Field Experience: Porting a Derivatives Exchange to Coroutine-Native I/O

Document Number:	P4125R0
Date:	2026-03-23
Audience:	SG14, LEWG

Reply-to:	Mungo Gill mungo.gill@me.com C++ Alliance Proposal Team

Table of Contents

Abstract

Revision History

- R0: March 2026

1. Disclosure
2. Background
   - 2.1 The Integration Partner
   - 2.2 The Libraries Under Test
   - 2.3 Scope and Limitations of the Integration
3. Methodology
4. Callback-to-Coroutine Migration
   - 4.1 Transition Difficulty
   - 4.2 Incremental Adoption
   - 4.3 Coroutine-Only Design
5. Error Handling
6. Early Assessment
7. Observations
   - 7.1 Error Return Paths
   - 7.2 Coroutine-Native I/O in Practice
   - 7.3 Accessibility
8. Limitations of This Study

Acknowledgements

References

Abstract

A derivatives exchange is porting from Asio callbacks to coroutine-native I/O. Early results: it works.

The paper reports qualitative findings from two structured interviews with the engineering team. The results are preliminary - the integration covers a subset of the platform and no performance benchmarks have been completed - but the field evidence is reported here for the committee's consideration.

Revision History

R0: March 2026

- Initial version. Early-stage qualitative findings only.

---

1. Disclosure

The author is affiliated with the C++ Alliance, which develops Capy[1] and Corosio[2], the libraries under test. The integration partner is an independent commercial entity. The author has papers before the committee proposing coroutine-based I/O (P4003R0 "Coroutines for I/O"[3]) and analysing `std::execution` (P4007R1 "Open Issues in std::execution::task"[4], P4014R0 "The Sender Sub-Language"[5]).

This paper documents findings. It does not propose changes, request committee time, or advocate for a position. The evidence is early-stage and the authors acknowledge its limitations explicitly in Section8.

---

2. Background

The integration involves a commercial derivatives exchange operator porting from Boost.Asio to a coroutine-native library developed by the C++ Alliance.

2.1 The Integration Partner

The integration partner has developed one of the world's highest performance derivatives exchange platforms. The platform supports 24/7 markets across all major asset types and is considered "equities grade" but with full derivative support including options. It was built by engineers who pioneered modern exchange architecture and who were behind the tech stacks that underpin markets such as NYSE, NASDAQ, and HKEx.

The primary codebase comprises numerous repositories (primarily C++), with Boost.Asio as a foundational process building block for application pipelines and session management. The majority of the code uses Asio callbacks; a small island of coroutine code exists in the most performance-critical path. As is standard practice in low-latency financial markets infrastructure, the organisation does not use exceptions in production code.

2.2 The Libraries Under Test

Capy is a C++ library providing async/coroutine building blocks and executor models. Corosio is a networking library built on Capy, providing async socket operations. Both are designed with C++20 coroutines as first-class citizens and are described in detail in P4003R0[3]. The key design commitment is: coroutines only. No callbacks, futures, or sender/receiver interfaces. Every I/O operation returns an awaitable.

2.3 Scope and Limitations of the Integration

The initial port focuses on a subset of repositories needed to re-run the partner's matching facility benchmark tests with Corosio replacing Asio. The scope covers core executor, IO context, and TCP socket functionality. UDP and WebSocket support - required for market data feeds and external client connectivity respectively - are not yet available in Corosio and are excluded from this evaluation.

The project is structured in phases: foundational library porting first, then TCP client/server components, then benchmark execution. The central research question is whether a coroutine-native I/O library can match or exceed Asio's performance in a mission-critical production system. The benchmarking phase has not yet begun.

---

3. Methodology

Two structured interviews were conducted with members of the integration partner's engineering team in March 2026:

- **Engineer A** (CEO, highly experienced C++ engineer): ~90 minute interview covering build integration, incremental adoption strategy, and architectural assessment.
- **Engineer B** (platform architect): ~70 minute interview covering callback-to-coroutine migration, error handling, and production readiness assessment.

Both engineers had been working with Capy and Corosio for approximately two weeks at the time of their interviews. All quotes in this paper are verbatim transcriptions. A project journal maintained by the engineering team provided supplementary context.

The interviews were qualitative. No metrics, benchmarks, or automated measurements are reported. The findings represent the subjective assessments of two experienced engineers at the early stage of an ongoing integration. They carry the weight appropriate to their scope.

Both interviews were conducted remotely over video call; the Capy/Corosio author was present for technical questions but did not participate in the assessment discussions.

---

4. Callback-to-Coroutine Migration

The partner's codebase is predominantly callback-based. The following subsections report how the transition to coroutines proceeded in practice.

4.1 Transition Difficulty

Migrating production callback-based code to coroutines was feasible and less disruptive than anticipated.

> *"It was actually easier than expected." - Engineer B*

> *"The actual move from callbacks to coroutines took a bit of thought but it was overall not a massive change." - Engineer B*

> *"The changes we've had to make haven't been as drastic as maybe once thought." - Engineer B*

Recursive callback patterns (retry loops, reconnection handlers) converted naturally to structured coroutine loops, which the engineers described as simpler and more readable than the callback originals.

4.2 Incremental Adoption

Engineer A designed a "springboard function" approach to insulate the existing callback codebase from requiring full coroutine propagation:

> *"Can I just do a springboard function? So if I have a Capy-style executor and I need to call a coroutine, can I just wrap that in a box and pretend I didn't do that?" - Engineer A*

> *"This might be a path that other people could follow. It's a band-aid. We know that." - Engineer A*

The approach uses an `executor_traits` abstraction layer allowing parallel Asio and Capy implementations. Tests run identically against both backends, validating behavioural equivalence. The dual-backend approach allowed the team to port incrementally without disrupting the existing Asio codebase.

4.3 Coroutine-Only Design

Engineer B endorsed the coroutine-only design philosophy:

> *"The tradeoffs around callbacks versus coroutines - if you've got a sufficiently modern codebase, I think there's a very strong reason that you could choose the coroutine route and have very little tradeoff." - Engineer B*

> *"Doing one thing and doing it very well and focusing just on that, I think, has a lot of merit." - Engineer B*

---

5. Error Handling

Engineer B's primary friction point was error handling. Corosio throws exceptions where Asio provides `error_code` overloads.

> *"In some cases in Corosio and Capy, I think there's only exceptions. It would be actually quite nice if you were able to either pass in an error code that gets updated or return an error code. The one I'm thinking of is socket set_option - it only throws." - Engineer B*

> *"We don't use exceptions at all. In some very non-intensive bits of code we'll maybe use exception handling, but overall we don't use that. Having a consistent way to report errors would be a worthwhile set of updates." - Engineer B*

The Capy/Corosio author proposed a boundary during the interview: exceptions for programmer errors (logic errors, misconfiguration) and error codes for runtime conditions. Engineer B found this reasonable in principle but probed the boundary:

> *"Should setting the no-delay flag fail? Is that an exceptional circumstance, or is it fine to continue on with the socket open and not having set that?" - Engineer B*

The exception-vs-error-code boundary is a recurring design question in C++ I/O libraries. This question is directly relevant to financial markets infrastructure, where exception-free code paths are a standard requirement.

---

6. Early Assessment

Engineer A's assessment is exploratory - the springboard approach needs performance validation:

> *"We might surprisingly find that actually it is feasible to take this approach. And this might be a path that other people could follow." - Engineer A*

Engineer B's assessment is more confident:

> *"I think this is a viable production replacement. So far, at least from what we've seen - not that we've had gigabytes of traffic flowing through this thing yet - but definitely the way it's been written, the way we've been porting code across to it, the way the tests continue to work... I would be quite confident that it would be robust and a reasonable alternative or replacement." - Engineer B*

> *"It doesn't feel brittle. It feels like it's been quite well thought out, and the changes we've had to make around some of this stuff haven't been excruciating." - Engineer B*

> *"I would definitely not steer anyone away from it after what I've experienced so far." - Engineer B*

Engineer B's endorsement carries a caveat for teams with deeply entrenched Asio codebases - particularly those with complex multi-threading and multiple thread pools, where the migration would be substantially harder.

These assessments are based on approximately two weeks of integration work covering a subset of the platform. No performance benchmarks have been completed. These are early indicators, not final verdicts.

7. Observations

The following observations connect the field findings to topics under discussion in the committee. They are stated as observations, not recommendations.

7.1 Error Return Paths

P4007R1[4] identifies error return as a "not fixable post-ship" gap in `std::execution::task`. The integration partner's experience illustrates the practical dimension of this question: production financial markets infrastructure requires consistent `error_code` paths. The partner does not use exceptions in production code. When Corosio's `socket.set_option()` threw where Asio provides an `error_code` overload, it was their primary API complaint.

How errors flow through coroutine pipelines is not an abstract design question. It has immediate consequences for adoption in domains where exception-free code is a requirement.

7.2 Coroutine-Native I/O in Practice

P4003R0[3] proposes that C++20 coroutines are suited to I/O, and P4014R0[5] argues that coroutines (direct style) are a natural complement to senders (continuation-passing style). The integration described in this paper provides early qualitative evidence bearing on these claims.

A derivatives exchange platform with a large, predominantly C++ codebase and a majority callback-based architecture is porting to a coroutine-native library. After two weeks of work, two engineers independently found the migration less disruptive than anticipated and endorsed the coroutine-only design philosophy. These are preliminary impressions from experienced engineers, not validated production results.

P4007R1[4] notes that zero open-source sender-based networking stacks exist, while six coroutine-based stacks are in production use. This integration adds a seventh data point.

7.3 Accessibility

P4014R0[5] argues that the sender model constitutes a "sub-language" with its own control flow, variable binding, and error handling - creating a learning burden distinct from standard C++. Both engineers in this study found a coroutine-native API at least as accessible as Asio's callback model. Engineer A observed:

> *"This library is probably something a new user could turn to and use pretty much directly." - Engineer A*

8. Limitations of This Study

This paper reports early-stage, qualitative findings. The following limitations apply:

- **Small sample.** Two engineers from one organisation. Their experience may not generalise.
- **Early stage.** Approximately two weeks of integration work. The porting covers a subset of the platform. No production traffic has been routed through the coroutine-native code.
- **No benchmarks.** The central research question - whether coroutine-native I/O can match or exceed Asio's performance - remains unanswered. The benchmarking phase has not begun.
- **Qualitative, not quantitative.** All findings are based on interview responses. No automated measurements, code metrics, or defect counts are reported.
- **Author affiliation.** The libraries under test were developed by the author's organisation. The integration partner is independent, but the study design and reporting are not.
- **Incomplete feature coverage.** UDP and WebSocket support are not yet available. The evaluation covers TCP socket operations only.

A follow-up paper with benchmark results and broader feature coverage is planned when the integration reaches that stage.

The qualitative question - whether a coroutine-native library is feasible for this domain - has an early answer. The quantitative question remains open.

Acknowledgements

The author thanks the engineering team at the integration partner for their time, candour, and willingness to share their experience. Their structured feedback on documentation, API design, and migration strategy has informed improvements to both Capy and Corosio.

Thanks to Vinnie Falco for developing Capy and Corosio and for supporting this evaluation. Thanks to Steve Gerbino for technical contributions to both libraries.

References

1. Capy - C++ async/coroutine building blocks. https://github.com/cppalliance/capy

2. Corosio - Coroutine-native networking library. https://github.com/cppalliance/corosio

3. P4003R0 - "Coroutines for I/O" (Vinnie Falco, 2026). https://wg21.link/p4003r0

4. P4007R1 - "Open Issues in std::execution::task" (Vinnie Falco, 2026). https://isocpp.org/files/papers/P4007R1.pdf

5. P4014R0 - "The Sender Sub-Language" (Vinnie Falco, Mungo Gill, 2026). https://wg21.link/p4014r0