Rust for MedTech:
A Language for Biomedical Engineering

Aditya Chaudhry

July 22, 2025

Abstract

The global shortage of transplantable organs has exposed critical limitations in our healthcare infrastructure, biomedical capabilities, and technological readiness. As a biomedical engineer committed to solving what I define as *The Organ Donor Problem*, this paper presents a new perspective: organ failure is not merely a clinical crisis, but a systems engineering challenge.

In this context, the Rust programming language emerges as a foundational tool for the future of medical technology. Designed with memory safety, concurrency, and performance at its core, Rust provides a powerful alternative to legacy languages like C and C++. This white paper explores how Rust can be used across multiple layers of biomedical innovation—from embedded firmware for implantable devices, to surgical robotics, artificial organs, simulation tools, and secure backend infrastructure.

The aim is not to advocate Rust for hype's sake, but to demonstrate how its unique strengths align with the sensitive, secretive, and safety-critical nature of MedTech. As we move toward an integrated ecosystem of artificial organs and intelligent surgical systems, Rust enables a future where software is not a liability, but a life-saving asset.

1 Introduction

Rust is a systems programming language built for performance, safety, and concurrency. Originally developed at Mozilla and released in 2015, Rust was designed to overcome the long-standing tradeoffs in low-level development—namely, the choice between speed and safety. It offers modern language features and tooling while maintaining control over hardware-level behavior, making it a compelling alternative to C and C++ for critical systems.

At the heart of Rust's design is its ownership model, which enforces memory safety at compile time without requiring a garbage collector. This model prevents entire classes of bugs—such as null pointer dereferencing, buffer overflows, and use-after-free errors—that are common in C/C++ and often catastrophic in medical applications. Rust also provides fearless concurrency, enabling developers to write multithreaded software without risking data races or undefined behavior.

For decades, C and C++ have been the de facto choices for embedded systems, surgical control software, and medical instrumentation. Yet these languages come with sharp edges that require constant vigilance and defensive programming. In safety-critical domains like MedTech, where software bugs can lead to device recalls or patient harm, such fragility is no longer acceptable.

This paper proposes that Rust is not only a viable replacement for traditional systems languages in medical technology—it is the language best suited for the next generation of biomedical systems. As we build artificial organs,semi-autonomous surgical systems, and intelligent diagnostic tools, the reliability and security of their software infrastructure will determine their success. Rust brings the rigor of formal methods, the efficiency of bare-metal programming, and the developer experience of modern software engineering—all in a package designed for scalability and safety.

In the chapters that follow, we will explore how Rust applies to secure medical device development, surgical robotics, artificial organ systems, toolchain innovation, and healthcare infrastructure. Each section presents a conceptual case for Rust as the backbone of a MedTech ecosystem aimed at solving the organ donor crisis—one line of safe, reliable code at a time.

2 Why Rust Is Poised to Transform MedTech

Medical technology systems operate under an uncompromising demand: software must not fail. Whether it controls an artificial heart, a surgical robot, or a biosignal monitor, the margin for error is effectively zero. Yet traditional development practices—especially those reliant on C or C++—continue to expose systems to memory corruption, concurrency bugs, and undefined behavior. In such an environment, Rust's core features are not just advantageous—they are transformative.

Memory Safety Without Garbage Collection

Rust's ownership and borrowing system enforces strict memory safety guarantees at compile time. This means that common issues like buffer overflows, null pointer dereferencing, double frees, and use-after-free bugs are virtually eliminated in safe Rust code. Unlike garbage-collected languages, Rust achieves this without introducing runtime overhead or unpredictable pauses—an essential requirement for embedded systems and real-time medical devices. This makes Rust particularly well-suited for life-sustaining applications where timing and determinism are critical.

Performance on Par with C/C++

Rust offers low-level control over memory and CPU resources, allowing developers to write code that performs as efficiently as C or C++. With zero-cost abstractions and advanced compiler optimizations through LLVM, Rust enables developers to build software that meets the performance needs of intensive biomedical computations, device control loops, and high-frequency data acquisition without sacrificing safety.

Fearless Concurrency

Concurrency is increasingly important in medical software, from real-time data analysis to multithreaded robotic control systems. Rust's type system statically prevents data races and enforces safe access patterns across threads. This makes it possible to write concurrent code that is both efficient and correct—without the uncertainty and fragility commonly associated with C/C++ thread management.

Security and Reliability by Design

In MedTech, security is not optional—it is mandated by regulation and demanded by ethics. Rust's memory-safe foundations dramatically reduce the attack surface for malicious exploits. Furthermore, the language encourages rigorous error handling and exhaustive case coverage, which minimizes undefined behavior and unexpected states. This aligns with safety standards like IEC 62304 for medical device software, making Rust a strong candidate for regulatory-grade development.

Developer Ergonomics and Tooling

Rust pairs its low-level power with modern tooling: a package manager (Cargo), a robust compiler with helpful diagnostics, and a growing ecosystem of libraries (crates) for embedded, web, and scientific computing. Developers can write, test, and maintain complex biomedical codebases with greater confidence and less overhead. In a field where both innovation and reliability are essential, Rust offers a productivity model that scales.

A Perfect Fit for MedTech Innovation

Taken together, Rust's properties—safety, speed, concurrency, and clarity—make it uniquely suited to the challenges of MedTech. As biomedical engineers push the boundaries of what is surgically, electronically, and biologically possible, the software underlying these systems must evolve accordingly. Rust offers the foundation for that evolution.

3 Secure Medical Device Engineering with Rust

Medical devices, especially implantables and wearables, form the backbone of modern patient care and organ replacement technologies. From pacemakers and insulin pumps to artificial kidneys and biosensor-equipped monitors, these devices operate under extreme constraints: limited memory, battery life, real-time responsiveness, and—most importantly—zero tolerance for software failure. Rust is uniquely equipped to meet these challenges across the entire embedded software stack.

Memory-Safe Embedded Firmware

Traditional device firmware is typically written in C or C++, where even small oversights can result in catastrophic memory issues. Rust eliminates entire classes of bugs at compile time, including buffer overflows, dangling pointers, and stack corruption. This is especially critical in firmware that runs without an operating system (bare metal) or with tight timing constraints.

Rust's `no_std` support allows developers to write efficient, allocation-free code for microcontrollers. Combined with `embedded-hal` (Hardware Abstraction Layer) traits, engineers can build drivers for sensors, actuators, and radios with guaranteed memory safety and deterministic performance.

Real-Time Control with RTIC and Interrupt Safety

Medical devices often rely on real-time control loops triggered by hardware interrupts. Rust's interrupt-driven concurrency models, such as the Real-Time Interrupt-driven Concurrency (RTIC) framework, allow for deterministic scheduling of tasks with compile-time guarantees of safety. RTIC ensures that shared data is accessed in race-free ways, preventing unpredictable behavior during concurrent reads and writes—a common source of instability in C-based embedded systems.

This is particularly beneficial in devices like:

- Artificial pancreas systems responding to glucose sensor input.
- Ventricular assist devices adjusting pump speed dynamically.
- Wearable ECGs processing biosignal data in real-time.

Long-Term Reliability in Implantables

Implantable devices must often operate for years without failure. Rust's absence of a garbage collector means it avoids unpredictable pauses or memory leaks, which are unacceptable in continuously running firmware. The language's strict compile-time checks and predictable runtime behavior make it possible to write code that is both low-power and fault-tolerant.

In long-lifespan systems like pacemakers or neurostimulators, this reliability translates directly into fewer field failures, longer device life, and improved patient outcomes.

Secure-by-Default Embedded Systems

Many modern medical devices are connected—either wirelessly or through hospital networks—making them vulnerable to attacks. Rust's compile-time safety model significantly reduces the surface area for exploits like buffer overflows, which account for a large percentage of embedded system vulnerabilities.

In the context of cybersecurity for medical technology, Rust allows developers to:

- Avoid memory corruption exploits by construction.
- Ensure strict typing and exhaustive error handling.
- Create firmware that is easier to verify and audit for regulatory compliance.

Towards a New Embedded Standard in MedTech

With Rust's embedded ecosystem growing rapidly—including projects like `probe-rs`, `defmt`, `RTIC`, and target support for ARM Cortex-M devices—biomedical developers now have access to modern, safe tooling for critical applications.

Rust enables a future where medical device firmware is not only faster and more reliable, but also easier to develop, maintain, and certify. For engineers building the next generation of therapeutic implants and biosignal-driven diagnostics, Rust is the clear step forward in embedded system design.

4 Rust for Surgical Robotics

Surgical robotics represents one of the most technically demanding fields in medical engineering. Precision, timing, and safety are paramount when designing systems that assist, augment, or even autonomously perform surgical procedures. These machines operate at the intersection of software and physical control, often under high concurrency, in real time, and within safety-critical contexts. Rust, with its emphasis on memory safety and concurrency correctness, provides an ideal language for developing surgical robotic systems that are both powerful and trustworthy.

Real-Time Precision Control

Surgical robots must coordinate multiple hardware components simultaneously—actuators, sensors, force-feedback systems, and vision modules—while ensuring millisecond-level response time. Rust's low-level control and zero-cost abstractions enable developers to write deterministic, efficient control loops without sacrificing code clarity or safety.

Furthermore, Rust avoids the unpredictable pauses associated with garbage collection. This makes it highly suitable for control systems where timing consistency is critical, such as:

- Microsurgery involving vascular or neural manipulation.
- Robotic suturing and anastomosis.
- Minimally invasive laparoscopy and catheter-based procedures.

Fearless Concurrency in Cyber-Physical Systems

Modern surgical robots rely heavily on concurrent task execution—processing camera feeds, sensor data, and user input while controlling multiple robotic limbs. Rust's concurrency model eliminates data races at compile time, enabling multi-threaded operations to be written with confidence.

This allows systems to handle:

- Parallel sensor fusion for motion compensation and stabilization.
- Multitasking of haptic feedback and tool positioning.
- Real-time teleoperation with latency-aware thread prioritization.

Where C++ demands fragile synchronization mechanisms and extensive testing to avoid race conditions, Rust enforces thread safety structurally.

Safety-Critical Autonomy and Assisted Surgery

As surgical robots move toward partial autonomy—assisting with routine sutures, vessel clipping, or tool positioning—software must behave in deterministic and verifiable ways. Rust's strict typing and exhaustive pattern matching help enforce invariant conditions and fail-safe logic. This is crucial in preventing undefined states that could cause unexpected or dangerous behavior in autonomous or semi-autonomous operation.

In addition, developers can express safety contracts clearly in Rust through type systems, 'Result'/'Option' handling, and hardware abstraction layers. This clarity helps in passing medical software audits and fulfilling standards such as IEC 62304 or ISO 13485.

Remote Surgery and Teleoperation

With advancements in high-speed networks and robotic miniaturization, teleoperated surgery is becoming viable in rural, battlefield, or disaster-zone applications. Rust is well-positioned to support such architectures due to:

- Its reliable multithreaded networking capabilities via async runtimes like `tokio`.
- Secure transport layers and encryption libraries with memory safety guarantees.
- Consistent and low-latency task scheduling in control software.

These attributes make Rust an excellent language for both the local control firmware of the robot and the remote command-and-control infrastructure.

A Platform for Next-Generation Surgical Systems

From compact surgical assistance robots to fully immersive telerobotic platforms, Rust offers a coherent and rigorous foundation to build robotic systems that surgeons can trust. It enforces software discipline where human lives are on the line—and enables innovation without compromising safety or performance.

As surgical robotics evolves toward smaller, smarter, and more autonomous systems, Rust is well positioned to become the language of choice for their control software, internal tooling, and secure integration with broader medical networks.

5 Rust for Artificial Organs

Artificial organs represent one of the most ambitious frontiers in biomedical engineering. These devices are designed to replace or replicate the function of natural organs—often with embedded control systems, real-time feedback, and continuous monitoring. Whether fully mechanical or biohybrid, artificial organs demand software that is stable, secure, and adaptive over long durations. Rust, with its focus on safety and correctness, is an excellent match for such life-sustaining systems.

Embedded Intelligence in Organ Systems

Artificial organs such as total artificial hearts, implantable lungs, and pancreas systems require embedded controllers that regulate physiological processes in response to sensor feedback. These systems operate in real-time and must guarantee deterministic performance.

Rust's ability to run in `no_std` environments on microcontrollers allows developers to build firmware that:

- Responds to biosensors with microsecond precision.
- Maintains accurate control over flow rates, pressure, or insulin delivery.
- Handles failure conditions gracefully without risking patient safety.

Using safe concurrency and predictable timing, Rust ensures that control software runs efficiently and securely—without memory leaks or undefined behavior that could jeopardize a patient's life.

Real-Time Feedback and Closed-Loop Control

Many artificial organs depend on closed-loop systems, where sensor readings dynamically influence actuator output. For instance:

- An artificial heart adjusting stroke volume based on patient activity.
- A synthetic kidney regulating filtration rate via embedded pressure sensors.
- A neurostimulator modulating brain signals in response to EEG patterns.

Rust allows developers to build such feedback loops with precise state management and thread safety. Its 'match'-based control flow, strong type system, and error handling via 'Result' and 'Option' enable highly readable, reliable logic—critical for debugging and certification.

Long-Term Autonomy and Energy Efficiency

Implanted artificial organs must run autonomously for months or years. Software written in Rust benefits from:

- Predictable memory behavior without garbage collection.
- Compile-time elimination of resource leaks.
- Highly optimized binary size and energy efficiency.

These characteristics are invaluable for low-power operation inside the human body, where thermal and battery constraints are severe. Rust enables embedded developers to maximize functionality without compromising reliability.

Diagnostics, Safety Checks, and Self-Monitoring

Rust's design promotes rigorous error handling and system introspection. Artificial organs can incorporate:

- Built-in diagnostics routines and state validation.
- Watchdog systems with Rust's panic safety mechanisms.
- Redundant fallback modes using exhaustively matched failure states.

This reduces the risk of silent faults and increases the chance of recovery in critical scenarios. Medical teams can trust that device behavior remains consistent—even under stress.

Paving the Way Toward Synthetic Physiology

As we advance toward scalable, patient-specific artificial organs—powered by biosensors, actuators, and biocompatible electronics—software will be a defining element of function and safety. Rust offers a development model where complex behavior can be encoded without sacrificing verifiability.

By using Rust in the core firmware and control logic of artificial organs, biomedical engineers can design next-generation organ systems that are:

- Safer by default.
- Easier to test and certify.
- Ready for global, life-saving deployment.

Rust empowers us to move beyond experimental prototypes and into a future where artificial organs become reliable, programmable components of modern medicine.

6 Rust to Develop Tools and Infrastructure for MedTech

Beyond powering embedded firmware and robotic control systems, Rust also excels in building the software ecosystem that supports innovation in medical technology. From diagnostic tools and design software to secure cloud services and remote interfaces, Rust's versatility makes it suitable for developing the entire stack. This includes both the tools engineers use to build devices, and the infrastructure hospitals and clinicians use to operate them.

Command-Line and Diagnostic Tools

Rust is widely recognized for its ability to create fast, safe, and portable command-line applications. For biomedical engineers, this enables development of custom tools for:

- Testing and calibrating device firmware.
- Running simulations of artificial organ behavior.
- Processing physiological datasets and biosensor logs.

Using crates like `clap` for argument parsing and `serde` for data serialization, developers can quickly build cross-platform utilities that are efficient and robust. Unlike Python or JavaScript tools, Rust CLIs have minimal dependencies, low memory usage, and no runtime surprises—making them ideal for regulatory-compliant workflows and long-term reproducibility.

Graphical Interfaces for Clinicians and Engineers

User-facing graphical tools are essential in MedTech, whether for configuring devices, guiding robotic procedures, or visualizing diagnostics. Rust now supports modern GUI development through frameworks such as:

- `Iced` – for desktop and embedded applications.

- `egui` – for interactive scientific visualization and quick prototyping.
- `Slint` – for professional-grade UIs with a declarative design model.

These frameworks allow biomedical developers to build intuitive, high-performance interfaces without compromising on reliability or cross-platform support. For example, a surgical robotics control panel or an artificial kidney dashboard could be built entirely in Rust—ensuring consistency, low latency, and memory safety from UI to hardware.

Simulation, Modeling, and EDA Workflows

Rust's performance and safety make it an excellent language for simulation-heavy workflows in biomedical engineering. Engineers developing artificial organs or bioelectronic circuits can use Rust to:

- Model hemodynamics or organ-specific fluid flow.
- Simulate electrical behavior of implantable systems.
- Develop EDA tools for designing custom PCBs and controllers.

Rust's support for parallel computation, precision numerics, and deterministic memory usage helps ensure simulation accuracy and reproducibility—key requirements in regulated medical research.

Secure and Scalable Healthcare IT Systems

Connected medical systems require secure, high-performance cloud infrastructure for data aggregation, device management, and clinical analytics. Rust is increasingly used to build these backend services thanks to:

- Web frameworks like `Actix Web` and `Axum` for RESTful APIs.
- Asynchronous runtimes like `Tokio` for scalable I/O.
- Strong typing and memory safety for secure, crash-free services.

These tools can support:

- Telemetry ingestion from organ-support devices in real-time.
- Encrypted firmware delivery and device updates.
- Remote diagnostics, alerts, and AI-powered decision support.

Rust reduces the risk of memory vulnerabilities, buffer overflows, and race conditions—making it easier to achieve HIPAA and IEC 62304 compliance.

A Unified Toolchain for Biomedical Innovation

By using Rust for both development tools and infrastructure, biomedical engineers can:

- Maintain a consistent, high-integrity codebase across systems.
- Share libraries and models between simulation, firmware, and analytics.
- Increase developer efficiency and reduce long-term maintenance costs.

Whether building a CLI to analyze biosensor data, a GUI to program an artificial organ, or a backend to manage a fleet of devices, Rust provides the performance, correctness, and reliability needed at every layer of the MedTech stack.

7 Conclusion

The organ donor crisis is not simply a medical issue—it is a grand engineering challenge. Addressing it will require new paradigms in how we build, control, and deploy biomedical technology. At the heart of this transformation lies the need for software that is fast, safe, secure, and maintainable across embedded systems, robotics, artificial organs, and digital health infrastructure.

Rust represents a turning point in how we approach this challenge. As a language, it eliminates entire categories of bugs before code is ever run. As a platform, it empowers engineers to write high-performance systems software with modern development ergonomics. And as a movement, it encourages rigorous thinking, strong typing, and safety by default—qualities that align perfectly with the demands of medical innovation.

In this white paper, we have examined how Rust can power the full spectrum of MedTech—from real-time device firmware and robotic control loops to diagnostics tools and secure backend services. We have shown that Rust is not only capable of supporting the technical requirements of artificial organs and surgical robotics, but also uniquely suited to the sensitive, secretive, and safety-critical nature of biomedical engineering.

As a biomedical engineer focused on solving the Organ Donor Problem, I view Rust as more than a language—it is a strategic enabler. It allows us to design systems we can trust, at scales that matter, with confidence in their correctness and safety.

The future of MedTech will not be built with brittle code and legacy assumptions. It will be engineered—with Rust.