COSIMATEAI INFRASTRUCTURE DOCUMENTATION

Technical Note 1

Subject:	Graphical Interface Refactoring & Modernization	**Date:**	May 2026
Core Stack:	Native C++ Integration	**Reference:**	www.cosimate.com/CosiMateAI

1. Abstract & System Context

This technical note details the architectural migration of the CosiMate Graphical User Interface (GUI) from a legacy **WxWidgets** framework to a modernized **Qt** environment. Because CosiMate's core application logic and heavy simulation-synchronization kernels are natively implemented in **C++**, migrating directly to a native C++ UI framework enables compile-time linking, monolithic binary optimizations, and zero-overhead memory layout sharing. To eliminate weeks of manual refactoring, we successfully structured an automated, AI-assisted engineering pipeline utilizing **Google Gemini** to parse legacy layout matrices and generate fully compliant Qt C++ source code compatible across Windows and Linux deployment targets.

2. Technical Overhead of the Legacy WxWidgets Framework

Historically, CosiMate utilized the WxWidgets framework to deliver desktop layouts. While functionally stable over previous development cycles, this framework incurred distinct engineering liabilities as the platform scaled:

- **Native Subsystem Fragmentation:** WxWidgets acts as a wrapper that maps controls directly onto the host operating system's native API (e.g., Win32 API on Windows, GTK on Linux). This strict dependence introduced layout discrepancies, distinct font rendering metrics, and disparate window-drawing behaviors across host OS targets, requiring highly customized, target-specific adjustments in the integration suite.
- **High Maintenance Latency:** The declarative layout scripts and structural window hierarchies within the legacy interface were old-fashioned and complex to configure, resulting in high developer friction and substantial code maintenance cycles when adding new simulation telemetry panes.
- **Cross-Platform Porting Inefficiencies:** Porting, compiling, and mapping native configurations onto newer operating systems required intensive manual engineering cycles due to changing underlying system libraries.

3. Target Framework Architecture: Direct C++ to Qt Integration

The **Qt** framework was chosen to replace WxWidgets due to its modern architecture and native rendering capabilities. Unlike WxWidgets, Qt relies on its own high-performance, internal vector graphics engine to directly paint controls, guaranteeing strict pixel-perfect visual and behavioral uniformity across Windows and Linux environments.

CosiMate Core C++ Engine (Simulation Kernels & Telemetry Loops)	↔	Modern Qt C++ Graphical Layer (Signals & Slots Engine / Direct Layouts)

By keeping the entire framework native to C++, we avoid foreign-language wrappers, marshalling costs, or complex binding plumbing. Communication between the simulation engine threads and the GUI render loop occurs with maximum efficiency via Qt's asynchronous **Signals and Slots** mechanism. Queued connections enable safe multi-threaded message passing without locking the core simulation runtime.

4. Multimodal AI-Assisted Generation Pipeline

Migrating dense component codebases manually represents an expensive, multi-week undertaking prone to positioning bugs and layout breakage. To achieve rapid turnaround, our engineers configured an automated generation pipeline using the **Google Gemini** large language model. This process moved from raw inputs to optimized Qt source files over a three-stage pipeline:

4.1 Layout Deconstruction & Computer Vision Analysis

The initial phase utilized Gemini’s vision capabilities. Engineers fed visual captures and schematic screenshots of the existing WxWidgets layout directly into the model. The model dissected the legacy interface, recognizing layout boundaries, grouping parameters, explicit multi-pane horizontal/vertical splitters, data telemetry grids, and localized control triggers.

4.2 Context Enrichment via RAG and Personal Data Environment (PDE) Protection

To bypass generic output and ground the model in production requirements, developers spent two hours structuring prompt data using a secure **Retrieval-Augmented Generation (RAG)** pipeline within our local **Personal Data Environment (PDE)**.

- **Secure Localized RAG:** To preserve proprietary trade secrets and comply with strict data-governance standards, our corporate source code repositories were isolated inside a dedicated PDE. The RAG architecture indexed only the specific C++ base header layouts (`.h`), telemetry hooks, and window-management interfaces.
- **Context Optimization:** Relevant technical snippets were vector-searched and safely injected into the model's context window as prompt extensions. By mapping this localized context alongside code examples within the secure boundaries of our PDE, Gemini was granted explicit alignment guidelines regarding our expected code architecture without risking exposure of our foundational simulation logic to public environments.

4.3 Deterministic Code Generation

With visual and architectural specifications mapped, Gemini generated structured, fully formed C++ code elements adhering to Qt layout schemas. The code correctly replaced legacy WxWidgets positioning containers with modern Qt layout classes (such as `QVBoxLayout`, `QHBoxLayout`, and `QSplitter`), maintaining full alignment with our backend event hooks.

> **Technical Invariant Maintained:** The generation pipeline was strictly constrained to isolate the UI viewport definition layer. This structural isolation guaranteed that CosiMate's underlying native simulation synchronization kernel remained entirely untouched and free from regression risks.

5. Build Verification and Operational Results

The generated Qt components were run through automated compiler sanity tests and deployed into testing environments with minimal manual adjustments. The results demonstrate significant improvements across key engineering metrics:

- **Absolute Cross-Platform Uniformity:** The generated Qt architecture compiled natively for both `x86_64` Windows (MSVC) and Linux (GCC) platforms out-of-the-box, ensuring exact cross-platform feature parity and eliminating OS-specific graphics workarounds.
- **Drastic Minimization of Cycle Time:** The end-to-end modernization cycle—including specification writing, AI-assisted code emission, source review, and compilation integration—was fully executed **within a single business day**.
- **Enhanced Maintainability:** The resulting UI layer uses clean object models, lowering maintenance overhead and enabling rapid implementation of future dashboard extensions.