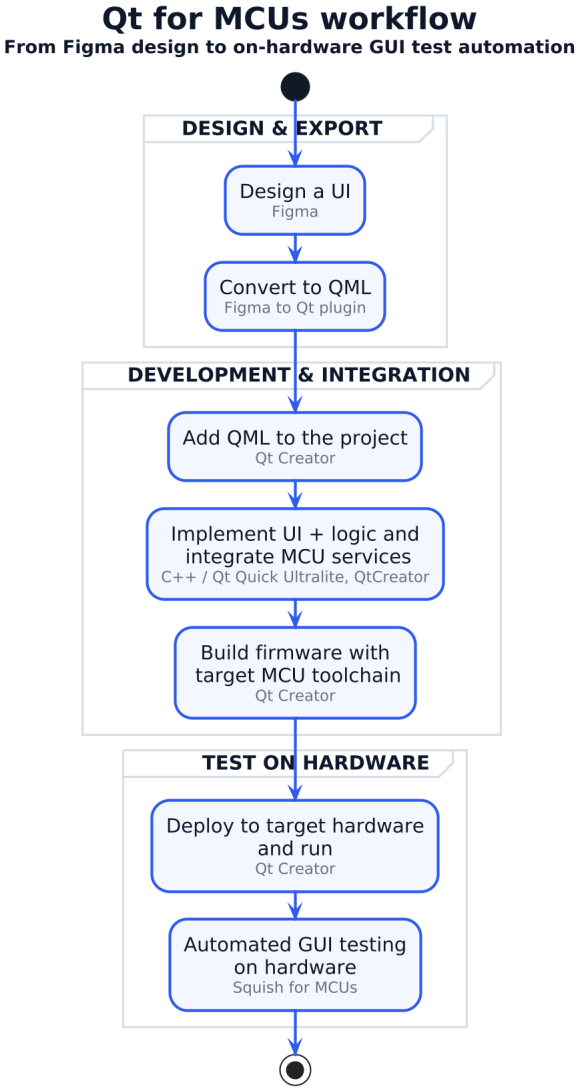
- Using physical hardware and measuring real performance (not just PC simulations) ensured that any pitfalls with drivers, memory constraints, or integration would surface. This is crucial because some frameworks might shine in a PC demo but struggle on device, or vice versa.

- The mix of quantitative metrics and qualitative insights helps decision-makers understand not just which framework is "faster" or "smaller," but why – e.g. does it demand more skilled developers? Does it come with better tools? This aligns the comparison with business considerations like team productivity, maintenance effort, and risk.

Overall, the methodology provides a balanced comparison that is relevant to choosing a GUI framework for real-world embedded projects. The following sections will discuss the results and insights from this study in detail.

Design-to-Code Workflow Comparison

One major area of differentiation between Qt for MCUs and LVGL is how each framework bridges the gap between a UX design (typically created in a tool like Figma) and the actual implementation in code. Modern GUI development often starts with designers, so the efficiency of moving from a design file to working software can greatly impact project timelines. Below, we compare how each framework handles the design-to-code transition, and how that affects fidelity and iteration speed.



Figma to Qt: Automated QML Generation

Qt offers an official plugin called Figma to Qt, which marks a significant advancement in embedded GUI development workflows. This plugin takes a Figma design and exports it as executable QML code. Instead of just getting static assets or specs from designers, developers using Qt can get a running UI prototype generated for them. Key aspects of the Figma-to-Qt workflow include:

- **Design-time validation:** As the designer works, the plugin checks the Figma design for any elements or properties that might be unsupported in Qt for MCUs. This early feedback helps ensure the design stays within feasible limits for the target hardware.

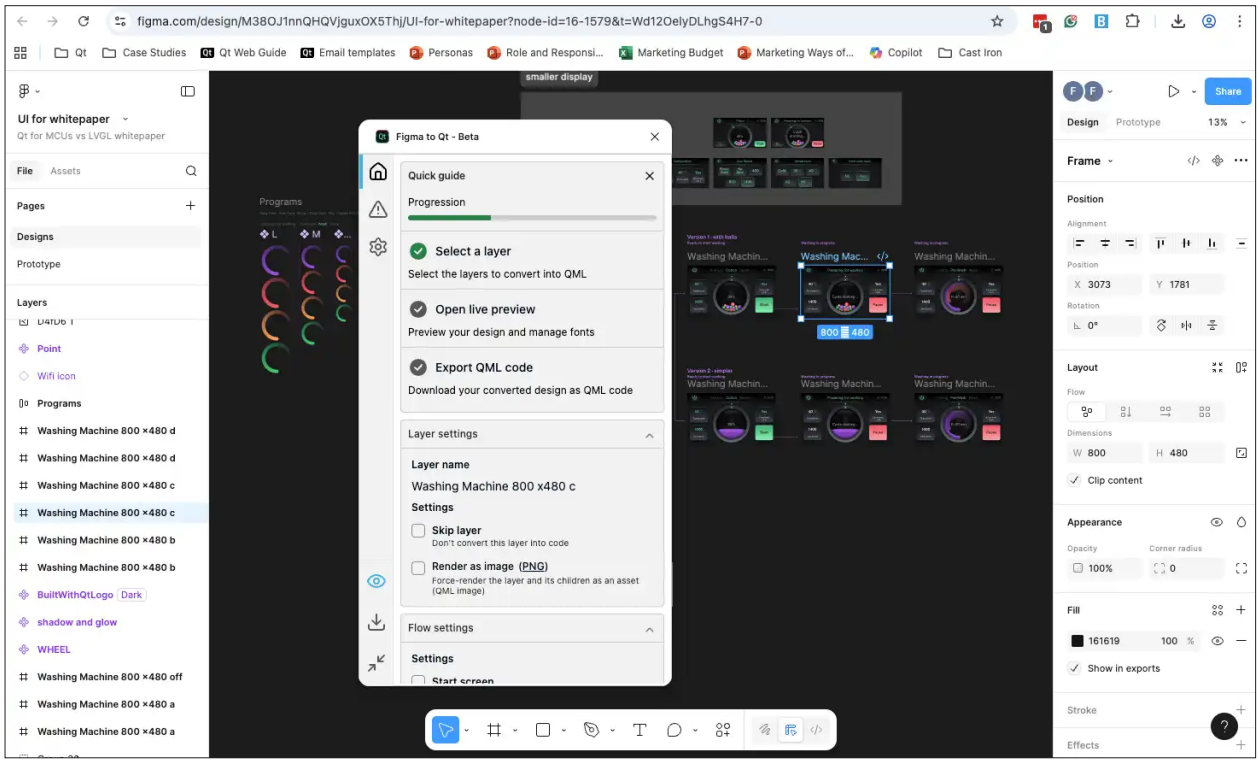
- **Code generation:** At export, the plugin produces a structured QML project. It preserves the full hierarchy of UI elements, layouts, and visual properties defined in Figma. Colors, fonts, spacings, images, and even basic interactive behaviors (e.g., component states) come through in the generated QML.

- **Immediate usability:** The generated QML is clean and structured enough to be used as a working starting point right away. In practice, developers can open it in Qt Creator or

Visual Studio Code to review the UI as imported and begin wiring it to application logic. For MCU-focused projects, Qt Creator remains the most natural fit within the Qt toolchain, while other editors can be used when teams standardize on them. The plugin effectively produces a first draft of the application—not fully final of course, but a substantial head start. We noted that the Figma to Qt plugin saves at least half of the development time for the project, even if some clean-up, further development, and optimization is needed afterwards.

- **Integration with development:** After import, the QML can be extended with C++ and/or additional QML to implement application functionality. Because the generated output is human-readable and serves as a solid foundation, developers can focus on behavior and integration rather than recreating layouts by hand. This also strengthens designer–developer collaboration: designers can hand off a runnable UI baseline instead of static screenshots, reducing guesswork and rework.

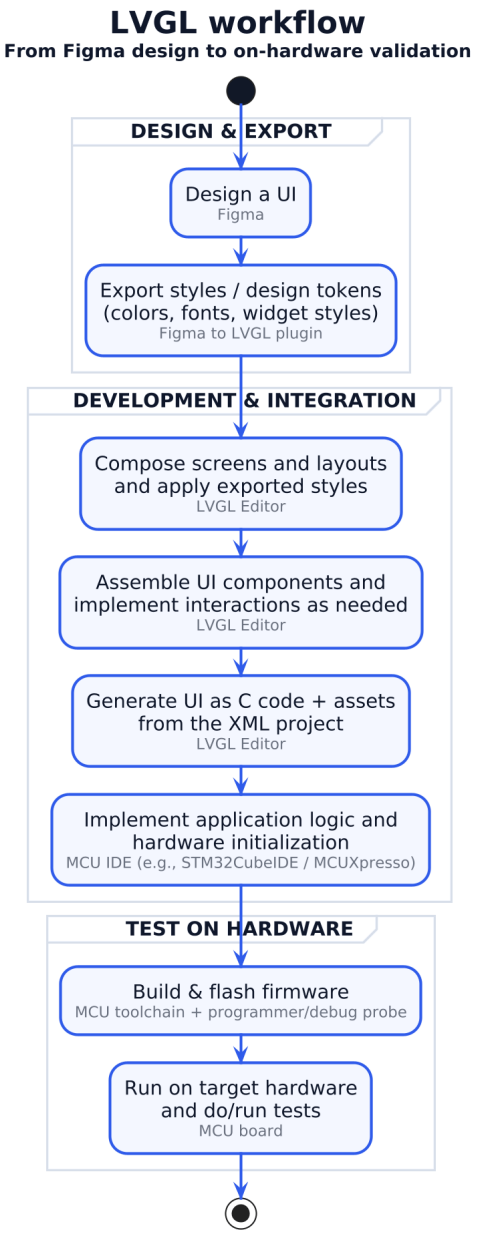
Overall, Qt's design-to-code pipeline automates a large portion of UI implementation. By trusting the Qt plugin to generate the bulk of the UI code, teams can iterate faster – if the design changes, the plugin can re-export updated QML, which the developers then merge or adjust as needed. The fidelity to the design is high because the code is derived from the actual Figma file. This approach clearly paid dividends in the comparative study, as reflected in the time savings.



Figma with Figma to Qt plugin

Figma to LVGL: Styles Export and Manual Rebuilding

LVGL's approach to design-to-code has historically been more manual, but it has evolved with the introduction of LVGL Pro tools. In the study, the team used Figma to LVGL plugin to assist the process. This plugin does not generate code; instead, it exports stylistic information from Figma: a list of colors, font definitions, text styles, and widget identifiers corresponding to the design. Essentially, it helps ensure that the LVGL project will use the same colors, fonts, and basic visual constants as the designer intended. However, all actual GUI screens, layouts, and interactions must be recreated by the developers:



- Using the LVGL Pro Editor, we manually placed UI components (buttons, images, labels, etc.) to match each Figma screen. We referenced the Figma design as a blueprint and adjusted positions, sizes, and properties in the editor's XML until the on-screen preview matched the design.

- We applied the exported styles to these widgets (for example, setting the theme colors or font sizes to those from Figma) so that things like brand colors and typography were consistent.

- Any dynamic behavior (screen transitions, input responses) had to be implemented either in the LVGL Editor's interaction editor or later in C code.

The LVGL Editor can be seen as an equivalent to Qt Design Studio for LVGL: it provides a visual environment to compose the UI with immediate feedback. It significantly improves productivity over writing C code for UI by hand. In fact, this was the first time LVGL had a robust official editor (previously, LVGL users often used third-party editors or coded directly). The editor produces an XML description of the UI, which then can generate C code. This is a boon to maintainability (XML is easier to diff and version than pure binary assets), and it enables features like an LVGL simulator on desktop to quickly test the UI on PC without hardware.

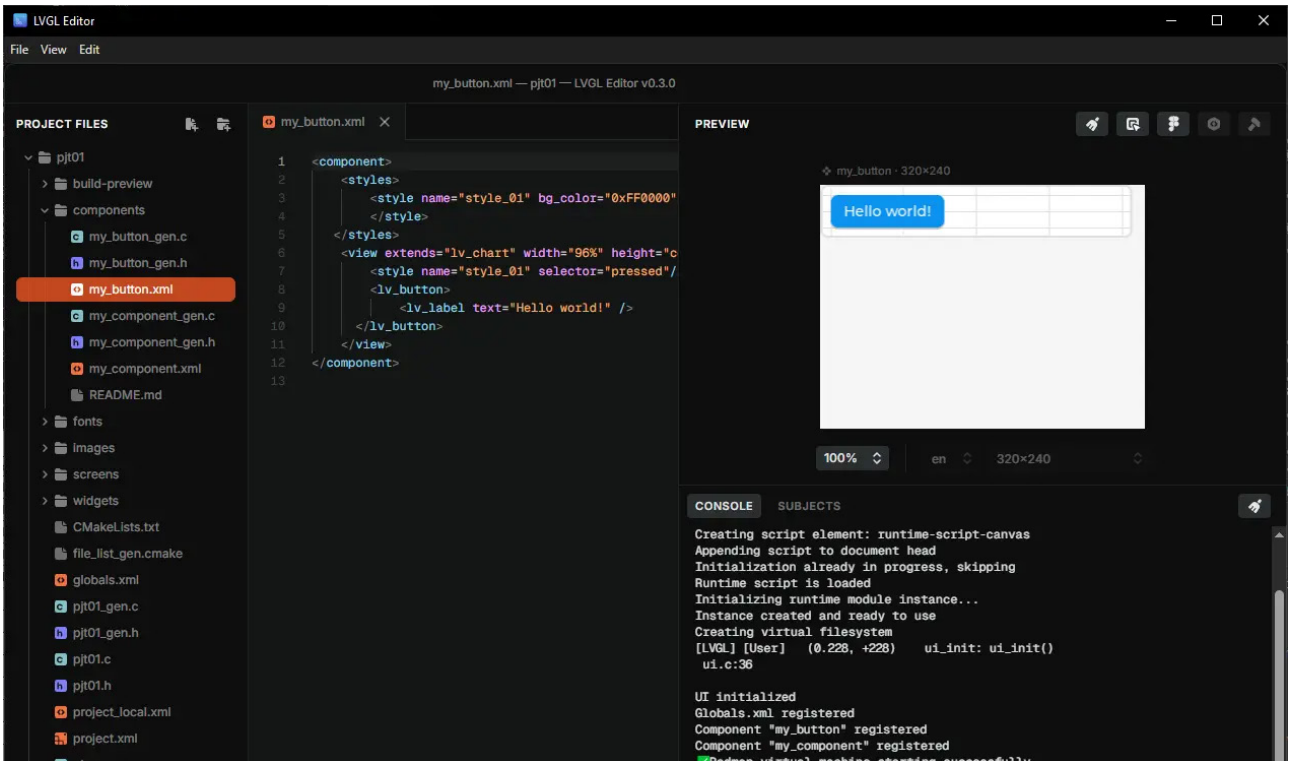
Despite these tools, the LVGL workflow required far more human effort to go from Figma to a running UI:

- The structure of the UI had to be reconstructed manually. Even with a perfect Figma design, developers needed to interpret it and rebuild every screen's widget tree in the editor. This is time-consuming and error-prone (e.g., a small misalignment or missing element could slip in).

- The LVGL plugin assured consistency in look (colors/fonts) but not in layout fidelity. Achieving pixel-perfect alignment with the design depended on the diligence of the person rebuilding the screens.

- Every interactive element (buttons, lists, animations) had to be wired up either in the XML (if supported by LVGL Pro's features) or later in C code, whereas in Qt's case some basic interactions were already present in the generated QML.

In summary, LVGL's design-to-code process in the study was partially assisted by tools but still largely manual. The introduction of LVGL Pro and its Figma integration has narrowed the gap compared to older LVGL workflows (which involved no design import at all), but compared to Qt's fully code-generating pipeline, it remains more labor-intensive. Developers using LVGL need to act as the "bridge" between the design and code, which impacts development time and requires stronger familiarity with the design details.



LVGL Editor (desktop)

Implications for Design Fidelity and Iteration

One benefit of both approaches is that neither required throwing away the design fidelity – both Qt and LVGL Pro allowed pixel-perfect previews and adjustments before deploying to device. In Qt's case, because the UI was generated in QML, the preview was effectively the real application running (just on PC). In LVGL's case, the LVGL Editor's preview approximated the UI accurately and with desktop backend it's also possible to run UI on PC. So in terms of visualizing the design on the target resolution and making tweaks, both frameworks provided a way to iterate without needing to constantly flash the device.

However, when it comes to iterating on design changes, Qt holds an advantage: if the designer updates the Figma file (say, changes a layout or a color scheme), the Figma to Qt plugin can re-export and produce updated QML in one step. Much of the previous integration work (like application logic added to the QML) can often be merged or is unaffected by purely cosmetic changes. With LVGL, a significant design change might force the developer to manually adjust the XML layout again (not just re-import styles), which is more effort. The integrated nature of Qt's tooling means design and development can proceed in parallel more smoothly – designers can drop new iterations and developers pull them in quickly – whereas with LVGL, design changes translate into developer TODOs to implement those changes.

From the study's perspective, this difference manifested in the development hours: the LVGL implementation simply takes more time building and fine-tuning the UI because we were effectively re-implementing the design, while the Qt implementation leveraged automation to jump-start that phase. It highlights a broader point for teams choosing a framework: if you expect a lot of design churn or need tight designer-developer collaboration, Qt's workflow offers tangible benefits in keeping everyone in sync. LVGL's approach can work if designs are static and only some minor styling changes or if the developers are comfortable with a more code-centric process, but it relies more on rigorous communication to avoid misinterpreting the design.

Development Environment and Deployment

Once the GUI code was created from the design, the next challenge was integrating it with the hardware and the rest of the system. Qt for MCUs and LVGL take notably different approaches in how developers build, debug, and deploy their applications onto microcontroller targets. This section examines the development environments and toolchains for each, as observed in the study's implementation on NXP and ST boards.

Qt for MCUs: Integrated IDE and Toolchain Support

Qt promotes a one-stop development environment through its Qt Creator IDE (or alternatively Qt Design Studio for UI focus), even for MCU projects. In the study, after generating the QML code, developers used Qt Creator as the central hub for all development tasks:

- We loaded the project (which included QML UI files and C++ code for application logic) into Qt Creator. Qt Creator has built-in support for Qt for MCUs projects, which means it knows how to compile the QML into optimized C++/C for Qt Quick Ultralite and link it with the Qt for MCUs libraries.

- Importantly, Qt Creator integrates with the MCU vendor's toolchains behind the

scenes. For example, to build for the NXP RT1064, Qt Creator was configured to use NXP's MCUXpresso GCC compiler and link against NXP's SDK for low-level drivers. Similarly, for STM32U5, it tied into the STM32Cube toolchain. This is achieved by setting up "Kits" in Qt Creator that point to the appropriate compiler, linker, and device programmer for each board. Once set up, developers did not have to open MCUXpresso or STM32CubeIDE for the GUI project – Qt Creator invoked those tools under the hood for compiling and flashing.

- Deployment and iteration (build-and-flash cycle) can be driven directly from Qt Creator: developers can build the firmware using the target MCU toolchain and flash it to a connected board via the vendor's programming utilities. However, on MCUs the on-target debugging experience is typically handled in the vendor or toolchain IDE/debugger (for example, IAR Embedded Workbench or Green Hills MULTI, depending on the platform) rather than in Qt Creator itself. Even with that split, the workflow still reduces context switching: Qt Creator remains the primary environment for editing QML/C++ and running fast desktop previews, while the final build/flash (and low-level target debug when needed) is performed with the platform's dedicated tooling.

The benefit of this approach is efficiency and consistency. The Qt developer writes code and tests it on desktop (since Qt for MCUs applications can often be run in a Desktop backend for early testing), then simply switches the build target to the MCU kit and deploys. All necessary low-level parts (like startup code, drivers for the display, etc.) were provided by Qt's board support packages or the vendor SDK and were configured in the Qt project. In the study, this meant bringing up the application on hardware was relatively straightforward once Qt Creator was set up with the board kits – a lot of heavy lifting (clock config, memory init, etc.) came from the provided Qt BSPs and the vendor's HAL, which Qt integrates with.

It's worth noting that Qt doesn't lock developers into its IDE. The project can be generated for other build systems (CMake based, etc.) and imported into tools like IAR Embedded Workbench or even a plain Makefile if needed. In contexts where companies have established build pipelines, this flexibility is valuable. But in the study and for most Qt users, sticking to Qt Creator simplified development considerably.

LVGL: Vendor IDE and Multi-Tool Workflow

LVGL, being just a library, does not provide an all-in-one IDE. Typical development with LVGL means using the MCU vendor's recommended tools or any general C/C++ toolchain the developers prefer. In the study we used:

- MCUXpresso IDE for the NXP board and STM32CubeIDE for the ST board. These are Eclipse-based IDEs provided by NXP and ST respectively, which include the necessary compilers, debuggers, and project wizards for those boards. They come with examples and board support packages which often include LVGL demos or templates.

- The LVGL-generated code (from the LVGL Editor) was added into an existing MCU project in those IDEs. Essentially, the developers started from a stock "hello world" project or an SDK example for the board (which sets up clocks, memory, etc.), then integrated the LVGL library and the C files output by the LVGL Editor for the UI. This requires manual configuration: adding include paths, linking the LVGL source or library, ensuring the LCD and touch drivers are configured, etc. In the study we used the vendor's provided LCD drivers and just plugged LVGL on top.

• Building and flashing was done through these vendor IDEs. For instance, in STM32CubeIDE, one would compile the project (which now includes LVGL) and use the built-in programmer (ST-LINK) to flash the board. Similarly for MCUXpresso with NXP's tool. There wasn't a unified "one click" from design to device; instead, after editing the UI in LVGL Editor, we had to switch to the IDE, rebuild, and flash.

A notable tool for LVGL is the PC simulator (based on SDL2), which the study also leveraged. This simulator allows running the LVGL UI as a desktop application for testing. We could test parts of the LVGL UI logic on a PC, which is faster than deploying to hardware repeatedly. However, the simulator is separate from the MCU build – it's essentially another project compiled for x86. Developers might use an IDE like Visual Studio Code or just CMake to compile the LVGL simulator app. This again illustrates the multi-tool nature: one might design in LVGL Editor, run a separate simulation on PC to verify, then go to the MCU IDE to run on actual hardware.

The LVGL Pro tools (Editor and CLI) add some integration features such as a command-line interface that can be used in continuous integration to auto-generate code and run tests. But they do not replace the vendor IDE when it comes to the final build and flash. In the study, after generating code from LVGL Editor, the steps to deployment involved moving that code into the MCU project and using the vendor's environment.

The consequence of this workflow is that developers have to manage multiple contexts:

1. The UI editor where they design and tweak the interface (LVGL Editor).

2. The code editor/IDE where they implement logic and integrate with hardware (MCUXpresso/STM32CubeIDE, etc.).

3. Possibly a simulator environment for quick tests (running on PC with SDL).

4. CLI tools or scripts to glue these together (especially if automating builds or tests).

This fragmentation showed up in our experience. For example, if a UI change was needed, we had to stop coding, open LVGL Editor, make the change, regenerate code, import it back to the IDE project, and rebuild. Each switch is a potential source of error (e.g., forgetting to copy a file, or differences between the simulation and real device if drivers behave differently). It also makes iteration cycles slower than in Qt's unified environment.

Vendor Tools and Low-Level Integration

One commonality between Qt and LVGL is that neither replaces the low-level board support code; they both rely on vendor SDKs for things like initializing the display, setting up memory, etc. In fact, the first step in both cases was installing the NXP and STM32 board SDKs and toolchains:

- For Qt, these were used behind the scenes via Qt Creator's kit.

- For LVGL, they were used directly in the vendor IDE.

So, regardless of framework, a developer still needs to interact with the MCU manufacturer's software (at least at configuration time). For instance, the study needed to configure the MCUs for use with a display (setting up DPI, resolution, frame buffer addresses) and both Qt and LVGL had to abide by those settings.

However, Qt for MCUs aims to wrap and simplify the vendor integration. Qt provides pre-

made Board Support Packages (BSPs)/platforms for certain boards, which include ready-made display drivers and examples of integrating the Qt rendering loop with the MCU's interrupt system, etc. In our context, using Qt's BSP/platform meant that we did not have to write a custom LCD driver or touch driver – Qt's runtime was configured to use the ones provided by the vendor. In LVGL's case, one might need to either write or configure an LVGL display driver (which essentially pushes LVGL's rendered frames to the display via SPI or parallel RGB interface). Many vendors like NXP and ST include LVGL drivers in their SDKs (or tools like NXP's GUI Guider generate a project with LVGL and drivers set up). Still, hooking everything together in LVGL is a developer's responsibility, whereas Qt's approach is more turnkey on officially supported boards.

In summary, when it came to deploying the app on devices, Qt for MCUs allowed us to stay within a single development environment, orchestrating the vendor tools invisibly, while LVGL required to directly use the vendor tools and juggle them with LVGL's own editor. This difference can impact productivity and the learning curve: A developer ramping up with Qt needs to learn Qt's way of doing things but once comfortable, they operate mostly in Qt's world. A developer using LVGL needs to be comfortable with the specific MCU's toolchain, the LVGL library, and how to integrate the two – which is a bit more bespoke for each project.

Testing and Quality Assurance

Ensuring the GUI works correctly and remains robust is another area where Qt for MCUs and LVGL diverge in philosophy. GUI testing on embedded devices can be tricky, because it involves verifying visuals and interactions on constrained hardware. The study highlighted how each ecosystem supports (or doesn't support) out-of-the-box testing and what that means for teams.

Qt for MCUs: Squish for MCUs – Automated UI Testing

Qt offers a dedicated automated testing tool called Squish for MCUs, which was utilized in the comparative study for the Qt implementation. Squish for MCUs is essentially a specialized version of the Squish GUI testing framework adapted for microcontroller targets. Its approach is to perform black-box testing of the application by interacting with it just like a user would, but in an automated fashion:

- Squish connects to the target device (e.g., the RT1064 board) via a small agent (using Qt's DeviceLink protocol). This allows the test system to capture the screen output of the MCU in real time and to inject input events (touch, button presses) into the running app. In effect, Squish can see what an end-user would see on the screen and simulate what an end-user would do with their fingers or a keypad.

- Test scripts are written in high-level languages like Python, JavaScript, or Tcl, making them accessible to QA engineers who may not know embedded C++. The QA engineer can write a script that says, for example, "press button X, verify that the screen shows the settings menu, then verify that a certain label reads 'Cycle started'," etc. Verification can be done via image comparison (ensuring the rendered screen matches a reference image) or even OCR for reading text on the screen.

- Because it's black-box, the tests do not need to know the internal code. They remain

valid even if the implementation changes, as long as the UI behavior looks the same. This is valuable when refactoring; QA scripts won't break due to code restructuring, only if the user-visible behavior changes (which is the correct sensitivity to have).

In practice, our QA could create a suite of automated tests that ran on the actual hardware, checking that the Qt GUI responded correctly and that animations remained smooth, etc. Squish for MCUs proved to have a gentle learning curve – it literally takes about an hour to get up to speed with it for an experienced software engineer or QA. This implies that even those not deeply versed in Qt or embedded development could contribute to testing, which is a major productivity win. In fact, the study noted that with Squish, a QA specialist (non-developer) could script tests, whereas for LVGL, it required a software developer to write test code. This difference underscores how Qt's more advanced tooling can enable parallelization of work: developers code while QA engineers independently create automated tests.

Squish's tests also cover performance aspects indirectly – by capturing actual framebuffers and timing, they can detect if something on screen stutters or if a frame takes too long. This was used in the study to confirm that the Qt app's animations were meeting frame rate targets on hardware.

LVGL: Unity/Ceedling and Simulator-Based Testing

LVGL does not have an official GUI testing tool comparable to Squish. Instead, testing an LVGL-based GUI typically involves more traditional embedded testing methods:

- Unit testing frameworks (Unity/Ceedling): Developers can write C test cases that instantiate LVGL widgets and simulate events to verify logic. For example, a test might create a slider widget, programmatically set its value, call the event handler, and assert that some internal state changed as expected. The Unity framework (not to be confused with the game engine; Unity is a C testing framework) along with Ceedling (a build system for C unit tests) is commonly used for this in embedded C projects. These tests run on a PC or maybe on an MCU in a special test mode.

- LVGL PC Simulator + Golden Images: A powerful method is using the desktop simulator to run the entire LVGL GUI and then capturing the drawn frames in memory. These can be saved as images and compared to "golden" reference images for regression testing. Essentially, the test would render a certain screen state and then do an image diff to see if anything changed unexpectedly. This catches graphical glitches. Some in the LVGL community have described setting up such test rigs (forum threads exist detailing how to automate LVGL testing using image comparisons and event injection).

- Manual and exploratory testing on device: Because there isn't an out-of-the-box tool, some testing for LVGL UIs inevitably falls to manual testers or developers who run the firmware on the device and interact with it to verify behavior. The study indicated that without Squish, creating a comparable automated test suite for the LVGL version would require significant custom effort from developers, which wasn't done within the study's timeframe. Instead, we relied on engineering to test the LVGL app's functionality.

The main drawback of the LVGL testing approach is that it's largely code-centric and developer-driven. Writing C test cases or scripts to pump events into the simulator is something an embedded developer would do, not a typical QA manual tester. As a result, the testing burden in LVGL projects often falls on the development team itself. This was explicitly noted: for

LVGL, a C/C++ developer is required to create tests, limiting the involvement of non-developer QA staff. In other words, testing is not as parallelizable; it competes with development for the same resources (embedded developers' time).

That said, the LVGL approach does have a silver lining: because it's at the code level, tests can introspect internal state. A developer writing a unit test can directly call into a widget's properties or the application state to assert things, which a black-box approach wouldn't see. This can lead to very fine-grained tests and is good for catching low-level logic errors. The trade-off is these tests are tightly coupled to the implementation – if you refactor the UI code or change how states are stored, all those tests may need updates.

Testing Philosophy: Black-Box vs White-Box

The study's outcomes on QA can be viewed through the lens of testing philosophy:

- Qt's approach (Squish) emphasizes black-box, user-centric testing. This aligns QA with what the end-user experiences – if the user can't see a bug, does it matter? Squish tests treat the GUI as a sealed component and ensure the user experience is correct. It also democratizes testing, letting non-programmers contribute.

- LVGL's approach (using unit tests and simulators) emphasizes white-box and code-level testing. It gives maximum control to developers to poke at the internals and ensure each widget or function works right. This can catch issues before they manifest on the UI, but it doesn't easily validate the overall user experience (e.g., animation smoothness, multi-component interactions) unless you simulate a lot of integration scenarios.

In practice, a robust project might use both approaches: for instance, if using LVGL, one might invest in writing a minimal custom testing harness that does something similar to Squish (some companies have done image-based tests for LVGL), and also write unit tests for the logic. But doing all that from scratch is effort that Qt provides largely out-of-the-box. Thus, the Qt team in the study could achieve a higher level of test coverage more quickly.

From a team perspective, having a tool like Squish meant that developers could focus on writing features while QA focused on writing tests in parallel. For LVGL, testing was another task on the developers' plate, reducing the time that could spend on features or requiring a larger dev team to cover both coding and testing.

Performance Benchmarks

A key part of the Qt vs. LVGL evaluation is how each framework behaves on constrained MCU-class hardware: not just "how fast it looks," but also what it costs in CPU time, flash, and RAM. The updated benchmark set below reports both runtime performance (CPU and frame rate) and footprint (flash/RAM), measured while running the same "washing machine" GUI flow and interaction scenarios on the two target platforms.

For LVGL, measurements are reported at both ~30 FPS and ~60 FPS because LVGL deployments on MCU-class hardware are often intentionally tuned to a lower refresh cadence for this type of GUI, rather than always targeting a maximum refresh rate.

Metrics collected

- CPU avg: average CPU utilization during active animations
- FPS min / FPS avg: the lowest and average frame rate observed during active animations.
- Heap peak / Stack peak: peak dynamic heap use and peak stack use observed during the run.
- Image size / Flash occupied: build output size and the amount of flash mapped/ consumed by the image.
- Runtime RAM reserved: RAM reserved by the build/runtime (static allocations, buffers, etc.).

LVGL is additionally shown in two operating points, because for this class of application on this class of MCU hardware it is common to run LVGL intentionally throttled to about 30 FPS in order to reduce CPU load and power consumption. LVGL makes that trade-off explicit through its refresh/update configuration.

- LVGL@30FPS: configured/scheduled for 30 FPS update cadence. This represents the practical low-load operating mode that many teams use for appliance-style or otherwise non-graphically-intensive embedded GUIs.
- LVGL@60FPS: configured/scheduled for 60 FPS update cadence. This shows how LVGL behaves when configured for a smoother, higher-refresh experience instead of the more conservative 30 FPS operating point.

With both Qt and LVGL, optimizations were applied: use of the 2D accelerators

- RT1064-EVK: PXP driver was used by both Qt for MCUs and LVGL
- STM32U5: NemaGFX was used by both, DMA2D draw disabled

RT1064 footprint

i.MX RT1064-EVK Storage and RAM Usage

Metric	Qt for MCUs	LVGL
Total RAM	641 KB	1,115 KB
Runtime RAM reserved	602 KB	1,070 KB
Total Flash	1,949 KB	2,132 KB
Image size	2,273 KB	2,270 KB
Flash occupied	1,949 KB	2,132 KB
Size of Assets	1,509 KB	1,956 KB

In this RT1064 build, Qt for MCUs and LVGL produced almost identical binary image sizes: 2,273 KB for Qt for MCUs versus 2,270 KB for LVGL. However, the deployed footprint tells a more nuanced story. Qt for MCUs occupied less flash, 1,949 KB versus 2,132 KB for LVGL, and reserved substantially less runtime RAM, 602 KB versus 1,070 KB.

This is the opposite of the common “LVGL is always smaller” intuition. In this build, Qt for MCUs is comparable in image size while being lighter in occupied flash by about 183 KB and in reserved runtime RAM by about 468 KB. The result highlights how sensitive embedded GUI footprint is to concrete implementation choices: asset handling, framebuffer and draw-buffer sizing, memory placement, linker layout, and platform glue code.

RT1064 runtime performance

i.MX RT1064-EVK Performance Metrics

Metric	Qt for MCUs	LVGL@30FPS	LVGL@60FPS
Max Framerate	60 FPS	30 FPS	40 FPS
Min Framerate	57 FPS	23 FPS	20 FPS
FPS avg	59 FPS	29 FPS	34 FPS
Max CPU Load	68%	65%	88%
CPU avg	58%	55%	74%
Heap peak	39 KB	42 KB	25 KB
Stack peak	0.3 KB	3.4 KB	3.4 KB

Interpretation:

- Qt for MCUs ran at near display-refresh smoothness in this RT1064 scenario: 59 FPS average, 60 FPS maximum, and a high minimum of 57 FPS, while using 58% average CPU load and 68% maximum CPU load.

- LVGL@30FPS delivered 29 FPS average, with a 30 FPS maximum and 23 FPS minimum, at 55% average CPU load and 65% maximum CPU load. This is slightly lower CPU usage than Qt for MCUs, but also roughly half the average frame rate and a much lower minimum frame rate.

- LVGL@60FPS increased CPU cost significantly, reaching 74% average CPU load and 88% maximum CPU load, but it did not reach a 60 FPS operating point in this setup. It averaged 34 FPS, peaked at 40 FPS, and dropped as low as 20 FPS. This suggests that, on this port and board configuration, raising LVGL's target cadence above the practical ~30 FPS operating point increased workload much more than delivered frame rate.

Practical takeaway:

If the product goal is consistently high frame rate on this platform, Qt for MCUs provided the strongest result in this benchmark, with the highest sustained FPS and by far the highest minimum FPS. LVGL appears well-suited to a lower-cadence operating point here, while the attempted 60 FPS configuration showed clear diminishing returns.

STM32U5 footprint

STM32U5G9J-DK2 Storage and RAM Usage

Metric	Qt for MCUs	LVGL
Total RAM	2,313 KB	1,373 KB
Runtime RAM reserved	1,582 KB	1,339 KB
Total Flash	4,964 KB	3,021 KB
Image size	5,060 KB	3,202 KB
Flash occupied	4,964 KB	3,021 KB
Size of Assets	4,271 KB	2,422 KB

On the STM32U5 build, LVGL is clearly smaller in both storage and RAM usage. Qt for MCUs occupied 4,964 KB of flash versus 3,021 KB for LVGL, a difference of approximately 1,943 KB. The generated image size shows a similar gap: 5,060 KB for Qt for MCUs versus 3,202 KB for LVGL. Runtime RAM reserved was also higher for Qt for MCUs: 1,582 KB versus 1,339 KB, a difference of approximately 243 KB.

However, this result should be interpreted together with the build configuration. In this STM32U5 build, the Qt for MCUs application was optimized for FPS rather than minimum footprint: resource compression was disabled, while caching was enabled. This is visible in the asset size as well: Qt for MCUs resources occupied 4,271 KB versus 2,422 KB for LVGL. In other words, the larger Qt footprint is not only a framework-level difference; it also reflects a deliberate performance-oriented configuration and uncompressed application resources.

STM32U5 runtime performance

STM32U5G9J-DK2 Performance Metrics

Metric	Qt for MCUs	LVGL@30FPS	LVGL@60FPS
Max Framerate	60 FPS	30 FPS	60 FPS
Min Framerate	59 FPS	28 FPS	59 FPS
FPS avg	60 FPS	29 FPS	60 FPS
Max CPU Load	17%	25%	85%
CPU avg	13%	17%	72%
Heap peak	731 KB	34 KB	20 KB
Stack peak	0.4 KB	0.2 KB	0.2 KB

Interpretation:

- On the STM32U5, Qt for MCUs reached a full 60 FPS average, with a 60 FPS maximum and a 59 FPS minimum, while keeping CPU usage very low: 13% average and 17% maximum. This is the strongest performance result in the table from a frame-rate-per-CPU perspective. The trade-off is memory: Qt for MCUs shows a much larger heap peak, 731 KB, which is consistent with the FPS-oriented configuration where caching was enabled.

- LVGL@30FPS stayed close to its target cadence, with 29 FPS average, 30 FPS maximum, and 28 FPS minimum. Its CPU usage was also low, at 17% average and 25% maximum, and its heap peak was much smaller at 34 KB. This makes LVGL@30FPS a compact and predictable lower-cadence operating point.

- LVGL@60FPS also achieved a stable 60 FPS result, with 60 FPS average, 60 FPS maximum, and 59 FPS minimum. However, it did so at a much higher CPU cost: 72% average and 85% maximum CPU load. Heap usage was low at 20 KB, but the CPU headroom cost is substantial.

Practical takeaway:

In this measured STM32U5 configuration, both Qt for MCUs and LVGL@60FPS achieved true 60 FPS-class smoothness. The difference is the trade-off profile. Qt for MCUs delivered 60 FPS with very low CPU load, but used more heap, due to enabled caching and FPS-oriented optimization. LVGL, configured with no resource compression (as well as Qt) and no caching, offered two clearer operating points: a lower-footprint ~30 FPS mode with modest CPU usage, or a 60 FPS mode with high CPU utilization.

Conclusions

These benchmarks do not identify a single universal winner. The observed results depend on the combination of framework, platform integration, target refresh rate, buffer strategy, and cache/compression configuration.

LVGL is shown at both ~30 FPS and ~60 FPS intentionally. For this class of embedded GUI on MCU-class hardware, LVGL is often deployed in a throttled ~30 FPS mode to reduce CPU load and power consumption. Presenting both operating points therefore reflects both a realistic low-load deployment mode and the higher-refresh alternative.

- On RT1064, Qt for MCUs delivered the strongest runtime result in this dataset, reaching 59 FPS average / 57 FPS minimum at 58% average CPU load. LVGL@30FPS delivered 29 / 23 FPS at 55% CPU, while LVGL@60FPS increased CPU load to 74% but achieved only 34 / 20 FPS. Qt for MCUs also had the smaller measured footprint on RT1064, with lower flash usage and substantially less reserved runtime RAM.

- On STM32U5, both Qt for MCUs and LVGL@60FPS achieved true 60 FPS rendering, but at very different CPU cost. Qt for MCUs reached 60 FPS average / 59 FPS minimum at just 13% average CPU load, whereas LVGL@60FPS required 72% average CPU load to achieve the same frame rate. LVGL@30FPS provided a lower-load 29 FPS operating point at 17% CPU. Footprint on STM32U5 favored LVGL, but these results should be interpreted together with configuration: Qt resource compression was not enabled in the measured Qt for MCUs application and caching was enabled, while the measured LVGL application used uncompressed C-embedded image resources and had LVGL image caching disabled.

The practical conclusion is that these frameworks should be evaluated as configurable operating points rather than as fixed identities. Different products require different balances of smoothness, CPU headroom, and memory footprint. For example, a washing machine, thermostat, home appliance, utility meter, or simple industrial controller may favor a compact ~30 FPS configuration with lower CPU and memory usage. By contrast, an automotive cluster, medical device UI, military operator panel, or premium animated HMI may justify a stable 60 FPS target and a larger memory budget in exchange for smoother rendering and stronger visual responsiveness.

Within this dataset, Qt for MCUs was particularly strong where sustained high frame rate and CPU headroom were important: it was the only measured near-60-FPS solution on RT1064, and on STM32U5 it achieved full 60 FPS with far lower CPU load than LVGL@60FPS for the price of higher memory footprint. The most appropriate product decision is therefore not “Qt vs. LVGL” in the abstract, but which framework-and-configuration combination most reliably meets the target user experience on the chosen MCU platform.

Source Code Footprint

Source Lines of Code Comparison

Platform	Qt	LVGL	Ratio
NXP	1,360	2,794	**2.1x**
ST	1,618	3,030	**1.9x**

The source-code comparison focuses on authored application code: lines that had to be written or edited for the washing-machine demo itself. Empty lines, comments, generated resource/font code, build output, framework sources, vendor SDK code, board startup code, linker scripts, and platform files used as-is are excluded.

Interpretation:

- On the NXP platform, the Qt implementation contains 1,360 authored SLOC, while the LVGL implementation contains 2,794 authored SLOC. That makes the LVGL implementation about 2.1x larger in authored application code.
- On the STM32U5 platform, the Qt implementation contains 1,618 authored SLOC, while the LVGL implementation contains 3,030 authored SLOC. That makes the LVGL implementation about 1.9x larger in authored application code.

Practical takeaway:

Qt for MCUs required substantially less handwritten application code in this comparison. The difference is mainly visible in the UI layer: QML expresses layout, state, and bindings declaratively, while the LVGL implementation requires more explicit imperative construction and update logic. The result is a smaller authored codebase to implement and maintain, while vendor-provided board support and framework/runtime code remain outside the comparison.

Development Productivity and Time-to-Completion

Perhaps the most impactful difference found in the Qt vs LVGL comparison was in how long it took to develop the application to completion with each framework. This encompasses all the workflow and tooling differences discussed earlier. Here we break down the development effort results and what they mean economically.

Development Effort Breakdown

The study tracked the hours spent by the development team in four main phases for each framework:

1. Design import and initial setup: Using the Figma export plugins and getting the base project running (including cleaning up the generated code).

2. UI implementation on desktop: Building out all the screens, interactions, and animations to match the design, tested on a PC or simulator.

3. Hardware bring-up: Porting/running the UI on the actual MCU hardware, integrating with drivers, fixing any platform-specific issues.

4. Performance tuning and final polish: Minor optimizations, ensuring frame rates are solid, and doing final testing rounds.

For each phase, Qt for MCUs consistently required equal or fewer hours than LVGL:

- Design to code: Qt ~8–12 hours, LVGL ~8–16 hours. Both spent some time here, but Qt was a bit faster on average. With Qt, those hours mostly went into tweaking the auto-generated QML (fixing small layout issues, optimizing assets) and establishing the project in Qt Creator. With LVGL, even though styles were imported, the higher end of time reflects manually setting up the initial screens in LVGL Editor.

- Building out the UI (desktop phase): Qt ~32–48 hours, LVGL ~48–80 hours. This was the largest gap. Qt's Figma-to-Qt gave a running start, so the team might have spent a couple of days fleshing out logic and maybe adding custom components, whereas the LVGL team spent up to a week or more constructing all screens and states from scratch. It underscores that manually coding or composing UIs is much slower. The delta here (potentially ~30 hours difference) is where the bulk of Qt's "30% faster" claim comes from.

- Device bring-up: Qt ~8–12 hours, LVGL ~8–12 hours. Roughly the same. Getting things running on hardware involved dealing with compilers, linkers, and drivers – and both frameworks had similar steps (with Qt maybe simplifying some parts, but LVGL benefiting from familiarity with vendor IDEs). This suggests that choosing Qt vs LVGL doesn't significantly change the time it takes to go from a PC prototype to something on an MCU, assuming both have decent documentation for the board – most of that time is spent wrestling with hardware issues that are common to any project.

- Performance tuning and testing: Qt ~12–16 hours, LVGL ~12–16 hours. Also comparable. After the app ran on the device, both teams spent a day or two optimizing and validating performance. Each would profile, adjust buffer settings, maybe simplify some animations, etc. This phase depends more on the hardware's idiosyncrasies and less on the framework.

In both cases, since the UI was already functional, it was more about fine-tuning.

Summing up the mid-range estimates: Qt for MCUs took ~74 hours vs LVGL ~100 hours of development effort in total. In other words, Qt's approach saved on the order of 26 hours, roughly a 25–30% reduction in development time for this project. In the best case (comparing upper bound of LVGL to lower bound of Qt) the difference could be even larger (~36 hours saved). And in worst case (lower bound of LVGL vs upper of Qt) maybe only ~16 hours saved. But consistently, Qt was faster to deliver the completed app.

The primary reason, as echoed earlier, is that Qt's integrated tools handle many repetitive or complex tasks automatically, whereas LVGL relies on the developer to do them manually. Specifically, generating code from Figma and having a single unified development flow (design → code → test in one environment) kept Qt's iteration loops tight. LVGL's team had more overhead in reimplementing UI and switching contexts.

Economic Implications

Time is money in software development. We extrapolated what the time savings mean in dollar terms:

- Using a notional engineering cost of $100/hour (this is a typical fully burdened cost for an experienced developer when factoring salary, overhead, etc.), saving 26 hours would equate to $2,600 less spent on that feature/UI. If the project was larger (the higher-end estimate of 36 hours saved), that's $3,600 saved.

- Now, on a single small project, a few thousand dollars might not seem huge, but consider an organization that does multiple GUI projects or updates each year. If each can be accelerated by ~30%, those savings multiply. For instance, 10 projects of similar scope could save ~$25k–$30k in development costs by using Qt over LVGL. That could pay for additional features or simply reduce the time-to-market.

- Moreover, these calculations don't even count the potential revenue impact of being earlier to market or having more polished UI due to time freed up. A 30% shorter development cycle could let a product hit store shelves a month earlier, which might be very valuable in a competitive market.

On the licensing side, the comparison depends on whether the team evaluates only the runtime library or the full professional workflow. LVGL's core runtime library is MIT-licensed and can be used without runtime license fees, which is a major advantage for cost-sensitive products and open-source-oriented teams. However, the LVGL workflow used in this study also involved LVGL Pro Editor. LVGL Pro is a commercial tooling product for many professional commercial use cases, with a free Community tier aimed at personal use, open-source projects, education, and public GitHub repositories. At the time of writing, LVGL Pro Starter is listed at $1,000/year/seat for startups and freelancers with yearly revenue up to $250k, while LVGL Pro Business is listed at $3,000/year/seat for companies of any size. Optional commercial support is priced separately.

Qt for MCUs normally requires a commercial Qt for Device Creation license. However, Qt also offers a Small Business tier for eligible companies. At the time of writing, Qt for Device Creation Professional Small Business is listed at 1,090 EUR/year. It is intended for companies with annual revenue and/or funding up to 1 million EUR/USD, with a maximum of three small-business licenses per company, limited annual support tickets, and eligibility subject

to Qt's terms and verification. While eligible, small-business customers are also exempt from purchasing separate Qt Device Creation Distribution Licenses.

- **The practical cost picture is therefore more nuanced than "Qt is paid, LVGL is free."** A regular LVGL runtime integration can indeed have zero license cost, and that remains one of LVGL's strongest advantages. But a professional LVGL workflow using LVGL Pro can also introduce paid tooling, while Qt's Small Business tier can make the Qt for MCUs toolchain accessible to qualifying startups and small companies. In the benchmarked project, the estimated Qt productivity saving was around 26 hours, or roughly $2,600 at a notional $100/hour engineering cost. In that context, licensing cost should be evaluated against engineering time, iteration speed, tooling quality, support expectations, distribution terms, and project risk—not only as an up-front expense.

The study's data provides a concrete case for that trade-off: paying for Qt's tooling yielded a net positive in productivity. However, every organization should run their own numbers. If you have a very low-cost team (say offshore developers at much lower rates) or the UI is extremely simple, the equation might tilt. But for many, that ~30% time reduction is compelling.

When Does a 30% Gain Matter Most?

The benefit of Qt's faster development cycle is not uniform for all projects. It tends to shine under certain conditions:

- **Complex HMIs:** The more complex the UI (lots of screens, fancy visuals, intricate state logic), the more time Qt's automation saves in absolute terms. A project with 2 screens might only save a few hours, but one with 20 screens and heavy animation could save hundreds of hours. In domains like automotive or advanced medical devices where UIs are large and sophisticated, Qt's advantage amplifies.

- **Cross-functional teams:** As noted, if you have dedicated designers, developers, and testers, Qt allows each to work in parallel with tools suited to them (Figma/Design Studio for designers, QML/C++ in Qt Creator for devs, Squish for QA). LVGL in such an environment can become a bottleneck, because the designers and QA can't contribute as effectively without the specialized tools – everything funnels back to the developers. So teams that have that mix of roles will see a bigger benefit from Qt's approach in terms of overall throughput.

- **Tight schedules / high urgency:** If you're in a race to deliver (perhaps a product launch tied to a specific date or needing to respond to competitors quickly), saving 30% of GUI dev time can be crucial. It might free up time for additional testing or iterations that improve quality. Or it might simply ensure you make the deadline with less overtime and stress. LVGL, being more manual, might be fine in a leisurely project but could become a liability if you need to churn out a lot of UI quickly.

- **Multiple product iterations:** If you plan to update or iterate on the UI frequently (maybe agile sprints, or versions 2.0, 3.0 down the line), the initial time saving continues to pay dividends in each update. Qt's pipeline can adapt to changes faster (e.g., new design versions, adding new features in the UI), so over a product's life, the cumulative time saved could be larger than the initial development alone.

In contrast, if a project has a long timeline and minimal UI complexity, a team might be able to manage with LVGL's slower process without issue. Or if cost of licenses is the overriding

factor and time-to-market isn't critical, some might accept longer dev time for zero runtime license cost. Each team must weigh these factors, but the data clearly favors Qt for anything where time or complexity are at a premium.

Strategic Ecosystem Advantages of Qt

Beyond immediate development metrics, Qt for MCUs benefits from being part of the broader Qt ecosystem, which can be a decisive factor for some organizations. LVGL, while popular, is a standalone library with a smaller ecosystem. This section explores how the larger Qt family and its commercial backing provide advantages in scalability, tooling, and long-term viability.

One Framework from MCUs to PCs

Qt is known for its cross-platform nature on desktops and embedded Linux; Qt for MCUs extends that down to microcontrollers. This unified technology stack means:

- Shared language and skills: Qt for MCUs uses QML (with Qt Quick Ultralite) which is essentially the same QML language used in Qt 6 on higher-end systems. A developer who knows how to build a UI in QML for an embedded Linux device or even a desktop app can transfer a lot of that knowledge to Qt for MCUs (with some adaptation for performance). Similarly, a designer familiar with Qt Design Studio can work on both MCU and non-MCU projects. This continuity is a major advantage if your company builds a range of products. By contrast, LVGL's pattern is completely different from, say, writing a Qt app for Linux – an engineer would have to learn a whole new paradigm if they move between an LVGL-based project and a Qt or web-based project.

- **Scalability of UI designs:** Because of Qt's consistency, it's feasible to take a UI initially made for an MCU and scale it up to run on an OS if needed (using full Qt), or vice versa. For example, you might prototype a UI in Qt Quick on a PC with rich effects, and then decide to deploy on an MCU – Qt for MCUs lets you reuse that work to a large extent, adjusting for MCU limits. Some companies design products in tiers (a high-end model with an i.MX 8 running Qt 6, and a low-end model with an STM32 running Qt for MCUs, but both offering a similar UI experience). With Qt, they can reuse parts of the UI and logic across these tiers, which is a huge efficiency boost. LVGL has no direct equivalent for high-end systems; if you start with LVGL on an MCU and later need a more powerful version on a higher chip, you might end up rewriting that UI in a different framework (or running LVGL on Linux which is possible but not common for complex apps).

- **Integrated tooling and services:** Qt's ecosystem comes with a suite of professional tools (some we've discussed: Design Studio, Squish, etc.) as well as things like Qt Creator IDE, Qt AI Assistant, and code coverage tools (Coco). All these are part of the same family. For instance, Qt's AI Assistant might help refactor QML code or suggest how to optimize something for Qt for MCUs. The key is that these tools work in concert. LVGL, being open-source, has third-party or community tools but not the same breadth or polish. It relies on general tools (like general IDEs, general test frameworks) which may not be tailored to UI development needs.

From a business standpoint, using Qt across your product line can reduce fragmentation of

technology and skills. Teams can collaborate more easily, assets can be shared (like a common style or component library in QML), and maintenance is simplified as there's a single vendor (Qt Company) providing support for the whole stack. LVGL in one product and Qt or Android in another means essentially supporting two different tech stacks internally.

Tooling and Productivity Ecosystem

We've touched on many of these, but it's worth summarizing Qt's broader tooling advantages:

- **Design workflow:** Figma to Qt plugin, Qt Design Studio – these keep the design to implementation loop tight within Qt. LVGL's comparable offering (LVGL Pro) has improved things, but is still maturing and not yet as automation-friendly (e.g., it doesn't fully generate code for everything, just styles and assets).

- **Development and debugging:** Qt Creator provides a consistent experience for MCU and non-MCU development, with powerful debugging and analysis tools. LVGL relies on the underlying MCU IDE or tools like GDB without a tailored GUI. Qt also supports modern workflows like VS Code integration for those who prefer it.

- **Testing and QA:** Squish for MCUs as discussed, plus Qt's Test Center for managing test results and Coco for test coverage analysis. These are enterprise-grade QA tools that plug directly into Qt projects. LVGL does not provide a comparable mature, end-to-end GUI testing product in the same category as Squish for MCUs.

- **Continuous Integration & Deployment:** Qt's build system (CMake) and tools can be set up in automated build/test pipelines readily – build for a host and target, run Squish tests on hardware, etc. LVGL's LVGL-Pro CLI does add some CI support (like validating XML, running XML-based tests), but it's not as extensive.

Overall, Qt's ecosystem provides a more out-of-the-box "professional grade" environment. This can reduce development friction (as the study showed in saved time) and also reduce risk – having a single vendor accountable means if something goes wrong (a bug in the framework, for instance), you have official support to turn to.

LVGL's ecosystem, being open, offers flexibility and a vibrant community. For some, the community support and the ability to inspect/modify the source freely is an advantage (we'll discuss that in control & extensibility). And LVGL Pro indicates an effort to bring more professional tooling to LVGL users. But as of the time of this writing, Qt's ecosystem is more mature and encompassing.

Long-Term Support and Stability

One often underappreciated factor in technology selection is how it holds up over the long term – e.g., 5+ years. Qt, being a commercial framework, follows a defined release and support strategy:

- Qt provides regular feature releases (Qt 6.x updates) and designates certain releases as LTS (Long-Term Support) where they commit to providing bug fixes and security patches for a duration (often 3+ years beyond release). For Qt for MCUs, which is tied to Qt's mainline to some degree, similar long-term maintenance is offered on specific versions. Commercial customers can thus standardize on an LTS release for a product and be assured of critical

updates without needing to constantly upgrade to new major versions.

- Qt's APIs are known for stability; even as Qt evolved from version 5 to 6, many core concepts remained compatible, and porting tools exist to help with transitions. This is valuable for companies maintaining products for a decade – they can update Qt minor versions for fixes with minimal changes to their code.

- Being a paid product, Qt comes with official support channels. If an embedded product runs into a bug in Qt for MCUs, the Qt Company can provide a fix or workaround (sometimes even a custom patch) as part of the agreement. That support safety net can reduce downtime and engineering effort if something goes wrong.

LVGL's long-term story is more community-oriented:

- LVGL releases are community-driven; they don't promise any given version will be supported for X years. If a critical bug is found, it depends on volunteers or the core team to issue a fix. There's no formal LTS branch of LVGL – typically, projects either freeze on a version or try to keep up with new releases which might have breaking changes.

- If you adopt LVGL 9.x for a product and 10.x comes out next year with improvements, upgrading might not be trivial if APIs changed. The onus is on your team to merge those changes or to backport security fixes if you choose to stay on 9.x. Some companies effectively treat an LVGL version as their own internal fork – doing maintenance themselves over the years.

- LVGL Pro (the paid offering from LVGL's maintainers) may offer some help (like priority support or maybe backports), but it's not the same scale as Qt's support organization. It's more about the editor tool and some consulting rather than guaranteeing the core library's upkeep.

- For a hobbyist or short product cycle, LVGL's model is fine. But for a safety-critical or long-life product (like automotive or medical device), depending purely on community support can be risky. You might eventually need to dedicate internal developers to maintain your fork, which is a hidden cost.

In the study context, we didn't explicitly measure long-term costs, but these factors were mentioned as part of the broader comparison. It contributes to the idea that Qt is a safer bet for products that need to be maintained a long time, whereas LVGL offers agility and no cost up front but may incur more maintenance cost later if the product lifecycle is long.

Frameworks Positioning: When to Choose One Over Another

Summarizing the comparative findings, we can outline scenarios where each framework is more advantageous – effectively providing guidance on choosing the right tool for a given project.

Choose Qt for MCUs if...

Your project is complex and involves a team of specialists. If you have UX designers, Q/A engineers, and embedded developers all working together on a rich GUI, Qt's toolchain allows each to contribute effectively. Designers can deliver QML via Figma→Qt, developers get a high-level language to implement logic, and testers have Squish to validate the UI on hardware. In such cases, Qt maximizes the team's productivity by leveraging each role.

- If your product's UI is a selling feature—smooth motion, rich transitions, and a "premium" feel—Qt for MCUs is a strong fit in this dataset, especially where high sustained frame rate and CPU headroom matter. On the i.MX RT1064, Qt for MCUs reached 59 FPS average with a 57 FPS minimum at 58% average CPU load, while LVGL@30FPS delivered 29 FPS average / 23 FPS minimum at 55% CPU, and LVGL@60FPS reached only 34 FPS average / 20 FPS minimum while consuming 74% average CPU. On STM32U5, both Qt for MCUs and LVGL@60FPS reached true 60 FPS-class smoothness, but Qt for MCUs did so at much lower CPU load: 13% average versus 72% for LVGL@60FPS, with the trade-off of higher heap usage due to the FPS-oriented cached configuration.

- Safety or regulatory compliance is required. If you're building something like a medical device UI or an automotive dashboard, the fact that Qt offers certified components and documentation for standards like ISO 26262 (ASIL-D) and IEC 62304 can save enormous effort. Qt for MCUs can integrate with Qt's Safe Renderer to handle safety-critical indicators. LVGL has no such certification; using it in a safety-critical system means you do all the work to certify it (which could cost far more than Qt's licenses). Also, Qt's security posture (commitment to the new EU Cyber Resilience Act compliance by 2026) is reassuring for IoT devices that need to be secure. If you cannot compromise on these aspects, Qt is the sensible choice.

- Your product line spans multiple device classes. Companies that plan to reuse software across MCU-based devices and more powerful devices (running Linux or Android) can benefit from Qt's cross-platform nature. For example, one codebase might target an MCU on a low-end model and an i.MX 8 with Qt 6 on a high-end model with more features. LVGL would not allow reuse in the same way; you'd likely end up using a different framework on the high end (maybe Qt or Android) and duplicating efforts.

- Time-to-market is critical or schedules are tight. As we saw, Qt can trim the development timeline significantly. If being first or fast in your market gives you a competitive edge, the productivity gains from Qt's automation and integration are a strong reason to prefer it. The up-front cost might be trivial compared to the opportunity cost of a delayed launch.

Choose LVGL if...

Your device is extremely resource-constrained or cost-sensitive. For tiny microcontrollers (Cortex-M0/M3 with very little RAM/flash) or when every cent of BOM counts, LVGL's lightweight footprint is ideal. You can run a basic GUI on a chip where Qt for MCUs simply can't fit or perform well. LVGL has been shown to run on chips with under 100 KB of RAM, which is outside Qt for MCUs' target range. If you're trying to add a small UI to a low-end device without upgrading hardware, LVGL is often the only choice.

- You require an open-source runtime with no runtime licensing costs. LVGL's core library is MIT-licensed, can be used freely, and gives teams full access to the source code. This is attractive for projects with open-source policies, strict runtime-license constraints, or teams that want to inspect and modify the GUI library itself. However, if the project uses LVGL Pro for professional editor, testing, Figma import, online collaboration, or CLI workflows, the tooling cost should be included separately in the project budget.

- Your development team is small and highly technical (and cost of their time is not the limiting factor). In cases where perhaps a couple of seasoned embedded C developers are doing the entire project (UI and firmware), they might prefer to avoid the overhead of a larger framework and do everything in C with fine-grained control. They may not need designer tools or fancy testing frameworks – they can manage with printf and their own ingenuity. LVGL can be very efficiently used by such developers who know exactly how to optimize for their hardware. It gives them ultimate control to dig into the library if needed. The absence of heavy tooling can be a boon if they are comfortable without it.

- The UI is relatively simple. If you're displaying a few info screens or basic menus, and don't need sophisticated transitions or a huge amount of UI logic, LVGL can cover it with minimal overhead. For a simple status display on a device, using Qt might be overkill. LVGL can achieve what's needed with a tiny footprint and straightforward code. The complexity of QML might not pay off for something a junior embedded developer could knock out in plain C in a day.

- You want to minimize third-party dependencies and keep the stack as simple as possible. LVGL is just a library; it doesn't impose a whole framework structure. For some, this is preferable as it means fewer points of failure or complexity. If your team likes a DIY approach (hand-picking each component of the software stack for drivers, OS, etc.), LVGL fits right in without dictating other parts. Qt, being full-stack, makes more decisions for you (which is helpful in general, but some may see it as limiting if they have niche needs).

In essence, Qt for MCUs is a top-down, high-productivity solution best for projects aiming high in UX quality and that have at least moderate hardware and budget to support it, whereas LVGL is a bottom-up, lean solution best for projects that prioritize minimal footprint, full control, and zero cost over rapid development.

Many real-world cases are not black-and-white; there are hybrid strategies (for example, using LVGL for a tiny secondary display and Qt for a main display in the same product line, each where appropriate). But having these criteria helps in making an informed decision.

Selection Criteria Summary

To help decide between Qt for MCUs and LVGL, consider the following key criteria (derived from both the study and general industry guidance):

- **Time-to-Market:** If you have aggressive deadlines or need to iterate quickly, Qt for MCUs' ~25–30% faster development can be crucial. LVGL might be acceptable for looser timelines or if developer hours are cheap relative to other costs.

- **UI Complexity:** For a complex HMI with many screens and animations, Qt scales better (thanks to QML and design tools) and the effort grows more linearly. LVGL's manual effort scales poorly as complexity increases (the more you have to code by hand, the more the workload balloons). For a simple UI, LVGL may suffice and be more memory-efficient.

- **Team Composition:** A cross-functional team (designers + QML/C++ devs + QA) will thrive with Qt's workflow. A small embedded-only team might lean LVGL if they prefer C and don't have separate designers or testers.

- **Hardware Resources:** On an MCU with, say, <256 KB RAM or very low clock speed, Qt might not even be an option – LVGL is designed for such constrained devices. On an MCU with ample RAM (1MB+), both can work; on one with 64KB, LVGL likely wins by default.

- **Regulatory Requirements:** If your product must pass stringent safety/security standards (automotive, medical, industrial controls), Qt's certified components and support for compliance (MISRA, ISO 26262, IEC 62304, etc.) give it a huge edge. LVGL would require doing all compliance work yourself or using the not-yet-released LVGL Safe (an initiative to make a safety-certified LVGL variant, still in early stages). For non-regulated consumer products, this may not matter.

- **Budget and Licensing:**

    - Qt for MCUs normally comes with commercial licensing costs, while LVGL's core runtime library is MIT-licensed and free to use. However, if the project relies on professional design-to-code tooling, the comparison becomes more nuanced. The LVGL Pro tooling used in this study has commercial tiers: at the time of writing, LVGL Pro Starter is listed at $1,000/year/seat for qualifying startups and freelancers with yearly revenue up to $250k, and LVGL Pro Business is listed at $3,000/year/seat for companies of any size. The free Community tier is aimed at personal use, open-source projects, education, and public GitHub repositories.

    - Qt also offers a Small Business tier that can significantly reduce the entry cost for qualifying companies. Qt for Device Creation Professional Small Business is listed at 1,090 EUR/year for companies with annual revenue and/or funding up to 1 million EUR/USD, with a maximum of three small-business licenses per company and eligibility subject to Qt's terms. While eligible, such companies are also exempt from separate Device Distribution licenses.

    - Therefore, LVGL remains the natural choice when zero runtime licensing cost, open-source licensing, or full source-code control is a hard requirement. But for teams using professional tooling, the decision should compare the total workflow cost: runtime licensing, editor/tooling licenses, support, distribution terms, and the

engineering time saved or added by each approach.

- **Long-Term Maintenance:** For products with long lifecycles (5+ years), Qt's guaranteed support and updates (LTS) reduce risk. LVGL's community support is good but not guaranteed; you may need in-house maintenance. If your org doesn't mind maintaining an open-source fork, LVGL is fine; if you want a vendor to handle the framework maintenance, Qt is safer.

- **Ecosystem & Extensibility:** Do you need additional modules like networking, file system, or other libraries integrated with the UI? Qt provides a full framework (sensors, connectivity, etc., in Qt 6), albeit Qt for MCUs is a subset. LVGL focuses purely on GUI – you'll rely on other libraries or custom code for everything else. If leveraging Qt's broader ecosystem (or reusing code from a Qt 6 app) is beneficial, that leans towards Qt. If you have a custom stack or you want minimal dependencies, LVGL is more appropriate.

By weighing these factors, one can make an informed framework selection. The good news is that both Qt for MCUs and LVGL are proven solutions within their domains – it's rarely a question of "can it be done at all," but rather which aligns better with the project's priorities.

Architecture and Full-Stack Considerations

The choice between Qt and LVGL also influences and reflects how you architect the overall software stack of your device. Qt for MCUs acts as a higher-level application framework, whereas LVGL is a lower-level graphics component. Here's how that plays out in system design.

Qt for MCUs as a Platform Layer

Qt for MCUs essentially sits on top of your MCU's OS/RTOS or bare-metal loop as a full application framework. It provides not just drawing routines, but a whole runtime (the QML engine), a UI tree structure, resource management for images/fonts, and integrations for input, etc. In a sense, it abstracts a lot of the device details – you program against Qt's APIs and Qt handles the underlying calls to drivers or OS. This is analogous to how full Qt (Qt 6) abstracts an embedded Linux system. The advantage is a well-defined architecture: your code is structured in QML/C++ according to Qt's patterns (signals/slots, model-view, etc.), which encourages separation of concerns and often leads to cleaner designs. The downside is you need to learn Qt's way and you might have less flexibility to do something unconventional, since Qt expects to control the main loop, etc. It's a trade-off between convenience and control.

LVGL as a GUI Library

LVGL, by contrast, is a composable component you add into your system. You have to provide it with a display buffer and tell it when to flush that to the screen, you call its tick functions, and so on. It doesn't enforce any particular project structure beyond how you use its API. This gives you a lot of freedom – you can integrate LVGL into an existing codebase easily, or call LVGL functions from any part of your code as you see fit. Your architecture can be tailor-made: you decide how to organize tasks if using an RTOS, how to handle data flow between UI and logic, etc. The risk is that without a guiding framework, it's up to the developers to maintain

good architecture. It's possible (and seen in practice) for inexperienced teams to entangle application logic with LVGL drawing code in a messy way, simply because the library doesn't impose structure. In the hands of a disciplined team, though, LVGL can be used in a very structured MVC pattern – it just doesn't force you to.

Integration of Other Software Components

If your device has other software components (connectivity stacks, sensors, file systems), Qt for MCUs will typically interact with those via its C++ layer. Many such components will have Qt-friendly integrations (for example, Qt has classes for serial communication, etc. if you're using Qt 6 on bigger systems; on MCUs you might directly call HAL APIs from your Qt app). LVGL, being agnostic, means you call whatever you want alongside it. There's no conflict because LVGL doesn't manage things outside GUI. Qt's approach might need adaptation – e.g., calling into your HAL or RTOS from Qt code is fine, but you have to ensure timing and threading align with Qt's expectations.

Pros and Cons – Architectural Trade-offs

- With Qt, you get an opinionated architecture that can accelerate development (less reinventing the wheel on how to organize code). It also aligns with higher-level Qt – so if you ever plan to have part of your system running on Linux with Qt 6, the patterns are similar. But Qt's structure may impose overhead (e.g., an event loop, dynamic object model) that might be "too much" for some tiny use cases or might limit doing extremely low-level optimizations.

- With LVGL, you get ultimate flexibility and minimal overhead – you only "pay" for what you use, and you can deeply customize. For instance, if you needed to heavily optimize drawing for a custom display, you can modify LVGL's draw routines since it's open source, or you can inject code at various points. Qt's internals are more of a black box in comparison (and Qt for MCUs source isn't open unless you negotiate for it). The flip side is more work on your part to assemble the building blocks and ensure the architecture is robust (especially for large projects).

One example scenario: Suppose your device needs to offload some UI tasks to a co-processor or needs to run two separate UIs (maybe two screens) – with LVGL, you could instantiate two LVGL instances or hack something to run on two cores. Qt for MCUs currently supports single display per app, single thread for UI. So LVGL might give exotic multi-UI flexibility where Qt wouldn't (this is a hypothetical case to illustrate flexibility).

Maintaining Separation of UI and Logic: Regardless of framework, good practice is to keep business logic decoupled from UI code. Qt encourages this by design (signals/slots, models for data, etc.). LVGL doesn't provide such patterns, so developers have to enforce it. If a team is not careful, they might write a lot of application logic inside LVGL event callbacks, etc., which can become tangled. Qt's structured approach can result in more maintainable code as projects grow, whereas LVGL projects rely on developer discipline. This echoes our earlier point in testing: Qt makes it easier to do the right thing (e.g., QA can test independently, code structured in QML/C++), LVGL doesn't prevent doing the wrong thing (like mixing logic in UI, or having only developers test).

In summary, Qt for MCUs is a more "batteries-included" application framework that can

simplify architecture at the cost of some flexibility and footprint, while LVGL is a lean component you integrate into your architecture, giving maximum control but placing more responsibility on the developers to build and maintain the overall system structure.

Lifecycle Maintenance and Stability

We already touched on this in the context of selection criteria, but let's dive a bit deeper into what using each framework means for maintaining your product over years:

Qt's Release and Support Model

The Qt (the company) typically releases updates frequently (three feature releases per year for Qt 6, for example) and designates long-term support versions that they'll maintain for longer. For Qt for MCUs, they've been releasing roughly a major version each year with minor updates in between, and aligning with Qt 6 versions for compatibility. What this means:

- If you base your product on, say, Qt for MCUs 2.x, and you need it to be stable for 5 years, you'd likely work with Qt to ensure you're on an LTS or you get backport patches for any critical fixes. Qt offers that to commercial licensees.

- Upgrading to a new Qt version later can bring performance improvements, new features (like Qt for MCUs 3.0 introduced more modularity, etc.), often with minimal changes needed in your code. Qt tends to preserve source compatibility as much as possible, especially within a major version series.

- You have a vendor commitment: if a critical bug or security issue is found, Qt is accountable to fix it (for commercial users). That's a big plus in terms of risk management.

LVGL's Community-Driven Evolution

LVGL is on a roughly annual major release cycle as well (e.g., v8, v9), driven by open-source contributions and the core team:

- LVGL might introduce breaking changes in a major release (for instance, between v7 and v8 there were API changes). If you want the new features, you have to adapt your code. If not, you stick to the old version.

- There's no official backporting of fixes. The community might release a minor version if there's a known bug, but there's no guarantee. Often, if a vulnerability or bug is found, you're expected to update to the latest version that has the fix.

- Some vendors, like NXP and ST, sometimes stick with a particular LVGL version in their SDK for a while. For example, NXP's GUI Guider tool might use LVGL 7 or 8 and they'll maintain that integration for some time. But again, that's vendor-specific and not centralized like Qt. If you used NXP's tool and they updated it to LVGL 9, you might have to manually migrate if you want the new NXP features.

Implications for a product team

- With Qt, you plan updates in collaboration with Qt's roadmap. You might decide to lock to an LTS and only take patches, or to upgrade to a new version if a feature is compelling. But you have control and support in doing so.

- With LVGL, you either become an active follower of the project (updating regularly and dealing with changes) or you freeze on a version and accept that you'll handle any issues yourself. Neither is inherently bad – many embedded products freeze a library version – but you need to allocate developer time for that maintenance. Some companies using LVGL assign someone to keep an eye on the LVGL GitHub for relevant fixes to cherry-pick.

- The ecosystem stability differs too: Qt's tooling like Design Studio, Squish, etc., will continue to be supported and improved (and you get those improvements). LVGL's ecosystem (LVGL Editor, etc.) is newer and subject to change – e.g., LVGL Editor might change file formats or capabilities as it evolves, which could affect your workflow.

Commercial Support vs Community Support

- **Qt's model:** you can file support tickets, you can even pay for consulting if needed. There's a formal channel to get help. And the Qt community (forums, etc.) is also there, but the guarantee comes from the company.

- **LVGL's model:** primarily community forums and GitHub issues. The core maintainers are quite responsive and helpful to the community, but they have limited capacity and no contractual obligation to, say, resolve your specific issue by your deadline. LVGL Pro (commercial) might give more direct support, but that might cover the Pro tools rather than deep fixes in the core library.

- For a small project/hobbyist, community support is often enough (and one avoids costs). For a large enterprise, the unpredictability of community support might be a concern – which is why many large companies lean towards frameworks with professional support for mission-critical stuff, even if the tech itself is open-source.

In the context of the study's findings, they highlight that using Qt for MCUs can de-risk long-term maintenance, whereas with LVGL you should be prepared to self-support to a greater degree. In practice, there are plenty of products running LVGL in the field successfully. It often comes down to the nature of the product: for a gadget or consumer device where occasional firmware updates are fine and end-of-life is shorter, LVGL is fine. For something that must be maintained a decade with compliance, Qt's stability and support may well justify its cost.

Security, Safety, and Robustness

Modern connected devices must consider cybersecurity and (in certain industries) functional safety. The frameworks differ significantly in how they address these non-functional requirements:

Qt for MCUs – Security and Safety Posture

Qt for MCUs benefits from Qt Company's broader initiatives in these areas:

- **Cybersecurity (EU CRA):** The EU Cyber Resilience Act will mandate certain cybersecurity standards for connected devices. Qt has announced commitment to make its products (including Qt for MCUs) compliant by mid-2026. This implies that Qt will implement secure development practices, provide vulnerability disclosure mechanisms, and ensure update mechanisms that help device makers meet regulatory needs. For a device maker, using Qt could simplify proving compliance, since Qt can supply documentation on how it meets these requirements.

- **Functional Safety Packages:** Qt offers optional safety-certified packages for their framework. While historically these were for Qt on larger systems (with Qt Safe Renderer, etc.), Qt for MCUs is expected to integrate with those. Qt Safe Renderer (QSR) is a module that allows rendering critical UI elements (like a speedometer or warning light) in an independent, simple manner certified to ISO 26262 ASIL-D. This is a big deal in automotive and similar domains. If you use Qt and QSR, you get documentation ready to include in your safety case – essentially Qt has done the hard work of proving the rendering code is reliable to a certain level. LVGL has nothing similar; you'd have to, for example, implement a redundant gauge for a speedometer or certify LVGL's code yourself which is a monumental task.

- **Support for Standards:** Qt covers not just automotive (ISO 26262) but also medical (IEC 62304) and others via their certification offerings. For instance, they have TÜV certified Qt versions for those standards. This means if you're building, say, a dialysis machine UI with Qt, you can possibly buy Qt's medical safety package and drastically reduce the paperwork and testing needed to certify the UI software. LVGL being MIT-licensed gives no such evidence; you'd have to perform static analysis, unit tests, etc., for all of LVGL's relevant code and demonstrate it meets safety requirements, which could be far more costly than simply using Qt's pre-certified route.

- **Robustness & Updates:** Qt's commercial nature ensures you have a pathway for receiving patches for critical bugs and security issues for the long term. If a vulnerability in Qt for MCUs is found, Qt Company will notify and patch (for commercial users), aligning with security standards. Qt's emphasis on LTS and support gives device makers a sense of reliability – there's someone accountable if a problem arises.

LVGL – Community-Driven Security and Safety

- **Security:** LVGL doesn't have a formal security policy beyond typical open-source project practices. That means if a vulnerability is found by a user, it might get reported on the forum or GitHub and fixed in a future release, but there's no SLA or official advisory process. No commitment to CRA or similar, since it's not a company product. For a connected product, the burden is on you to monitor LVGL's repository for security-related fixes and to integrate them. Also, to ensure secure usage, you have to treat LVGL like any third-party code – e.g., run static analysis, fuzz testing if applicable, etc. Qt likely does those internally as part of compliance. That said, because LVGL runs typically on microcontrollers (often with no OS), the attack surface is smaller than a complex OS framework. But as GUIs get

more connected (like over network or pairing with mobile apps), any buffer overflow in rendering could be a vulnerability. Without formal support, you must be vigilant.

- **Functional Safety:** LVGL has no certification or official documentation for use in safety-critical systems. If you plan to use it in, say, an automotive cluster or a medical device, you have to do a full hazard and risk analysis of it as if it were your own code. The cost of that can indeed exceed a Qt license by far, as the study notes. There is mention of "LVGL Safe" – the LVGL team has indicated interest in creating a safety-certifiable subset of LVGL (they posted about MISRA compliance and an ISO 26262 effort on LinkedIn). However, as of now that's not available as a product. So any team using LVGL in a high-safety context is effectively on their own. In practical terms, some have used LVGL in such contexts by heavily sandboxing it (e.g., treating the entire UI as QM (quality managed, non-safety) and keeping anything safety-related out of it). That limits what you use LVGL for in those products – maybe just the infotainment part, not the critical alerts, for instance. And if any part needs to be critical, you'd implement that separately or redundantly.

- **Maintenance & Robustness:** Over a long term, LVGL's reliability depends on how actively you update and monitor it. If a bug is found, are you prepared to dive in and fix it? Many companies using open source maintain internal forks for stability. Also, if you need a fix now, the community timeline might not match yours; you may need internal expertise to implement fixes/workarounds. On robustness, LVGL has a pretty good reputation for stability given its wide use, but formally, no one is "on the hook" to guarantee it won't crash or leak memory, etc. With Qt, you have some expectation of enterprise-level QA (and if something is wrong, you can escalate it). We do note here that LVGL Pro could offer some support, but it's not an end-to-end safety solution – it's more about improving the dev process, not certifying the runtime.

Comparative View for Safety/Security

- If you have strict requirements (e.g., you are going to get audited by a regulatory body for your software process), using Qt can offload a lot of compliance work since you can leverage Qt's certifications and processes. Using LVGL means you'll have to show your own process to ensure LVGL is safe/secure, which can be done but is extra work.

- If your device is non-critical and mostly offline (say a toy or a simple consumer gadget), then LVGL's lack of formal security process is probably fine; you just test it like any other component.

- For connected devices, consider how you will deliver security updates if an issue is found. With Qt, you'd get a patch from Qt; with LVGL, you'd have to integrate the open-source patch (which might also come quickly, but it relies on you noticing and doing it).

- A point to note: sometimes companies mitigate LVGL's risk by wrapping it with their own abstraction and limiting usage to a safe subset. But that is effectively building your own framework on top of it.

In essence, Qt for MCUs is geared for industrial-strength, regulated, long-term products with an emphasis on safety and security, whereas LVGL caters well to open-source and simpler use cases where such guarantees aren't required or can be managed by the user. If your project lives in a heavily regulated world, Qt is likely worth it purely for the compliance support.

Hybrid Local and Remote UIs

Many modern embedded devices not only have a local GUI, but also connect to remote interfaces – like a smartphone app or a web dashboard. How do Qt and LVGL factor into an architecture where you want both a local HMI and a remote UI?

Common Patterns for Local + Remote UX

- **Mirrored UI Streaming:** One approach is to run the UI on the device and simply stream it (as video or via a protocol) to a remote display. For example, some devices use VNC or remote frame buffer protocols. Qt actually has a feature called WebGL streaming where a Qt UI can be mirrored to a web browser. LVGL, theoretically, could be paired with something like a VNC server on an MCU (if resources allow) but that's not typical. This approach doesn't require two implementations – you literally replicate the local UI remotely – but it often isn't interactive or optimized (or even feasible on low MCU).

- **Dual Front-End:** Another common pattern is to have two separate front-ends: the device has its UI (Qt or LVGL), and then you have a separate mobile app or web app that talks to the same backend on the device (maybe via Bluetooth, Wi-Fi, etc.). So you maintain two codebases for UI but share the underlying logic via a communication protocol.

- **Headless Engine + Multiple UIs:** In some designs, the device might not even have a full UI of its own (maybe just an indicator), and most of the UI is remote; or vice versa. Or the device could run a headless service and multiple UIs (local and remote) connect to it over network.

Now, how do Qt and LVGL support these?

Qt-Centric Architectures

Qt has a big advantage if you want both local and remote UIs: you can use Qt across the board. For instance:

- Qt for MCUs for local HMI + Qt 6 for a remote UI (on a mobile app or a web app with Qt's WebAssembly). Because both speak QML, you might even reuse QML files or at least the design. You could have a shared UI component library that is used in Qt for MCUs and in Qt Quick on a higher platform. This way, the local and remote UIs stay consistent in look and feel with less duplicated effort.

- Qt's WebGL streaming: Qt can stream a live UI to a web browser. This means you could control the device remotely without writing a separate web UI – the actual UI from the device is projected. This is more relevant for powerful boards though; on an MCU with Qt for MCUs, I'm not sure WebGL streaming is supported (likely not directly, since Qt for MCUs doesn't have all Qt 6 features).

- If you need a separate native mobile app, Qt also offers cross-platform development (Qt on Android/iOS). So you could, in theory, also use Qt to make your companion mobile app, sharing code with the device UI. That is a unique proposition of Qt – one technology for embedded and mobile.

LVGL-Centric Architectures

LVGL doesn't extend to mobile or web. So typically:

- **Local LVGL UI + separate web/mobile app:** You'll likely implement the remote UI using web technologies (JavaScript, etc.) or native mobile frameworks. The LVGL team might in future provide some bridging (maybe an LVGL viewer on web?), but currently remote UIs are independent. They just communicate with the device's firmware over an API (say REST or MQTT). - The advantage here is you can choose the best tech for remote – maybe a cutting-edge web framework for your cloud dashboard, etc. The disadvantage is nothing is shared – you have two completely different codebases for UI. Ensuring consistency in design falls to design guidelines rather than shared code.

- There is an LVGL viewer (viewer.lvgl.io), but that's more for previewing UIs during development, not for end-user remote control. So not directly a solution for product remote UI.

Communication & Separation

Regardless of framework, a good approach is to design a clear interface between the device logic and any external UI:

- Qt or LVGL, you'd implement some communication (could be as simple as a BLE GATT profile, or as complex as an on-device web server).

- It's recommended to keep business logic in the device and make UIs (local or remote) clients to that logic. Qt encourages this by being able to run parts headless, but on MCUs you often have a single process so you might structure it in tasks or modules.

- Both Qt and LVGL architectures can be made to work in these multi-UI scenarios, but Qt offers more inbuilt support to create matching UIs on different platforms.

- **When to leverage Qt's strength:** If a selling point of your product is that the user can seamlessly switch between using the built-in screen and an app, and have a cohesive experience, Qt's ability to reuse components could save a lot of design and dev effort. For example, you might have a thermostat with a small screen (Qt for MCUs) and also a smartphone app (perhaps built with Qt/QML or Qt's HTML5 integration). They could share UI logic and design assets.

- **When LVGL is fine:** If your remote UI is just a basic status monitor or something and you have web developers to do it, having two code paths isn't a big deal. Many IoT devices today do exactly that: embedded C GUI locally, and a totally separate mobile app – they manage to keep features in sync via good planning.

Headless or multi-screen considerations

Qt doesn't natively support multiple independent UIs on one MCU (like two screens each with separate UI, though you can have multiple displays showing one extended UI). LVGL can be instantiated on multiple displays if you handle it, I believe, which might be needed for some multi-screen setups.

If the product might drop the screen in a variant (headless variant that only has remote UI), with Qt you might have overhead if you included Qt for MCUs just for consistency. With LVGL, if you drop the screen, you drop LVGL and nothing else changes.

In conclusion, Qt provides a path to a unified multi-platform UI strategy (MCU, MPU, mobile, web) which can be a strategic advantage for product lines that need it, whereas LVGL focuses purely on the local UI, and any remote/web integration has to be built separately with other technologies. This can be a deciding factor for companies thinking about their ecosystem of devices and apps as a whole.

AI-Assisted Development and Agentic Workflows

AI has begun to change embedded GUI development less by replacing frameworks and more by changing how teams move through the workflow. Modern coding models and coding agents can inspect repositories, propose multi-file changes, generate C/C++/QML/XML, explain build failures, run local commands or tests under approval and even implement Figma design, if it's well structured. For embedded GUI projects this matters because much of the effort is not the rendering code itself, but the translation between design assets, generated UI, application logic, board support, build configuration, testing, and iteration.

Qt for MCUs

For Qt for MCUs, the impact is already visible in the toolchain. Qt AI Assistant can help with QML, C++, Python, code explanations, refactoring, code fixes, unit tests, and documentation. This is particularly relevant for Qt because QML is declarative, component-oriented, and close to the language used by UI engineers and designers. That gives an assistant a structured textual representation to reason over. Qt Creator's MCP support is also important: it allows AI assistants to interact with Qt Creator for project, build, debug, issue, and file-management tasks. In practice, this points toward an agentic workflow where an assistant can read the project, modify QML/C++, trigger a build, inspect warnings, and propose fixes without leaving the IDE context.

The QA side is also changing. Squish AI Assistant adds LLM-based help inside the Squish test environment, including support for writing and refactoring test scripts, explaining failures, and analyzing logs. This does not replace hardware-in-the-loop testing or deterministic GUI test automation; rather, it reduces the effort of creating and maintaining test assets. For embedded UI teams, this is valuable because GUI test automation often becomes a bottleneck. AI assistance can make automated testing more approachable and can shorten the time between a failed test and a useful diagnosis.

LVGL

For LVGL, the situation is different. LVGL does not currently appear to have a dedicated first-party AI assistant comparable to Qt AI Assistant or Squish AI Assistant. However, LVGL Pro is moving in an AI-friendly direction because it represents UI structure in text-based XML, supports C code generation, provides a CLI, and fits naturally into Git-based and automated workflows. This makes it practical for general coding agents to help write or modify XML

components, generate C integration code, adjust styles, create test definitions, and interpret build or simulator failures. LVGL's Figma integration also improves the machine-readable handoff, although it remains style synchronization rather than full automatic design-to-code import. As a result, AI can reduce some of LVGL's manual implementation burden, but it does not yet remove the need for developers to rebuild and structure the UI carefully.

What to expect

MCP is likely to become a key infrastructure layer for both ecosystems. Instead of hard-coding one AI assistant into one tool, MCP allows agents to connect to IDEs, local files, build systems, documentation, issue trackers, CI pipelines, and hardware-test systems through a common interface. For Qt, this is already concrete through Qt Creator's MCP server. For LVGL, the natural path is to expose the LVGL Pro CLI, simulator, generated-code checks, and project-specific firmware commands as local tools that an agent can call. In both cases, the strongest workflows will combine an AI agent with deterministic scripts and human approval rather than allowing free-form changes directly into production code.

The near-future impact is therefore nuanced. AI will narrow some productivity gaps by helping LVGL developers with XML/C implementation and by helping Qt developers move even faster through QML, C++, and test automation. But AI is unlikely to make framework choice irrelevant. Frameworks with structured project files, strong CLI/IDE integration, automated testing, and clear diagnostics will benefit more from agents because they give the model reliable actions and feedback loops. In that respect, Qt currently has a stronger first-party AI/tooling story, while LVGL benefits from being lightweight, open, textual, and easy to pair with general-purpose coding agents.

For product teams, the practical recommendation is to treat AI as a workflow accelerator, not as a substitute for engineering discipline. AI-generated code should still pass normal review, static analysis, build checks, simulator tests, hardware-in-the-loop validation, and security or safety review. The teams that gain the most will be those that prepare their projects for agents: clean component boundaries, version-controlled UI descriptions, repeatable builds, CI scripts, documented board bring-up steps, and clear test commands. In that environment, both Qt for MCUs and LVGL can benefit from the latest AI models, but Qt's integrated AI, MCP, and QA tooling currently give it a more complete out-of-the-box path for agent-assisted development.

Conclusion

The comparative analysis of Qt for MCUs and LVGL illustrates that both frameworks are capable of delivering modern GUIs on microcontrollers, but they cater to different priorities and project styles. Qt for MCUs shines in situations where development speed, UI sophistication, and ecosystem integration are paramount. Its seamless design-to-code workflow, integrated tooling (IDE, automated testing), and alignment with a larger cross-platform framework result in roughly 25%–30% less development time for complex interfaces. This makes Qt an excellent choice for projects with ambitious UI/UX goals, multi-disciplinary teams, or strict market deadlines. Moreover, for products in regulated or long-lived domains, Qt's offerings in safety certification and long-term support provide peace of mind and reduced risk.

On the other hand, LVGL appeals to those who value minimal footprint, full code control, and zero runtime licensing cost. Its lightweight C library can run on very constrained hardware that Qt cannot, enabling interactive displays on the most cost-sensitive devices. LVGL's open-source runtime means there are no runtime library fees, although professional LVGL Pro tooling may introduce separate costs. However, this comes at the cost of more manual effort in development and a reliance on the development team to fill in the gaps in tooling, architecture, and testing that Qt provides out-of-the-box.

In summary, Qt for MCUs is a comprehensive solution well-suited for complex, fast-paced, or safety-critical projects, whereas LVGL is a lean solution well-suited for simple, resource-limited, or budget-constrained projects. Many organizations may even use both, selecting Qt or LVGL on a case-by-case basis per product tier. The "best" choice is context-dependent: it should be guided by the specific requirements of hardware capability, team skillset, project timeline, user experience expectations, and regulatory needs.

By carefully considering the factors detailed in this whitepaper – from development workflow and performance results to ecosystem and maintenance implications – stakeholders can make an informed decision between Qt for MCUs and LVGL that aligns with their product goals and business constraints. Both frameworks have proven their worth in the field; the key is to choose the one whose strengths best match the success criteria of your project. With either choice, leveraging the available tools and adhering to best practices will be crucial in delivering a robust and engaging embedded GUI for the next generation of devices.

References

1. Qt Company – Figma to Qt (Design Integration) – Figma to Qt (Official Qt Design Integration page)

2. Qt Company – Qt for MCUs Product Page and Supported Platforms – Qt for MCUs Product Page, Supported Hardware Platforms (Qt for MCUs documentation)

3. LVGL Official Documentation – LVGL Overview and PC Simulator – LVGL Documentation (Introduction), Simulator on PC (Official LVGL docs)

4. NXP Semiconductors – i.MX RT1064 Crossover MCU Product Summary – NXP i.MX RT1064 Product Page

5. STMicroelectronics – STM32U5 Microcontroller Series Overview – STM32U5 Series Overview (ST official page)

6. Spyrosoft – "Top 5 Embedded HMI UI Frameworks for 2025" – Spyrosoft Tech Blog article

7. CNX Software – "Qt for MCUs – Qt Announces Support for Microcontrollers" – CNX Software article

8. Bivocom – "Differences and Advantages Between Qt and LVGL" – Bivocom industry article

9. Somco Software – "LVGL GUI Development: A Comprehensive Overview" – Somco Software blog post

10. Qt Company – "Safe and Effective: Functional Safety and Qt" (Whitepaper) – Qt Functional Safety Whitepaper page

11. LVGL Forum – "Can LVGL be used for safety-critical systems?" (Discussion Thread) – LVGL Forum thread

12. LinkedIn – "LVGL Safe: A New UI Library for Safety-Critical Products" (Announcement post) – LinkedIn announcement by LVGL

13. Qt Company – Cybersecurity and Qt (Cyber Resilience Act readiness) – Qt Cyber Resilience Act resource page

14. The Armchair Trader – "Qt Group: an expensive Finnish fintech, but for a good reason" – The Armchair Trader analysis

15. Qt Company – Qt Pricing and Licensing – Qt Pricing page

16. Reddit r/QtFramework – "Do companies consider Qt for new applications?" (Community discussion on Qt adoption) – Reddit discussion thread

17. GitHub – LVGL project repository (Community stats and contributions) – LVGL GitHub repository

18. Qt Quality Assurance – "Squish for MCUs" (Product page) – Squish for MCUs product page

19. LVGL Forum – "Automated testing for LVGL" (Guidelines for testing LVGL UIs) – LVGL Forum discussion

20. Qt Company – Coco, a modern code coverage tool – Coco Product Page

21. Qt Company – Qt Success Stories from Customers – Qt Success Stories

About Spyrosoft

Spyrosoft is a group of companies delivering consulting and bespoke software development across industries ranging from automotive and finance to healthcare & life sciences, Industry 4.0, and robotics. Its “one-stop shop” model brings together embedded engineering, product design, QA, cloud, and cybersecurity—so clients can build complete, integrated products rather than stitching together isolated components. With a team of nearly 2,000 people, Spyrosoft operates across multiple delivery locations, including Poland, the UK, Germany, Denmark, Croatia, Romania, India, Argentina, and the United States.

In embedded and HMI programs, Spyrosoft supports end-to-end delivery—from UX/UI design and prototyping, through proof-of-concept work, to implementation on real devices and production deployment. This includes building touch-driven interfaces and connected experiences, integrating the HMI with underlying platform software, and validating the whole system through disciplined QA and integration testing. The focus is on reducing friction between design and engineering while keeping the interface aligned with usability, performance, and hardware constraints.

Learn more at www.spyro-soft.com

About The Author

Mikalai Arapau is a Software Architect and Lead HMI Engineer with 20+ years of experience across embedded, desktop, and real-time systems. He pairs architecture ownership with hands-on engineering, building production HMIs and the platform software that makes them robust on real hardware. His work spans automotive, industrial, media, and UAV domains—from early bring-up and integration through system design and implementation. *mikalai.arapau@gmail.com*