Coroutine Executors and P2464R0

Document Number:	P4096R0
Date:	2026-04-17
Intent:	Inform
Audience:	LEWG, SG1

Reply-to:	Vinnie Falco vinnie.falco@gmail.com C++ Alliance Proposal Team

Table of Contents

Abstract

The committee set aside the Networking TS in 2021. The process had no mechanism to verify that the analysis examined every applicable framing, and no mechanism to revisit the outcome against evidence.

P2464R0[1], "Ruminations on networking and executors," identified three deficiencies in P0443R14[2]'s `execute(F&&)` - no error channel, no lifecycle for submitted work, and no generic composition - and concluded the Networking TS should be set aside. The committee acted on the analysis. The Networking TS was removed from the committee's work program.

This retrospective examines the analysis under both framings of `execute(F&&)` defined in P4094R0[3] - the work framing and the continuation framing. Under the work framing, the three deficiencies hold. Under the continuation framing, they do not arise. P2464R0[1] analyzed under the work framing only. The coroutine executor described in P4003R0[4] constrains the argument type to `coroutine_handle<>`, enforcing the continuation framing through the type system. The coroutine executor concept did not exist in 2021. P2464R0[1] could not have evaluated it.

Section 2 documents what P2464R0[1] did and what followed. Sections 3-4 apply both framings to the three criteria. Section 5 places the analysis next to the published outcomes since 2021.

Revision History

R0: April 2026 (post-Croydon mailing)

- Initial version.

1. Disclosure

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

This paper is part of the Network Endeavor (P4100R0)[5], a project to bring coroutine-native I/O to C++.

The author developed and maintains Capy[6] and Corosio[7] and believes coroutine-native I/O is a practical foundation for networking in C++.

Coroutine-native I/O and `std::execution` are complementary. Each serves the domain where its design choices pay off.

This paper examines the published record. That effort requires re-examining consequential papers, including papers written by people the author respects.

P2464R0[1] was consequential. It provided the analytical framework that led the committee to set aside the Networking TS and focus on the sender/receiver model. Decisions of that magnitude deserve periodic review. This paper provides one.

P2464R0[1] critiqued P0443R14[2]. P2300R10[8], "std::execution," addressed those deficiencies with the sender/receiver model. P4003R0[4] provides a second async model. The committee now has two implementations to evaluate against P2464R0[1]'s criteria. This paper evaluates P4003R0[4] and uses P2300R10[8] as a reference. Structural comparisons are observations, not arguments that `std::execution` should be modified or removed.

The authors co-authored P2469R0[9], "Response to P2464 - Ruminations on networking and executors," defending the Networking TS in 2021 and built Boost.Beast on Boost.Asio's completion-token model. That history makes the following statement necessary: the authors no longer believe the Networking TS would have been right for C++. The committee's conclusion was correct. This paper disagrees with the analytical path, not the destination. There is no attempt to relitigate the Networking TS or to restore it to the committee's work program.

The research that produced P4003R0[4] is what changed the authors' position. The Networking TS completion token is a generalization: any callable, any future, any coroutine, any continuation type can serve as the async resumption mechanism. That generality is real and it works. But it is a trade-off. Three async models now exist for C++, each making a different trade-off:

- P2300R10[8] trades runtime flexibility for compile-time work graphs.
- The Networking TS trades a single async model for support of every continuation type.
- P4003R0[4] trades every continuation type for coroutines only.

Each is a valid choice with real benefits and real costs. The coroutines-only trade-off is the one the authors believe is correct for networking in C++. When the completion mechanism is fixed to `coroutine_handle<>`, the operation state becomes concrete, the stream becomes type-erasable without per-operation allocation, the I/O library compiles once, and the ABI stabilizes across transport changes. None of these properties are achievable when the operation state must be parameterized on an arbitrary completion handler type. P4088R0[10] traces the full causal chain. The generality that the completion token provides is genuine, but the cost is that it forecloses the optimizations that networking in C++ has needed for twenty years. Constraining to coroutines is the trade-off that unlocks them.

This paper asks for nothing.

---

2. What P2464R0 Did

In October 2021, Ville Voutilainen published P2464R0[1] on behalf of the Finnish National Body. The paper analyzed P0443R14[2]'s `execute(F&&)` and identified three deficiencies:

1. No error channel.
2. No lifecycle for submitted work.
3. No generic composition.

The paper concluded that the Networking TS should be set aside. The committee acted on the analysis. The Networking TS - published as an ISO TS in 2018, built on Boost.Asio's fifteen years of deployment - was removed from the committee's work program. The committee's async effort redirected toward P2300R10[8], the sender/receiver model.

P2464R0[1] analyzed the specification as written. The analysis was procedurally correct. The three deficiencies it identified in `execute(F&&)` are real properties of that API surface under the framing the paper adopted. The committee evaluated the analysis, voted, and acted. The process functioned as designed.

---

3. The Two Framings

P2464R0[1] analyzed `execute(F&&)` under one framing. P4094R0[3] documents the scope expansion that produced P0443R14[2] from three independent executor models and the terminology shift that erased the continuation framing from the API surface. Two framings of `execute(F&&)` exist.

The continuation framing. `dispatch`/`post`/`defer` schedule a continuation on an execution context. The callable is a resumption handle. The operating system performs the work. The result is delivered to the continuation when it wakes up. The executor never touches the result.

The work framing. `execute(F&&)` submits work. The callable is a unit of work. The executor runs it. If dropped, the work and its result are lost. Error handling, lifecycle, and composition are the executor's responsibility.

The structural difference is in what happens to the caller. `post` ends the caller's chain of execution. The caller returns. There is no live caller on the other end to receive a report. The continuation will be resumed later, on a context, and the result will be delivered *to* the continuation when it wakes up. Under the work framing, `execute` is a fork: the caller submits work and continues. The caller is alive, running, and expects to learn what happened. A live caller needs an error channel. A caller that has returned does not.

The work framing imposes requirements on the executor that the continuation framing does not. Under the work framing, the executor is responsible for error channels, lifecycle management, and generic composition - the three deficiencies P2464R0[1] identified. Under the continuation framing, the executor schedules a resumption, the OS performs the work, and the result is delivered to the continuation when it wakes up. A continuation carries no result because the result does not yet exist. The generalization from `dispatch(handler)` to `execute(F&&)` introduced these requirements. The coroutine executor's `dispatch(coroutine_handle<>)` and `post(coroutine_handle<>)` constrain the argument to a type with exactly two operations - `resume()` and `destroy()`. The continuation framing is enforced by the type system, not by naming convention.

---

4. The Three Criteria Under Both Framings

P0443R14[2]:

```
void execute(F&& f);
```

Coroutine executor (P4003R0[4]):

```
std::coroutine_handle<> dispatch(
    std::coroutine_handle<> h) const;

void post(std::coroutine_handle<> h) const;
```

P2464R0[1] identified three deficiencies in P0443R14[2] that made it inadequate as a foundation for async programming. P2300R10[8] addressed all three with the sender/receiver model. The coroutine executor concept did not exist in 2021. P2464R0[1] could not have evaluated it. The point of the following subsections is not that P2464R0[1] was wrong about P0443R14[2] - it is that the deficiencies it identified are properties of `execute(F&&)` under the work framing, not inherent properties of executor-based async. Under the continuation framing, the same `execute(F&&)` does not exhibit them. The

coroutine executor demonstrates this by constraining the argument type to `coroutine_handle<>`, which eliminates the conditions that create those deficiencies.

4.1 Error Channel

P2464R0[1] argued that `execute(F&&)` has no error channel at the call site, and that errors after the call are an "irrecoverable data loss."

P2464R0[1]:

> *"The only thing that knows whether the work succeeded is the same facility that accepted the work submission."*

Three terms. All three are downstream consequences of the work framing that the rename created. Under the continuation framing, each term describes something different.

1. "The work." Under the continuation framing, there is no work. The operating system performs the I/O. The callable is a continuation that resumes after the OS completes. It is not work. It is the opposite of work - it is the thing that was suspended while the work happened.
2. "Succeeded." Under the continuation framing, a continuation does not succeed or fail. It resumes. The I/O result - error code, byte count - is delivered to the continuation when it wakes up. The continuation did not produce the result. The OS did.
3. "Accepted the work submission." Under the continuation framing, no work was submitted. The caller yielded control. Kohlhoff's original API named this `dispatch`, `post`, and `defer` - continuation-scheduling primitives (P4094R0[3] Section 6). The rename to `execute` changed the name. The operation remained the same.

P2464R0[1] analyzed the renamed API under the work framing. The use cases it describes - queue full, executor shutting down - presuppose a fire-and-forget model where a caller hands off work and moves on. The original API did not operate that way.

The coroutine executor's `dispatch` and `post` take `coroutine_handle<>`, not `F&&`. `co_await` suspends the caller. The caller is not running. It does not need an error channel back because it is not alive to act on one. It will either be resumed with a result or destroyed, propagating cancellation. There is no state where the caller is alive, running, and uninformed.

P2464R0[1] said errors after submission are an "irrecoverable data loss." The error-channel question has a timeline. What exists at each stage determines what can be lost.

Trace the path of an asynchronous read in the Networking TS:

1. The caller initiates `async_read(socket, buffer, handler)`. The implementation stores the handler - a completion token - and begins the I/O. The handler means: this is how you resume me.
2. The operating system performs the read. The caller is not running. The handler is not running. The OS is doing the work.

3. The OS completes. The implementation now holds the result: an error code and a byte count. It calls `executor.dispatch(bound_handler)`, where `bound_handler` carries the result and the original handler together. In the committee's renamed API, this became `executor.execute(bound_handler)`.
4. The executor resumes the handler on the correct execution context. The handler receives the error code and the byte count.

At steps 1 and 2, there is no error information. The I/O has not completed. The operating system has not returned a result. There is no error code. There is no byte count. The only thing being passed to the executor is a resumption handle: this is how you resume me. That is all a callback is. That is all a future is. That is all a completion token is. That is all a `coroutine_handle<>` is. If the executor drops the handle at this stage, the handle carries no error information. The handle is a resumption. P2464R0[1] described information loss. At this stage, the handle contains no information. It contains a resumption.

At step 3, the result exists. The OS has completed. The result is bound into the handler before the executor sees it. The executor's job is to resume the handler on the right context. The result travels inside the handler, not alongside it.

Under routine operation, the executor resumes the continuation. The result is delivered. There is no information loss. P2464R0[1]'s concern was structural - that the API *by design* has no error channel, so routine errors are lost. Under the continuation framing, routine errors are not lost. They are delivered to the continuation when it resumes. The executor never interprets the result. It does not need an error channel because it is not the entity that acts on the result.

The remaining question is catastrophic conditions: the execution context is shutting down, the reactor is being destroyed, the system has exhausted resources. If the executor drops the bound handler under these conditions, the result is destroyed with the handler. This is real. But it is not specific to the continuation model. Every async model has the same property under catastrophic conditions:

Networking TS (continuation model):

```
async_read(socket, buffer,
    [](error_code ec, size_t n) {
        process(ec, n);
    });
// caller returns here - gone
```

If the execution context shuts down, the bound handler is destroyed. The caller has returned. The program is not hung.

Coroutine model:

```
auto [ec, n] =
    co_await socket.read_some(buffer);
// coroutine is suspended
```

If the execution context shuts down, the ownership contract requires the executor to destroy the handle. The coroutine frame is destroyed. The parent is notified via destruction propagation. The program is not hung.

Sender model:

If the execution context shuts down, the operation state is destroyed. `set_stopped` propagates cancellation. In-flight results are lost.

Model	Catastrophic drop	Initiator state	Program state
Networking TS	Handler destroyed	Gone (returned)	Not hung
Coroutine model	Handle destroyed	Suspended	Not hung (ownership contract)
Sender model	Op-state destroyed	Connected	Not hung (`set_stopped`)

Under catastrophic conditions, all three models lose in-flight results. This is not a deficiency of any one model. It is a property of catastrophic shutdown. P2464R0[1]'s "irrecoverable data loss" concern described a structural property of routine operation under the work framing - a fire-and-forget API where the caller hands off work and moves on, with no channel to learn what happened. Under the continuation framing, routine operation delivers the result to the continuation. The information loss that P2464R0[1] described does not occur during routine operation. It occurs only under catastrophic conditions that affect all models equally.

`sync_wait` is a testing and bridging utility, not a production async pattern. This comparison is limited to models where the initiator is asynchronous.

4.2 Lifecycle

P2464R0[1]:

> *"A P0443 executor is not an executor. It's a work-submitter."*

Under the continuation framing, the callback is a continuation - it is how the initiator resumes after the OS completes. The executor takes it and resumes it on a context.

> *"Trying to entertain the fantasy that the work submission and result observation are separable concepts is just wrong."*

Under the continuation framing, work submission and result observation are not separable because they are not separate things. The executor takes a continuation and resumes it on a context. The work is done by the operating system. The result is delivered *to* the continuation when it resumes - result observation is not separable from the continuation because the result

arrives inside it.

The coroutine model makes the lifecycle explicit. `post(coroutine_handle<>)` has two call-site dispositions:

- **Normal return.** Ownership transferred. The executor holds the handle and is responsible for it.
- **Exception.** Scheduling rejected. The caller still owns the handle.

After ownership is transferred, the execution context has two dispositions:

- **Resume.** The continuation runs.
- **Destroy without resuming.** The awaiting coroutine is also destroyed, propagating cancellation upward through the coroutine tree. This is analogous to `set_stopped` in P2300R10[8].

The executor holds a typed resource with a defined lifecycle. A `coroutine_handle<>` has exactly two operations: `resume()` and `destroy()`. The proposed wording specifies the ownership contract:

> ```
> std::coroutine_handle<> dispatch(
>     std::coroutine_handle<> h) const;
>
> void post(std::coroutine_handle<> h) const;
> ```
>
> *Preconditions:* `h` is a valid, suspended coroutine handle.
> *Effects:* Arranges for `h` to be resumed on the associated execution context. For `dispatch`, the implementation may instead return `h`. For `post`, the coroutine is not resumed on the calling thread before `post` returns.
> *Postconditions:* For `post`, ownership of `h` is transferred to the implementation. For `dispatch`, if `noop_coroutine()` is returned, ownership of `h` is transferred to the implementation; if `h` is returned, ownership of `h` is transferred to the caller via the return value and the implementation has no further obligation regarding `h`. When ownership is transferred to the implementation, the implementation shall either resume `h` or destroy `h`. No other disposition is permitted.
> *Returns (`dispatch` only):* `h` or `noop_coroutine()`.
> *Throws:* `std::system_error` if scheduling fails. If an exception is thrown, ownership of `h` is not transferred; the caller retains ownership.
> *[Note:* The returned handle enables symmetric transfer. The caller's `await_suspend` may return it directly. - *end note]*

The ownership contract specifies two dispositions: resume or destroy. This is a semantic requirement, not a syntactic one - the same class of constraint that `Allocator`, `Iterator`, and every other standard library concept carries alongside its syntactic checks.

4.3 Composition

P2464R0[1]:

> *"Composing Networking TS completion handlers or asynchronous operations (which aren't even a thing in the NetTS APIs, so how would I be able to compose those other than in an ad-hoc manner?) seems like an ad-hoc and non-generic exercise. Whether they were designed for or ever deployed in larger-scale compositions that scale from low-level to high-level with any sort of uniformity, I don't know."*

On the axis of algorithmic composition - `retry`, `when_all`, `upon_error`, chaining operations generically into higher-level workflows - the Networking TS had no generic mechanism. Each composed operation required a hand-written state machine. Boost.Beast (Boost 1.66, 2017) deployed three layers of composed asynchronous operations - socket reads into HTTP parsing into WebSocket framing - and every layer required its own state machine, its own intermediate completion handler, and its own lifetime management. The composition worked. It was deployed. It scaled from low-level to high-level. But it was genuinely difficult to write, difficult to review, and difficult to get right. The authors wrote those state machines and can attest to the cost.

P2464R0[1] was right that this was ad-hoc. Senders provide a generic composition algebra that the Networking TS did not have. That is a real achievement.

Coroutines address the same axis differently. `co_await` replaces the state machine. A composed read that required a hundred-line state machine in the callback model is a ten-line coroutine body. The coroutine frame holds the state. The compiler manages the suspension points. C++20 was ratified in 2020.

---

5. Outcomes

P2300R10[8] provides compile-time sender composition, structured concurrency guarantees, and a customization point model that enables heterogeneous dispatch. These are real achievements. P2464R0[1] identified real deficiencies in P0443R14[2] and the committee acted on them. The outcomes are now observable.

5.1 The Empirical Record

Criterion	P2464R0[1] claim (2021)	Predicted outcome	2026 evidence
Error channel	`execute(F&&)` has no error channel	P2300R10[8] provides `set_error`	P2430R0[11] (Kohlhoff, 2021): compound I/O results cannot use `set_error` without losing the byte count. P2762R2[12] (Kühl, 2023): five routing options documented, single-argument error channel "somewhat limiting." LEWG reflector (March 2026): no standard facility exists for runtime dispatch of compound results onto channels.
Lifecycle	`execute(F&&)` has no lifecycle for submitted work	P2300R10[8] provides structured lifecycle via sender/receiver	P3801R0[13] (2025): `execution::task` lacks symmetric transfer. P3552R3[14]: `AS-EXCEPT-PTR` converts routine `error_code` to `exception_ptr`.
Generic composition	No generic composition	P2300R10[8] provides sender algorithms	P2470R0[15] (2021): deployments at Facebook, NVIDIA, Bloomberg - GPU dispatch, thread pools, infrastructure. Seven published examples (2024): thread pools, embedded systems, cooperative multitasking, custom algorithms. None involve networking. None involve I/O.
Deployed networking	"I don't know" whether Networking TS compositions scale	P2300R10[8] replaces the Networking TS	No published paper documents production-scale networking using the sender model. N1925[16] (2005): first networking proposal. 2026: networking is not in the C++ standard.

5.2 The Symmetry Test

A constraint applied to one model applies to both.

Ecosystem-scale validation:

Property	Coroutine executor	P2300R10[8] for networking
Age	P4003R0[4] (2026). New.	P2464R0[1] redirected the committee toward P2300R10[8] in 2021. Five years.
Deployments	Capy[6], Corosio[7].	P2470R0[15]: Facebook, NVIDIA, Bloomberg - GPU dispatch, thread pools, infrastructure.
Networking deployments	New.	None published. Boost.Asio and Boost.Beast - the deployed networking that the committee set aside - used the continuation model.

The `noexcept` context. Both models can reach `std::terminate` in a `noexcept` coroutine. The difference is in the trigger condition:

Property	Coroutine executor	`execution::task` (P3552R3[14])
What throws	`post(coroutine_handle<>)` throws `std::system_error` on scheduling failure.	`AS-EXCEPT-PTR` converts routine `error_code` to `exception_ptr`.
Trigger condition	Scheduling failure on an I/O reactor.	`ECONNRESET`, `ETIMEDOUT`, `EWOULDBLOCK` - routine I/O outcomes.
In a `noexcept` context	`std::terminate` on catastrophic system condition.	`std::terminate` on routine I/O.

5.3 The Outcome

The analysis was procedurally correct. P2464R0[1] analyzed the specification as written. The three deficiencies it identified are real properties of `execute(F&&)` under the work framing. The committee evaluated the analysis, voted, and acted. The process functioned as designed.

The outcome: the Networking TS was published as an ISO TS in 2018. It was removed from the committee's work program in 2021. In 2026, no replacement has shipped.

It is natural to prioritize process. Process is visible. Process is checkable. A procedural review can verify that an analysis is correct on its own terms - that the specification says what the paper claims it says, that the deficiencies identified are real properties of the API surface. That verification happened. It was done correctly.

What the process did not check - what it has no mechanism to check - is whether the analysis examined the specification under every applicable framing. P2464R0[1] analyzed `execute(F&&)` under the work framing. The continuation framing,

documented in P0113R0[17], "Executors and Asynchronous Operations, Revision 2" (2015), and carried by institutional knowledge rather than by the API surface, was not examined. Under that framing, the three deficiencies do not arise. The process had no mechanism to require that both framings be examined before acting on the analysis.

The process also had no mechanism to check outcomes. No revisit trigger was set. No follow-up criteria were established. The committee acted on the analysis in 2021. The prediction that P2300R10[8] would replace the Networking TS for networking was not revisited against evidence. No sender-based networking has shipped.

Voutilainen analyzed what was in front of him, and what was in front of him was all the specification provided. The continuation framing had been erased from the API surface by the terminology shift documented in P4094R0[3]. The coroutine executor concept did not exist. The process gave him no reason to look elsewhere and no mechanism to revisit the outcome.

5.4 Summary

Criterion	P2464R0[1] (2021)	P2300R10[8] (2026)	Coroutine executor
Error channel	`execute(F&&)` has none	`set_error` exists; does not handle compound I/O results without information loss (P2430R0[11])	Not needed; result delivered to continuation on resume
Lifecycle	`execute(F&&)` has none	Structured lifecycle exists; `task` converts routine errors to exceptions (P3552R3[14])	Ownership contract: resume or destroy
Generic composition	`execute(F&&)` has none	Sender algorithms exist; deployed for GPU dispatch, thread pools, infrastructure	`co_await` replaces state machines
Deployed networking	"I don't know"	None published	New.

The question that P2464R0[1] asked in 2021 - whether the executor abstraction is adequate for async programming - now has two published answers instead of one. The question that the process did not ask - whether the analysis examined every applicable framing before the committee acted on it - did not have a mechanism to be asked.

Acknowledgments

The authors thank Peter Dimov for identifying that P0443R14's callable destruction is detectable, correcting an earlier version of Section 4; Dietmar Kühl for `beman::execution` and ongoing work on P3552R3; Ville Voutilainen for P2464R0, which provided the evaluation framework for this paper; Steve Gerbino and Mungo Gill for Capy and Corosio implementation work; and Klemens Morgenstern for Boost.Cobalt and the cross-library bridge examples.

References

[1] P2464R0 - "Ruminations on networking and executors" (Ville Voutilainen, 2021).

[2] P0443R14 - "A Unified Executors Proposal for C++" (Jared Hoberock, et al., 2020).

[3] P4094R0 - "Retrospective: The Unification of Executors and P0443" (Vinnie Falco, 2026).

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

[5] P4100R0 - "The Network Endeavor: Coroutine-Native I/O for C++29" (Vinnie Falco, Steve Gerbino, Michael Vandeberg, Mungo Gill, Mohammad Nejati, 2026).

[6] Capy - Coroutine I/O foundation library (Vinnie Falco).

[7] Corosio - Coroutine networking library (Vinnie Falco).

[8] P2300R10 - "std::execution" (Michał Dominiak, Lewis Baker, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach, 2024).

[9] P2469R0 - "Response to P2464 - Ruminations on networking and executors" (Vinnie Falco, Christopher Kohlhoff, 2021).

[10] P4088R0 - "What C++20 Coroutines Already Buy The Standard" (Vinnie Falco, 2026).

[11] P2430R0 - "Slides: Partial success scenarios with P2300" (Christopher Kohlhoff, 2021).

[12] P2762R2 - "Sender/Receiver Interface For Networking" (Dietmar Kühl, 2023).

[13] P3801R0 - "Concerns about the design of std::execution::task" (Jonathan Müller, 2025).

[14] P3552R3 - "Add a Coroutine Task Type" (Dietmar Kühl, Maikel Nadolski, 2025).

[15] P2470R0 - "Slides for presentation of P2300R2: std::execution (sender/receiver)" (Eric Niebler, 2021).

[16] N1925 - "Networking proposal for TR2 (rev. 1)" (Gerhard Wesp, 2005).

[17] P0113R0 - "Executors and Asynchronous Operations, Revision 2" (Christopher Kohlhoff, 2015).