Coroutine-Native I/O at a Derivatives Exchange

Document Number:	P4125R0
Date:	2026-04-17
Intent:	Inform
Audience:	SG14, LEWG
Reply-to:	Mungo Gill mungo.gill@me.com C++ Alliance Proposal Team

Table of Contents

Abstract

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

The paper reports qualitative findings from three 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: April 2026 (post-Croydon mailing)

- Initial version. Early-stage qualitative findings only.

---

1. Disclosure

The author provides information and serves at the pleasure of the committee.

This paper is part of the Network Endeavor (P4100R0), a project to bring coroutine-native I/O to C++.

Falco developed and maintains Capy[1] and Corosio[2] and believes coroutine-native I/O is a practical foundation for networking in C++.

The author is affiliated with the C++ Alliance, which develops the libraries under test. The integration partner is an independent commercial entity. The evidence is early-stage and the authors acknowledge its limitations explicitly in Section 7.

This paper asks for nothing.

---

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

Three 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.
- **Engineer C** (developer, newer to the codebase): ~55 minute interview covering build experience, documentation quality, timer migration, and coroutine design assessment. Engineer C had no prior production experience with C++ coroutines.

Engineers A and B had been working with Capy and Corosio for approximately two weeks at the time of their interviews. Engineer C was interviewed one week later. All quotes are from interview transcriptions, lightly edited for readability. 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 three engineers at the early stage of an ongoing integration. They carry the weight appropriate to their scope.

All 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*

Engineer C, working independently on a different part of the codebase, reached the same conclusion:

> *"It was easier than expected I think." - Engineer C*

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 Timer Migration

Engineer C's work focused on replacing Asio timer callbacks with coroutine equivalents. The Asio pattern uses recursive callbacks - a timer fires, executes a handler, and re-arms itself within the same handler. This pattern does not translate directly to coroutines:

> *"We relied on an Asio callback which does a recursive thing - it's kind of like set up the timer and it expires on a given interval, keeps re-entering the same function. But with coroutines I don't think you can do that recursively." - Engineer C*

Engineer C considered symmetric transfer but concluded it was not the right fit for this pattern. The correct coroutine equivalent is a simple loop with `co_await` on each timer expiry - a structurally simpler construct, but one that requires recognising the pattern translation.

4.3 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.

4.4 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*

Engineer C found the coroutine model more intuitive to write but was more cautious about its fit with existing code:

> *"It feels like there's a ground up approach with Capy and the use of the tasks which is really easy to use and it feels like it's quite easy to start constructing a larger application. But I'm still not sure how it fits into an existing application." - Engineer C*

4.5 Build Experience

Engineer C reported a straightforward build experience. The CMake snippet in the repository README compiled cleanly on the first attempt, and the dependency relationship between Capy and Corosio was handled automatically. Time from source code to a running programme was less than one hour. Compilation times were not noticeably different from Asio.

4.6 Documentation

Documentation was generally praised. Engineer C, who had no prior coroutine experience, found the introductory topics in Capy on coroutines and concurrency brought him up to speed effectively. He compared the documentation favourably to Asio's:

> *"With the Capy documentation there's obviously a lot more of a background explained with your documents." - Engineer C*

Areas identified for improvement included the stream concepts section (which assumed background knowledge the developer did not have) and code examples (which could benefit from wider context - full programmes rather than isolated snippets). These are documentation issues, not library design issues, but they affect adoption.

---

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 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 C's assessment was more measured - he is earlier in the integration process and working on a narrower scope:

> *"I would say it's felt like a smooth process so far." - Engineer C*

> *"We'd have to be sure that production or similar deployment of all our existing components have the same behaviour. So that's still a while off." - Engineer C*

When asked what he would warn another team considering the port:

> *"I'd probably warn about the length of time that it would take. Wondering whether trading off the length of time would be worth it." - Engineer C*

Engineers B and C both noted caveats for teams with deeply entrenched Asio codebases - particularly those with complex multi-threading and multiple thread pools, where the migration would be substantially harder. The time investment required for a complete port should not be underestimated.

Engineer A also noted the library's accessibility:

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

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

---

7. Limitations of This Study

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

- **Small sample.** Three engineers from one organisation. Their experience may not generalise.
- **Early stage.** Approximately two to three 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.
- **Author presence.** The Capy/Corosio author was present during all interviews for technical clarification. While the author did not participate in assessment discussions, the author's presence may have influenced responses. No interviews were conducted without the author present.
- **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.

[2] Corosio - Coroutine-native networking library.

[3] P4003R0 - "Coroutines for I/O" (Vinnie Falco, Steve Gerbino, Mungo Gill, 2026).