

A Pragmatic Guide to Introducing Rust into C/C++ Codebases with Minimal Risk

Table of Contents

Why is Rust the new standard for backends?

3

5 key benefits of Rust adoption

6

3 myths about Rust

8

Step-by-step C/C++ to Rust adoption framework

10

Case in point: How Yalantis helped adopt Rust in a legacy C++ system

12

Wrap-up

14

About Yalantis

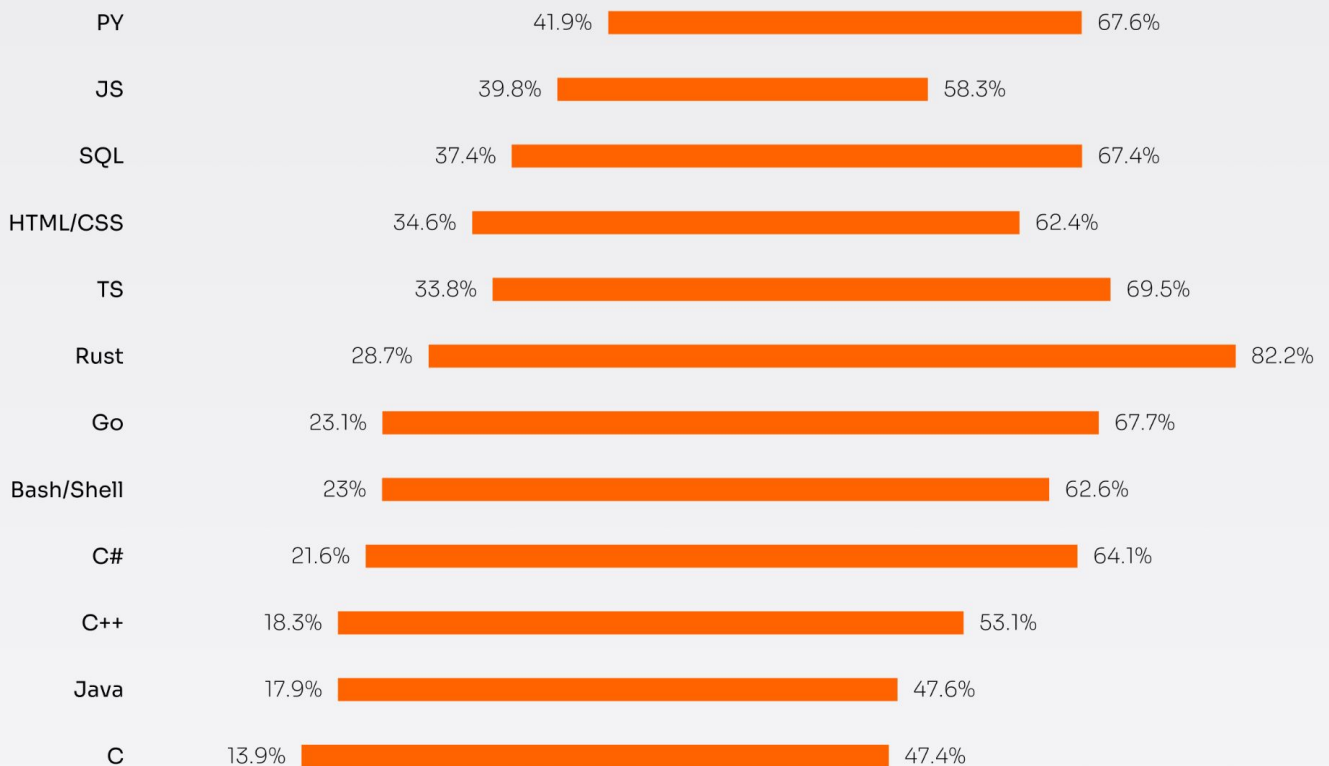
15

Why is Rust the new standard for backends?

For years, C and C++ were the go-to for high-performance systems. But things are shifting. More and more companies are choosing Rust, a language that gives you the **speed of C++ without the constant worry about memory bugs**, plus modern tools and a much smoother developer experience.

And it's not just hype. Rust has been voted the **most loved programming language** nine years in a row in the [Stack Overflow Developer Survey](#).

82.2% of developers say Rust is their most-admired language this year.



Here are the main aspects that make Rust a go-to:

Safe concurrency

Thread safety is enforced by the compiler, making scaling multi-threaded backends much less risky.

Security by design

Rust eliminates whole classes of memory bugs (buffer overflows, use-after-free).

Stable performance

No garbage collection, fewer runtime surprises, lower infrastructure costs.

It's not just developers driving the shift, governments are now weighing in too.

Interesting fact: The U.S. government recommended eliminate C/C++ in security-critical systems

Around 70% of software vulnerabilities in systems like Windows and Chrome stem from memory-unsafe languages such as C and C++. That's why the [U.S. Cybersecurity and Infrastructure Security Agency \(CISA\)](#), together with the FBI, has taken its strongest stance yet:

By January 1, 2026, software manufacturers handling critical infrastructure must publish a **memory safety roadmap**. Developing new critical systems in C or C++ when memory-safe alternatives like Rust exist will be considered a “dangerous” and negligent practice.

Rust advantages have a visible impact on the development process. **Here's what companies adopting Rust report:**

+30%

Faster development speed

-50%

Fewer bugs reach production

-60%

Lower vulnerability count

Here are the main Rust benefits that explain such results (→)

5 key benefits of Rust adoption

Legacy stacks slow teams down, but Rust makes a difference.

Security risks and downtime

1. Minimal vulnerabilities

C and C++ lack built-in memory safety, and 60–70% of critical security bugs come from these issues. Microsoft has acknowledged that roughly 70% of Windows security vulnerabilities are caused by memory safety issues. Rust enforces memory safety at compile time, preventing buffer overflows and use-after-free errors before they even run.

Concurrency bugs

2. Reliable, crash-free code

Multithreaded legacy code often causes data races and deadlocks that are nearly impossible to debug. Rust's compiler enforces thread safety at the type level, eliminating entire classes of concurrency bugs. This makes large-scale systems (think databases, backends, OS kernels) far more reliable under heavy load.

Slow feature delivery

3. 30%+ faster releases

Legacy C/C++ stacks are tightly coupled, making every change risky. Rust's modular design and strict compiler checks allow safe refactoring and cleaner APIs. Companies report 30–40% faster release cycles thanks to fewer test-fix-retest loops and safer upgrades.

Complex build systems

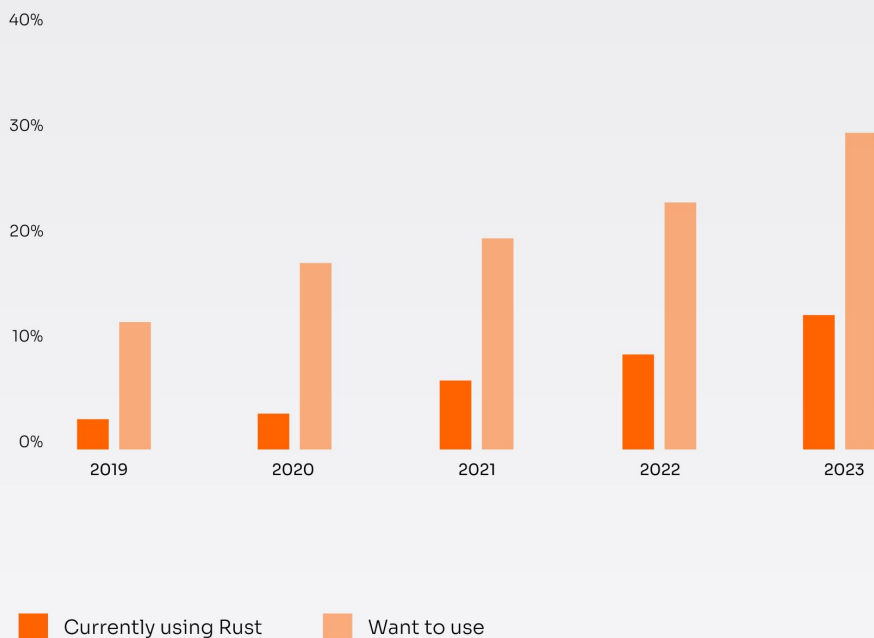
4. Smooth, predictable integrations

C/C++ builds are notoriously fragile, relying on custom scripts and dependency jungles. Rust ships with Cargo, a built-in package manager that guarantees reproducible builds and smooth dependency handling. This lowers integration friction across CI/CD pipelines and cloud environments.

Legacy code experts

5. Future-proof teams

It's true: Rust developers are fewer than C++ veterans. But those who do know Rust are often early adopters, open to modern practices like DevSecOps and cloud-native tooling. Rust's community has grown 5× in the last 5 years, making hiring easier every year.



For scaling teams, this means smaller but more forward-thinking talent pools that push projects toward long-term maintainability.

But companies often postpone Rust adoption due to several concerns that aren't true in reality.

3 myths about Rust

Rust can be integrated safely into large, complex systems without downtime or full rewrites, and your team can rest the same.

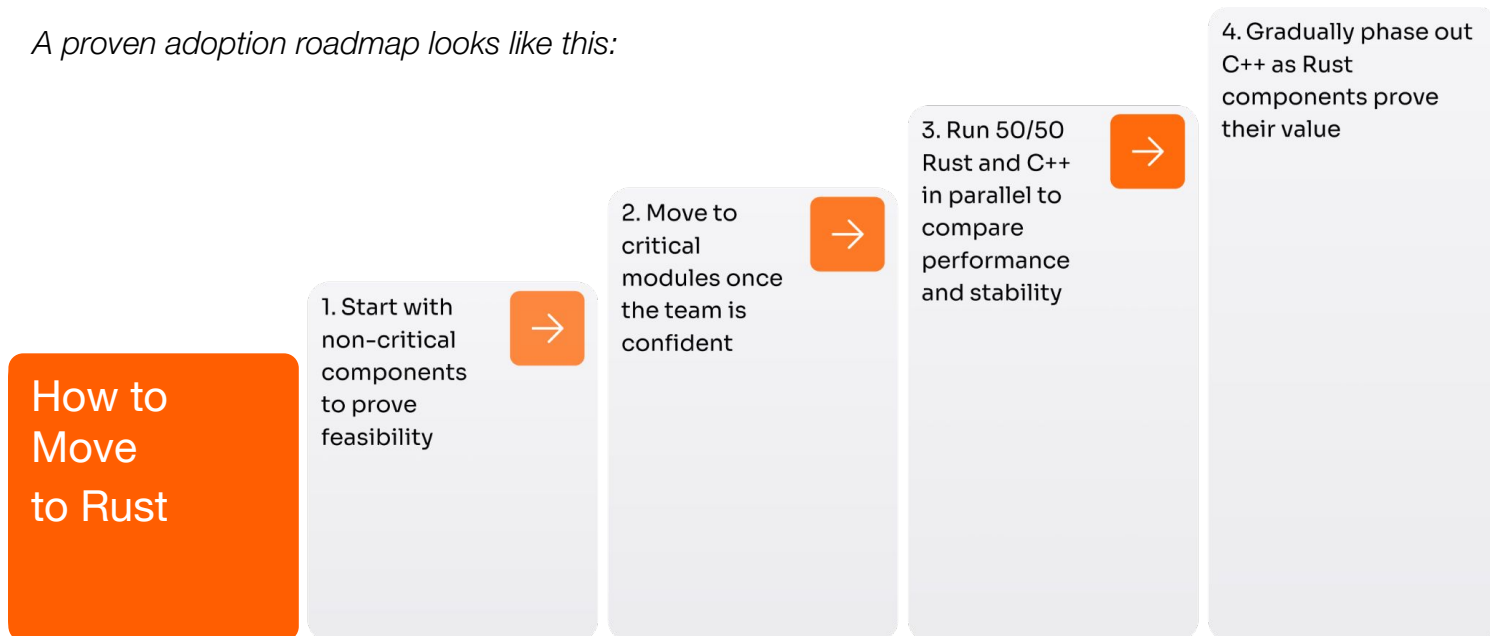
Myth #1: “We’d have to rewrite everything”

This is the biggest misconception. Adopting Rust doesn’t mean throwing away your existing codebase. Instead, teams can start small: choose one non-critical module, rebuild it in Rust, and test how it interacts with the current system. From there, adoption can expand gradually. Many companies run mixed systems where C++ and Rust coexist — no massive rewrite required.

Myth #2: “Adopting Rust will disrupt delivery or introduce downtime”

Rust is designed to be integrated step by step.

A proven adoption roadmap looks like this:



Because Rust is fully compatible with C and C++, new modules can run alongside the old system. This ensures continuous delivery with **no downtime** and minimal disruption to ongoing development.

Myth #3: “We’ll have to rebuild our team entirely”

Another common concern is staffing. In reality, you don’t need a new team — you need a learning roadmap. Typically, senior engineers start first, improving their tech stack with Rust while still maintaining the C++ system. With the right mentorship, the same team can transition smoothly, continuing to support legacy code while introducing Rust. Over time, more engineers pick it up, and the team naturally shifts to Rust without hiring a whole new workforce.

Here’s a step-by-step guide to Rust adoption [→](#)

Step-by-step C/C++ to Rust adoption framework

Adopting Rust doesn't require rewriting your entire C/C++ codebase. Instead, follow a structured, low-risk approach: start small, prove value, and scale gradually.

Step 0: Assessment & Planning (2 weeks)

Start by taking a close look at your codebase. Pick non-critical but high-value modules that could benefit most from Rust. Gather metrics like performance, memory usage, and error rates to understand where the pain points are. Define what success looks like and set realistic expectations with your team. This step is about knowing where Rust can make a difference without putting existing operations at risk.

Step 1: Feasibility (2–12 weeks)

Next, test the idea with a pilot project. Bring together a small cross-functional team and develop a single Rust module alongside your C++ system. Benchmark its performance, check how well it integrates, and evaluate maintainability. By the end of this phase, you'll know whether Rust can coexist with your legacy code and deliver real improvements.

Step 2: Proof of Concept (1–2 months)

With feasibility confirmed, move to a working proof of concept. Build the Rust component in a production-like environment, design APIs, run tests, and integrate it with your build system. This step lets the team see Rust in action. It's also the perfect opportunity to gather feedback, fine-tune integration practices, and show stakeholders the tangible benefits of Rust before committing further.

Step 3: Development & Integration (3–6 months)

Once the PoC succeeds, it's time to start expanding Rust. Teams can be split roughly 50/50 between Rust and C++ so legacy features continue running smoothly. New features are developed in Rust, integrated gradually with the C++ system, and monitored through dashboards for stability and performance. At the same time, knowledge is shared across the team so everyone gains confidence with Rust. This approach lets you scale adoption without disrupting ongoing operations.

Step 4: Wider adoption (6–18 months)

Finally, Rust becomes the main language in the system. New modules are written in Rust, while legacy C++ is kept only where necessary. Continuous practices like mentorship, CI/CD, security scans, fuzzing, and code reviews ensure quality and maintainability. Over time, your system is more secure, more stable, and easier to evolve, while the team has grown confident in Rust without facing downtime.

Based on Yalantis' experience of helping businesses move to Rust, here's the roadmap you can use.

Language	Memory Safety	Concurrency	IoT Suitability
Step 0: Assessment & Planning	2 weeks	Identify modules suitable for Rust	<ul style="list-style-type: none">→ List non-critical, high-value modules→ Collect baseline metrics
Step 1: Feasibility	2-12 weeks	Prove Rust works with your system	<ul style="list-style-type: none">→ Integrate a small Rust module with C++→ Generate benchmark report and integration plan
Step 2: Proof of Concept	1-2 months	Deliver a working Rust component	<ul style="list-style-type: none">→ Design APIs, test, and integrate builds→ Deploy PoC in production-like environment
Step 3: Development & Integration	3-6 months	Incrementally replace non-critical modules	<ul style="list-style-type: none">→ Split teams: ~50% Rust, ~50% C++→ Develop new features in Rust→ Use monitoring dashboards to track performance and stability→ Transfer Rust knowledge to the broader team
Step 4: Wider adoption	6-18 months	Expand Rust, gradually phase out C++	<ul style="list-style-type: none">→ Shift to Rust-majority team, keeping C++ only for legacy support→ Implement continuous integration, security scans, and mentorship→ Refine CI/CD pipelines, quality gates, and fuzzing

Case in point: How Yalantis helped adopt Rust in a legacy C++ system

Our client was struggling with a legacy C++ system. Frequent bugs, security vulnerabilities, and slow feature delivery were holding the team back. They wanted to start developing new functionality in Rust, but were understandably cautious about disrupting ongoing operations. Yalantis proposed a low-risk approach.

Here's what we did:

- Assembled a cross-functional team: Rust engineer, Systems architect, Analyst, PM, and QA lead
- Conducted a 2 month feasibility study
- Selected a single module as a pilot
- Integrated Rust alongside the existing C++ code
- Tested compatibility, performance, and maintainability of the pilot module

The Rust component ran smoothly within the C++ system. Following the pilot, the client's team **gradually expanded Rust adoption**. With Yalantis mentorship, senior engineers **led the transition**, while the rest of the team continued maintaining the legacy C++ code. Over time, more modules were rewritten in Rust without downtime or disruption.

Here are the results our client achieved right after:

+45%

Faster feature delivery

-60%

Reduction in bugs

+30%

More secure system operations

✦ Team upskilled

in Rust while maintaining C++ expertise

Wrap-up

Adopting Rust doesn't have to be risky or disruptive. If you follow a structured, incremental approach, teams can modernize their systems, improve security, and accelerate development without rewriting legacy code.

Rust adoption checklist

1. Start small & run a feasibility study on a single module to test Rust compatibility

2. Prove value by developing a PoC to see how it performs

3. Roll out an MVP, incrementally adding Rust modules

4. Starting with seniors, provide guidance, and gradually upskill the broader team

5. Track faster feature delivery, fewer bugs, and improved system security

About Yalantis

Yalantis is an engineering partner specializing in complex digital product development, with deep expertise in IoT, embedded systems, and Rust adoption. Since 2008, we have been helping companies across industries, including FinTech, healthcare, real estate, and supply chain, build reliable software solutions that evolve with their business needs.

Our Rust development services ensure mission-critical systems run faster, safer, and more efficiently, while integrating seamlessly with existing C, C++, or other legacy systems.

Yalantis

yalantis.com