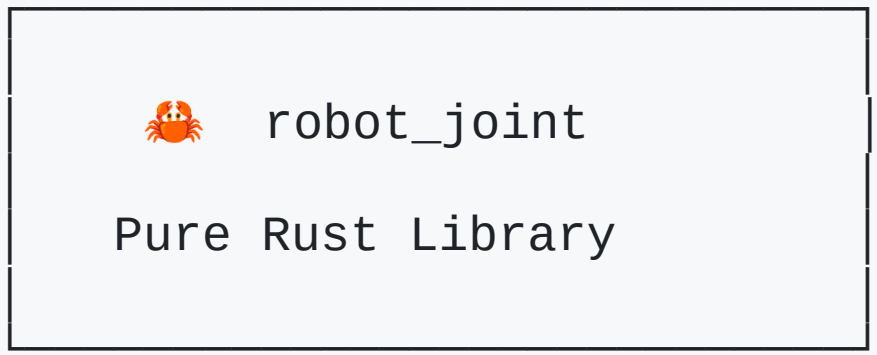
Building Bridges: Patterns for Rust/C++ Interop

2025-11-10

betterCode() Rust

Tyler Weaver

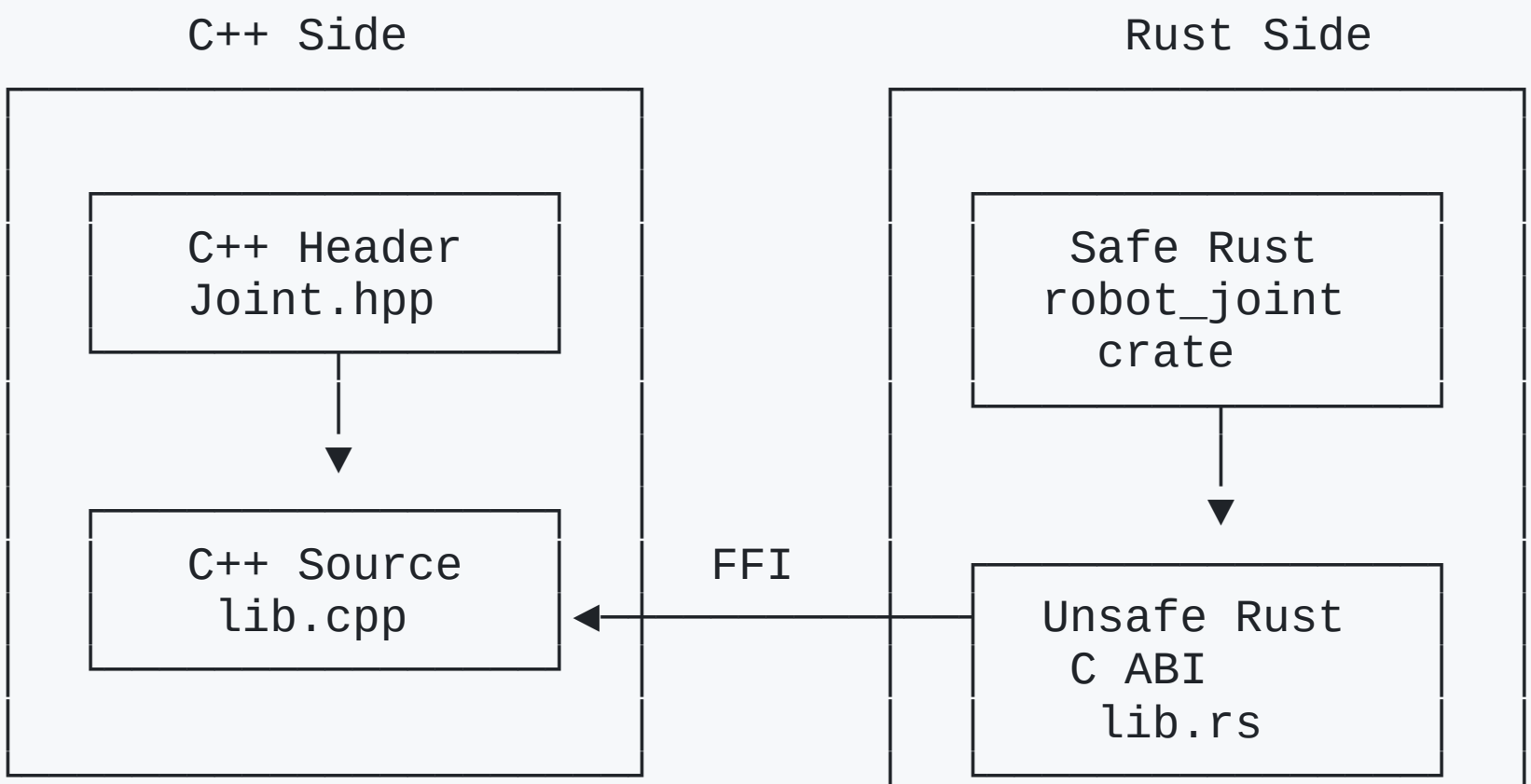
Part 1: The Core Component (A Pure Rust Library)



crates/robot_joint/src/joint.rs

```rust
/// Represents a robot joint with its kinematic properties
#[derive(Clone, Debug)]
pub struct Joint {
    /// Name of the joint
    name: String,
    /// Transform from parent link to joint origin
    parent_link_to_joint_origin: Isometry3<f64>,
    // ... other fields
}

impl Joint {
    /// Create a new joint with the given name
    pub fn new(name: String) -> Self { /* ... */ }

    /// Calculate the transform for this joint given joint variables
    pub fn calculate_transform(&self, variables: &[f64]) -> Isometry3<f64> {
        // ... implementation
    }
}
```

Part 2: Manual FFI (The Foundational Approach)



We manually create a C-compatible API (`extern "C"`) and a C++ wrapper that provides a safe, idiomatic interface.

`crates/robot_joint-cpp/src/lib.rs`

```rust
/// Opaque handle to a Rust Joint object
pub struct RobotJointHandle {
    joint: Joint,
}

#[unsafe(no_mangle)]
pub extern "C" fn robot_joint_new(name: *const c_char) -> *mut RobotJointHandle {
    // ... create Joint, box it, and return a raw pointer
}

#[unsafe(no_mangle)]
pub extern "C" fn robot_joint_free(joint: *mut RobotJointHandle) {
    if !joint.is_null() {
        unsafe { drop(Box::from_raw(joint)); }
    }
}
```

crates/robot_joint-cpp/include/robot_joint.hpp

```cpp
// Forward-declaration of the opaque Rust type
namespace robot_joint::rust {
struct RobotJointHandle;
}

// Custom deleter that calls our extern "C" free function
template<auto fn>
struct deleter_from_fn {
    template<typename T>
    constexpr void operator()(T* arg) const { fn(arg); }
};

class Joint {
public:
    explicit Joint(const std::string& name) noexcept;
private:
    // This unique_ptr now safely manages the Rust object's memory
    std::unique_ptr<rust::RobotJointHandle,
                    deleter_from_fn<robot_joint_free>> joint_;
};
```

For complex types like matrices, we define a C-compatible struct and pass pointers.

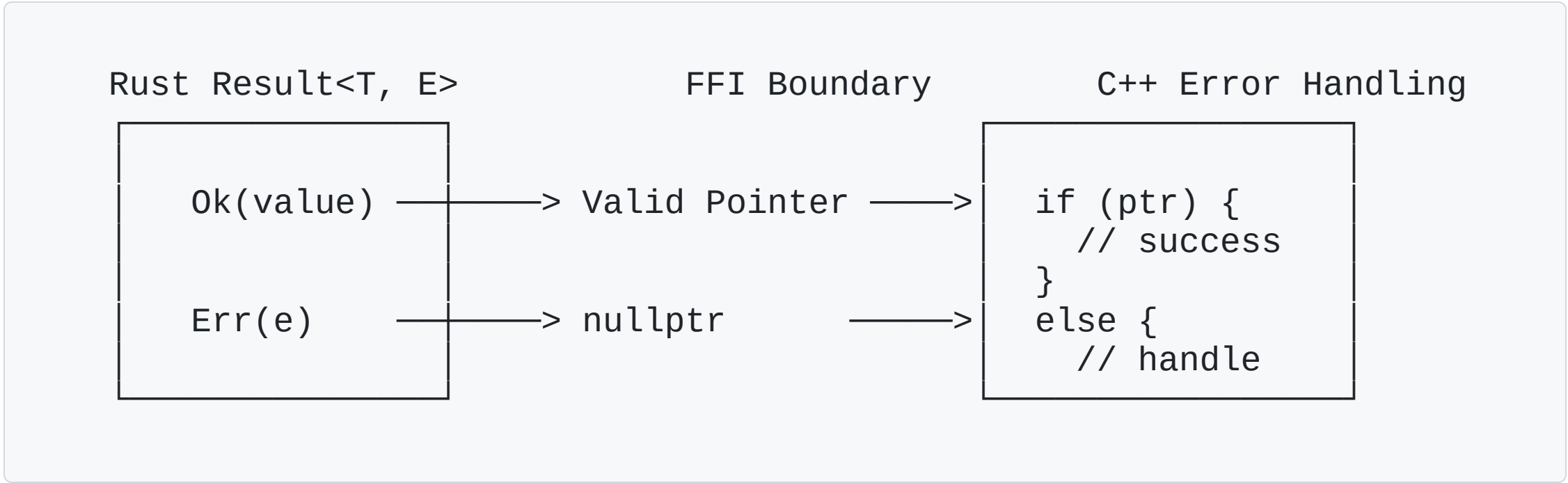
`crates/robot_joint-cpp/src/lib.rs`

```rust
#[repr(C)]
pub struct Mat4d {
    pub data: [c_double; 16], // Column-major for Eigen
}

#[unsafe(no_mangle)]
pub extern "C" fn robot_joint_calculate_transform(
    joint: *const RobotJointHandle,
    variables: *const c_double,
    size: c_uint,
) -> Mat4d {
    // ... implementation ...
}
```

`crates/robot_joint-cpp/src/lib.cpp`

```cpp
Eigen::Isometry3d Joint::calculate_transform(const Eigen::VectorXd& variables) const {
    const auto rust_transform = robot_joint_calculate_transform(
        joint_.get(),
        variables.data(),
        static_cast<unsigned int>(variables.size())
    );
    // Map the raw data directly into an Eigen type
    Eigen::Isometry3d transform;
    transform.matrix() = Eigen::Map<const Eigen::Matrix4d>(rust_transform.data);
    return transform;
}
```

Error Handling at FFI Boundaries



Pattern 1: Null Pointers for Errors (Manual FFI)

`crates/robot_joint-cpp/src/lib.rs`

```rust
#[unsafe(no_mangle)]
pub extern "C" fn robot_joint_new(name: *const c_char) -> *mut RobotJointHandle {
    if name.is_null() {
        return ptr::null_mut();  // Error: invalid input
    }

    let name_cstr = unsafe { CStr::from_ptr(name) };
    let name_str = match name_cstr.to_str() {
        Ok(s) => s,
        Err(_) => return ptr::null_mut(),  // Error: invalid UTF-8
    };

    let joint = Joint::new(name_str.to_string());
    Box::into_raw(Box::new(RobotJointHandle { joint }))
}
```

C++ Side: Check and Handle

`crates/robot_joint-cpp/src/lib.cpp`

```cpp
Joint::Joint(const std::string& name) noexcept
    : joint_{robot_joint_new(name.c_str())} {
    // Constructor handles null gracefully
}

std::string Joint::name() const {
    if (!joint_) {
        return "";  // Return safe default
    }
    const char* name_ptr = robot_joint_get_name(joint_.get());
    if (!name_ptr) {
        return "";  // Handle Rust returning null
    }
    std::string result(name_ptr);
    robot_joint_free_string(const_cast<char*>(name_ptr));
    return result;
}
```

Pattern 2: Return Default Values on Error

`crates/robot_joint-cpp/src/lib.rs`

```
#[unsafe(no_mangle)]
pub extern "C" fn robot_joint_calculate_transform(
    joint: *const RobotJointHandle,
    variables: *const c_double,
    size: c_uint,
) -> Mat4d {
    // Return identity matrix on error
    let identity = Mat4d {
        data: [
            1.0, 0.0, 0.0, 0.0,  // Identity matrix as
            0.0, 1.0, 0.0, 0.0,  // safe default
            0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 1.0,
        ],
    };
    if joint.is_null() || variables.is_null() {
        return identity;  // Safe fallback
    }
```

Pattern 3: Error Codes with Out Parameters

```rust
#[repr(C)]
pub enum ErrorCode {
    Success = 0,
    NullPointer = 1,
    InvalidUtf8 = 2,
    OutOfBounds = 3,
}

#[unsafe(no_mangle)]
pub extern "C" fn robot_joint_try_calculate(
    joint: *const RobotJointHandle,
    variables: *const c_double,
    size: c_uint,
    out_matrix: *mut Mat4d,
) -> ErrorCode {
    if joint.is_null() || out_matrix.is_null() {
        return ErrorCode::NullPointer;
    }

    // ... perform calculation
    unsafe { *out_matrix = result; }
    ErrorCode::Success
}
```

FFI Error Handling

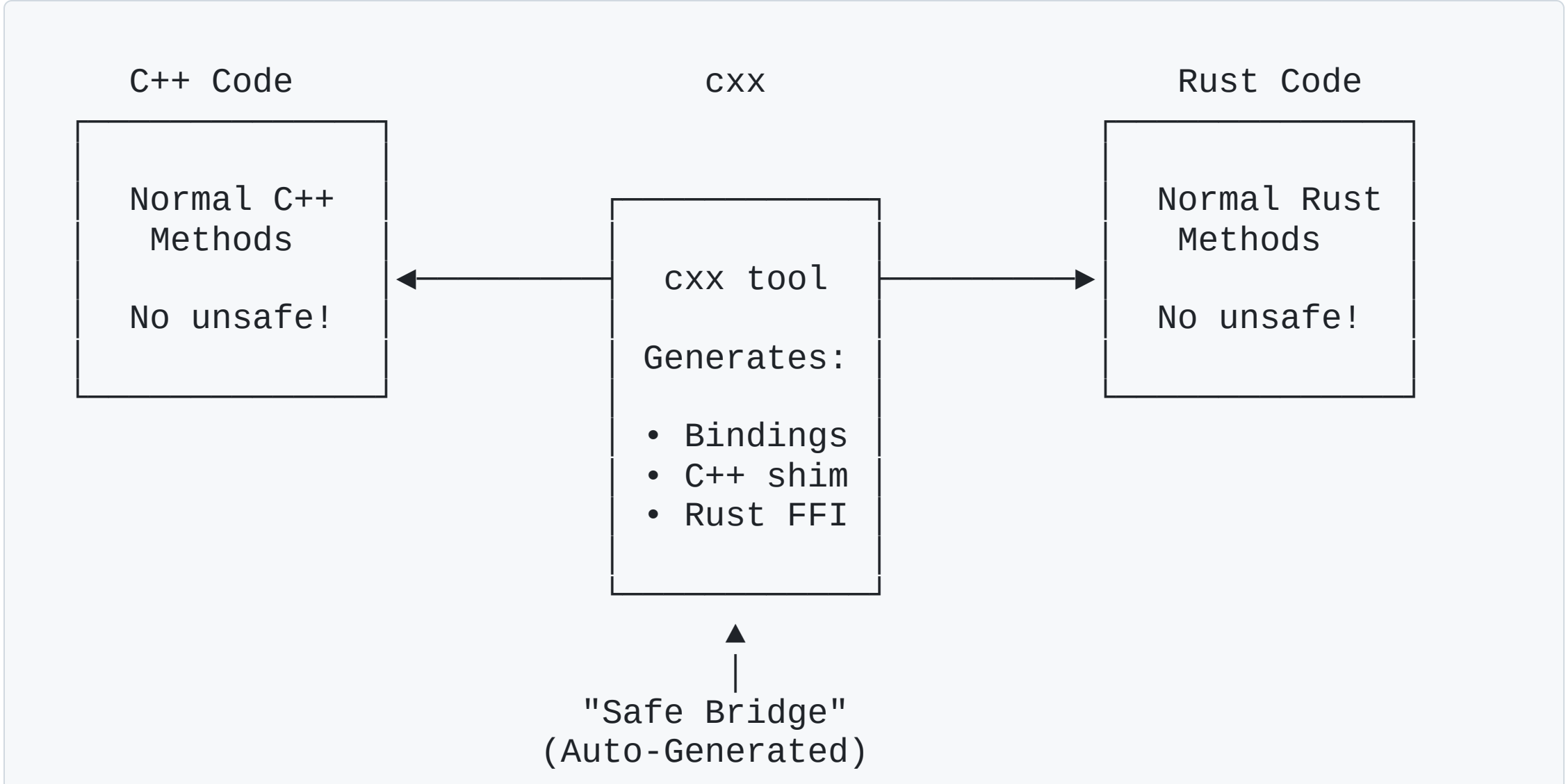
DO

- Use null pointers for optional returns
- Return error codes for complex failures
- Provide safe defaults (identity matrix, etc)
- Document error conditions in comments
- Validate all inputs at FFI boundary

DON'T

- Panic in FFI functions
- Propagate Result<T,E> across FFI
- Use exceptions across FFI boundary
- Assume C++ will handle Rust unwinding

Part 3: `cxx` (Safe Code Generated Interop)



`crates/robot_joint-cxx/src/lib.rs`

```rust
#[cxx::bridge(namespace = "robot_joint")]
mod ffi {
    extern "Rust" {
        type Joint;
        fn new_joint(name: &str) -> Box<Joint>;
        fn name(self: &Joint) -> String;
        fn calculate_transform(self: &Joint, variables: &[f64]) -> Vec<f64>;
        // ... other methods
    }
}

// Wrapper around the pure Rust Joint for cxx interop
pub struct Joint(robot_joint::Joint);

// Implementation of the bridge functions
fn new_joint(name: &str) -> Box<Joint> { /* ... */ }
// ...
```

Note the complete lack of `unsafe` code!

The `build.rs` script coordinates the code generation.

`crates/robot_joint-cxx/build.rs`

```
fn main() {
    cxx_build::bridge("src/lib.rs") // Path to the file with the bridge
        .std("c++20")
        .compile("robot_joint_cxx"); // Name of the generated library

    println!("cargo:rerun-if-changed=src/lib.rs");
}
```

On the C++ side, you simply include the generated header and call the functions.

`examples/cxx_example.cpp`

```cpp
#include <robot_joint/robot_joint.hpp> // Helper functions
#include <robot_joint-cxx/src/lib.rs.h> // cxx-generated header

int main() {
    // Create a joint using the cxx interface
    auto joint = robot_joint::new_joint("cxx_example_joint");

    // `cxx` handles the types automatically
    std::cout << "Joint name: " << joint->name() << std::endl;
}
```

Error Handling in `cxx`

`crates/robot_joint-cxx/src/lib.rs`

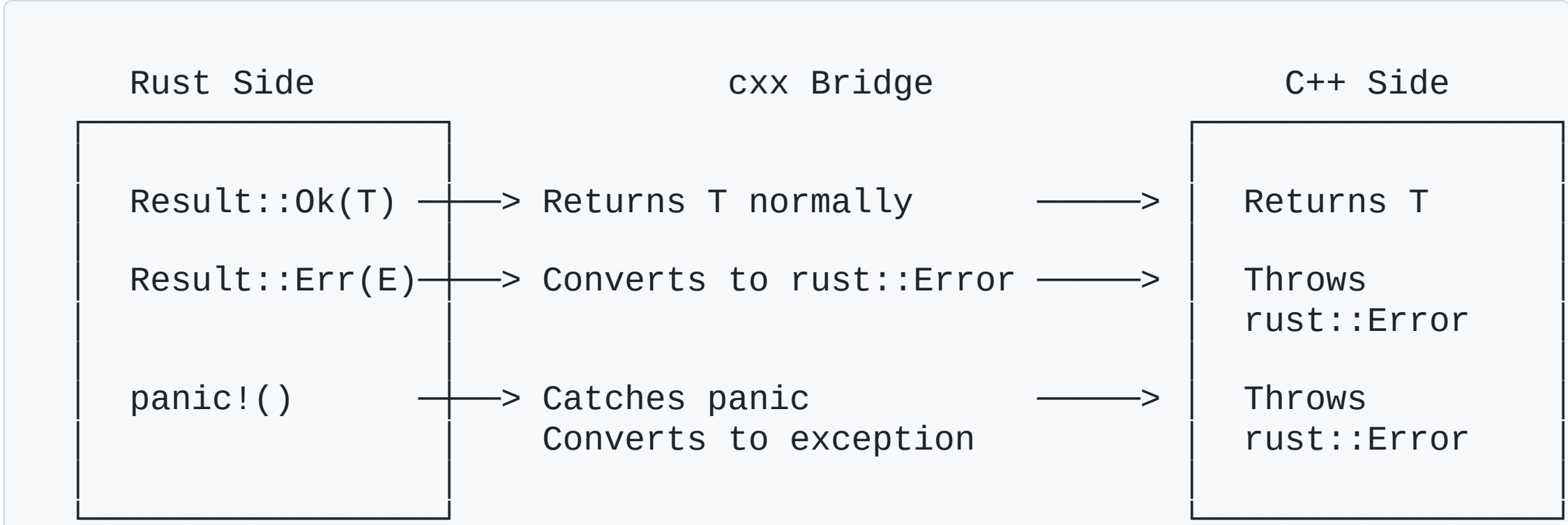
```rust
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        fn try_new_joint(name: &str) -> Result<Box<Joint>>;
    }
}

fn try_new_joint(name: &str) -> Result<Box<Joint>, Box<dyn std::error::Error>> {
    if name.is_empty() {
        return Err("Joint name cannot be empty".into());
    }
    if name.len() > 100 {
        return Err("Joint name too long".into());
    }
    Ok(Box::new(Joint(robot_joint::Joint::new(name.to_string()))))
}
```

C++ Side: Automatic Exception Translation

`examples/cxx_example.cpp`

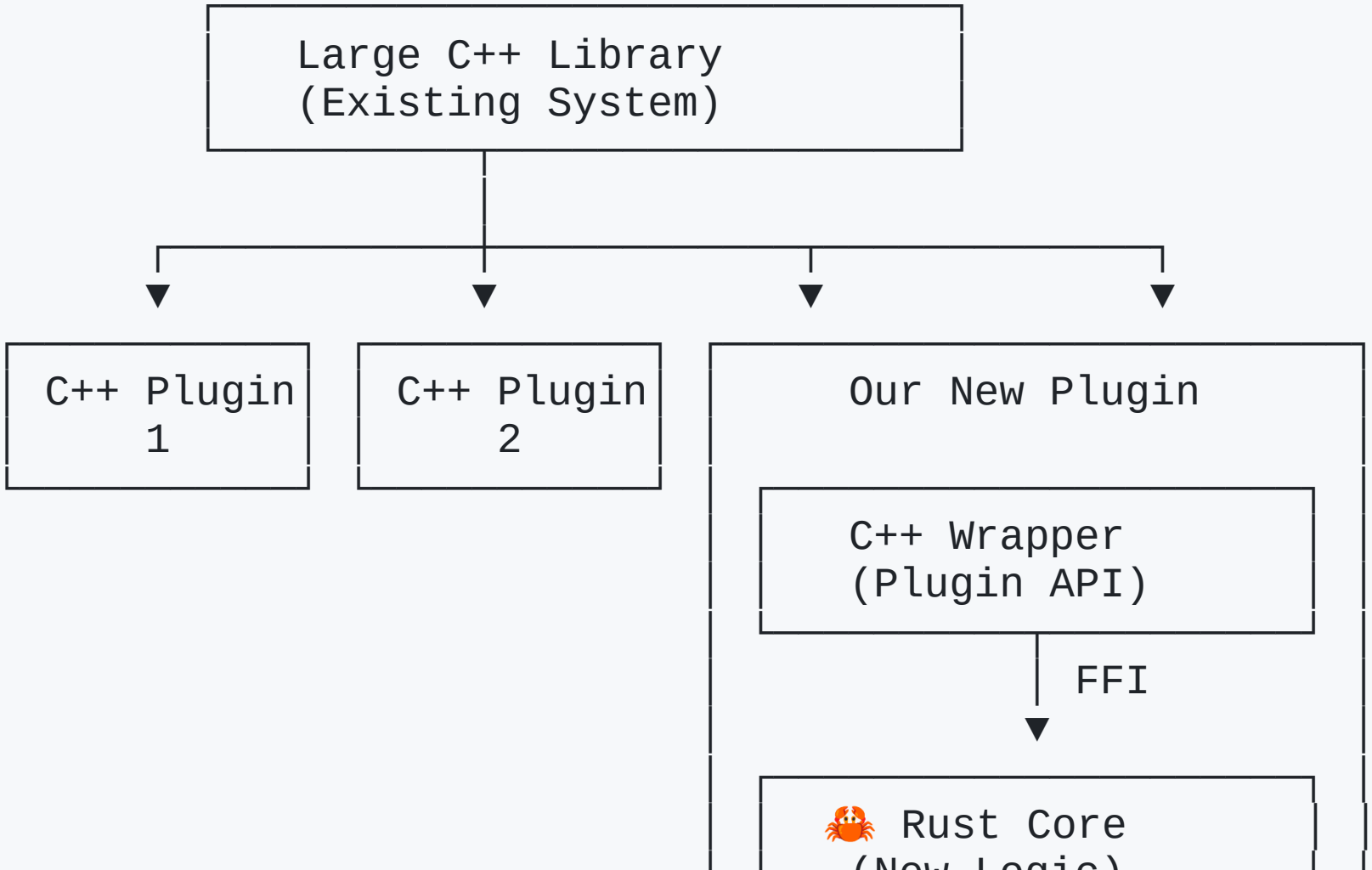
```cpp
#include <robot_joint-cxx/src/lib.rs.h>
#include <iostream>

int main() {
    try {
        // cxx automatically converts Result::Err to C++ exceptions
        auto joint = robot_joint::try_new_joint("");  // Empty name!
    } catch (const rust::Error& e) {
        std::cerr << "Rust error: " << e.what() << std::endl;
        // Output: "Rust error: Joint name cannot be empty"
    }
}
```

`cxx` Error Handling Magic



Key Benefits:
- No manual error code checking
- Type-safe error propagation
- Automatic panic protection
- Zero unsafe code needed

Part 4: CMake Integration (Making it Usable)



The goal: allow a C++ developer to use our Rust library without needing to know Cargo.

```
# The C++ developer experience we want to enable:
include(FetchContent)
FetchContent_Declare(
  robot_joint
  GIT_REPOSITORY https://github.com/tylerjw/tylerjw.dev
  GIT_TAG main # Or a specific version
  SOURCE_SUBDIR "rust-cpp-interop-example/crates/robot_joint-cxx"
)
FetchContent_MakeAvailable(robot_joint)

# Link to your executable
target_link_libraries(my_cpp_app PRIVATE robot_joint::robot_joint)
```

We use `Corrosion`, a tool for integrating Rust into CMake projects.

`crates/robot_joint-cxx/CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.16)
project(robot_joint)

# Fetch and configure Corrosion for Rust integration
include(FetchContent)
FetchContent_Declare(
  Corrosion
  GIT_REPOSITORY https://github.com/corrosion-rs/corrosion.git
  GIT_TAG v0.5)
FetchContent_MakeAvailable(Corrosion)

# This single command tells Corrosion to build our Rust crate
# (including running the cxx build.rs script)
# and creates a CMake target for the resulting static library.
corrosion_import_crate(MANIFEST_PATH Cargo.toml)
```

`crates/robot_joint-cxx/CMakeLists.txt`

```
# Create a C++ wrapper library
add_library(robot_joint_wrapper INTERFACE)

# Add include directories for our headers and the cxx-generated ones
target_include_directories(robot_joint_wrapper INTERFACE ... )

# Link the Rust library (from corrosion_import_crate) and other deps
target_link_libraries(robot_joint_wrapper INTERFACE
  robot_joint_cxx # The Rust library target
  Eigen3::Eigen)

# Create an alias for easy consumption
add_library(robot_joint::robot_joint ALIAS robot_joint_wrapper)

# Standard CMake install rules...
install(TARGETS ...)
install(EXPORT ...)
```

Part 5: Choosing Your Pattern

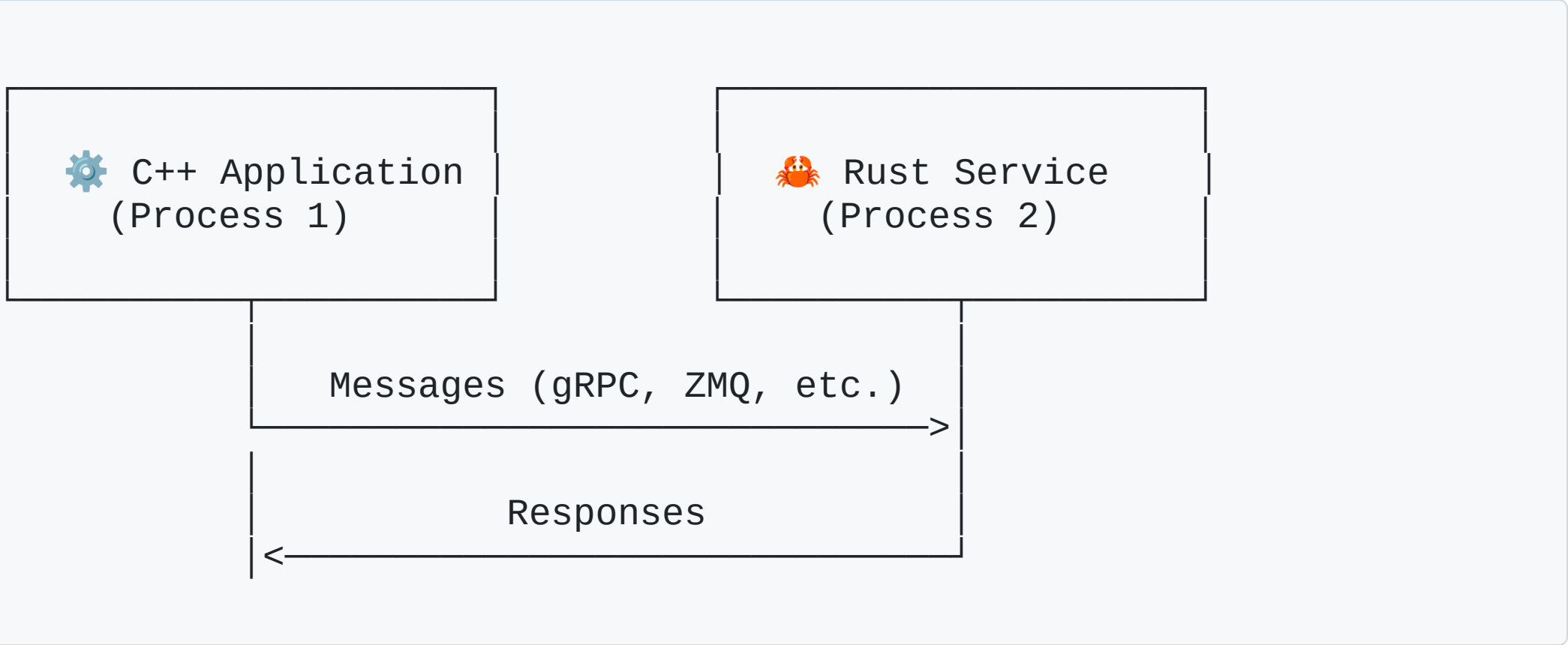
Comparison: Manual FFI vs. `cxx`

Aspect	Manual FFI	`cxx`-based
Safety	`unsafe` required. Prone to human error.	Mostly safe. `cxx` handles unsafe code.
Effort	High. Requires writing and maintaining C-API, C++ wrapper, and build scripts.	Low. `cxx` generates the bridge code.
Expressiveness	Limited to C-compatible types (pointers, primitives, `repr(C)` structs).	Richer. Supports strings, vectors, `unique_ptr`, and shared types.

Comparison: Manual FFI vs. `cxx`

Aspect	Manual FFI	`cxx`-based
Build Complexity	Simpler `build.rs`, more complex C++ build.	Requires `cxx-build` dependency and code generation step.
Performance	Optimal. Zero-overhead calls.	Very good. Minimal, often negligible overhead.
Recommendation	For performance-critical paths or when avoiding build-time dependencies is key.	**The default choice for most new projects.**

Alternative: Message Passing for Loose Coupling



This pattern is less about *linking* and more about *communicating*. Instead of a shared library, your C++ and Rust code run as independent processes.

Alternative: Message Passing for Loose Coupling

Choose This Pattern When:

- **You Need Strong Isolation:** A crash in the Rust service cannot crash the C++ application.
- **Services are on Different Machines:** The only choice for distributed systems.
- **You Have a Polyglot System:** Your C++ app needs to talk to services in Rust, Go, Python, etc.
- **You Need Independent Deployment & Scaling:** The Rust service can be updated and scaled on its own.

Alternative: Message Passing for Loose Coupling

Trade-Offs:

- **Performance Overhead:** Not suitable for tight, high-frequency loops due to serialization and IPC/network latency.
- **Complexity:** Requires defining a message format (e.g., Protobuf, JSON) and managing the communication channel.

Bonus: Catching Bugs with Sanitizers

`Makefile`

```
.PHONY: sanitizer-tests
sanitizer-tests:
        # Manual FFI with AddressSanitizer
        cd $(ROBOT_JOINT_CPP_DIR) && \
        cmake -B build-asan -S . -DCMAKE_CXX_FLAGS="-fsanitize=address ..." && \
        cmake --build build-asan && ctest --test-dir build-asan

        # Cxx version with AddressSanitizer
        cd $(ROBOT_JOINT_CXX_DIR) && \
        cmake -B build-asan -S . -DCMAKE_CXX_FLAGS="-fsanitize=address ..." && \
        cmake --build build-asan && ctest --test-dir build-asan
```

Conclusion

- For most cases, use `cxx` for a safe, code-generated bridge (`robot_joint-cxx`).
- Use **manual FFI** when performance is critical (`robot_joint-cpp`).
- Package your Rust library as a **CMake target** using `Corrosion` for easy C++ consumption.
- Consider **message passing** for loosely coupled, distributed systems.
- **Test the boundary!** Use sanitizers to catch FFI errors.
- See the full example at https://tylerjw.dev