Error Models of Regular C++ and the Sender Sub-Language

Document Number:	P4091R1
Date:	2026-05-01
Intent:	Inform
Audience:	LEWG

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

Table of Contents

Abstract

Both coroutines and senders destroy compound data at an abstraction floor - the difference is that the sender floor sits below the composition algebra, and the coroutine floor is opt-in.

In March 2026, Andrzej Krzemieński asked the lib-ext reflector [1]:

> *"Is there a component, or a technique, in `std::execution`, going into C++26, available today in one of the reference implementations where I could get a value via the value channel, and dispatch it to either of the three channels based on the value I get?"*

The question launched a discussion. Ville Voutilainen, Ian Petersen, Jens Maurer, and the author explored the design space across several dozen messages. This paper follows that discussion and examines its implications for how the sender three-channel model interacts with compound I/O results.

The finding: both coroutines and senders have an abstraction floor - a boundary where compound results are reduced to a single value. The coroutine floor is `throw`. The sender floor is `set_error`. Both destroy compound data when crossed. The difference is where the floor sits relative to composition. In coroutines, the floor is opt-in. In senders, the composition algebra lives above it.

Revision History

R1: May 2026 (pre-Brno mailing)

- Corrected echo-server implementation count to match P4090 (five implementations: four sender-based, one coroutine-native).
- Formatting corrections.

R0: April 2026 (post-Croydon mailing)

- Initial version.
- Reframed in response to reflector discussion with Voutilainen, Petersen, Maurer, Sutter, and Krzemieński.
- Replaced abstract with Krzemieński's reflector question and the abstraction floor finding.

- Incorporated Ian Petersen's observation that a previous draft held senders and coroutines to different standards regarding their error channels.
- Added the abstraction floor as a symmetric concept: both paradigms have one.
- Replaced the "Four Defenses, Four Concessions" framing with an honest trade-off space (Section 11).
- Added Sections 2 (The Question), 7 (The Equivalence), 8 (The Symmetry), 9 (The Boundary), 10 (The Abstraction Floor).
- Added reflector quotes from Voutilainen, Petersen, Maurer, Kohlhoff, Kühl, Shoop, and Sutter throughout, with permission.
- Added `std::execution` support disclosure paragraph.
- Added P4088R0 [2] to the companion list.

---

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++.

The author developed and maintains Capy [3] and Corosio [4] 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 asks for nothing.

---

2. The Question

Krzemieński's question was direct: given a value on the value channel, how do I route it to different channels at runtime? Voutilainen, Petersen, Maurer, and the author spent the next several days working through the answer. The conversation was technical, constructive, and occasionally funny. Each participant contributed something the others had not seen.

This paper follows that conversation. The sections that follow present the technical background, then trace the discussion's arc: from the partition that makes compound results difficult, through the constructions that address them, to the equivalence and symmetry that emerged, and finally to the abstraction floor - a concept that applies to both paradigms.

---

3. The Partition

Operations partition into two classes based on postcondition structure.

3.1 Infrastructure Operations

`malloc`, `fopen`, `pthread_create`, GPU kernel launch, thread pool submit, timer arm. The operation either executes or it does not. An error means the postcondition was violated.

Operation	Success	Failure
`malloc`	Block returned	Allocation failed
`fopen`	File handle returned	Open failed
`pthread_create`	Thread running	Creation failed
GPU kernel launch	Kernel running	Launch failed
Thread pool submit	Task queued	Pool exhausted
Timer arm	Timer armed	Resource limit
Mutex acquire	Lock held	Deadlock / timeout

Every row is binary. The postcondition was satisfied or it was not. On failure, the non-status fields carry no meaningful data: `malloc` returns NULL, `fopen` returns NULL, `pthread_create` leaves the thread ID indeterminate.

3.2 Compound-Result Operations

Read, write, accept, parse, convert. The operation completes and returns associated data that is meaningful regardless of the status classification.

Operation	Result
`read`	`(status, bytes_transferred)`
`write`	`(status, bytes_written)`
`from_chars`	`(ptr, errc)`
`recvmsg`	`(ssize_t, msg_flags)`
`accept`	`(status, peer_socket)`

The partition is not about asynchrony. It is about what the operation promises to return.

The author initially framed this challenge as specific to I/O. Voutilainen observed that the problem is far more general:

> *"The need to runtime-dispatch product types on chosen completion channels arises for all sorts of coroutines, i/o or not. For example, it comes up when senderifying Qt signals that have multiple parameters. [...] The i/o use cases are just a particular example of the general programming problem, and we will inevitably have to make handling that problem easier." - Ville Voutilainen* [5]

The partition is not about I/O. It is about product types meeting channels.

3.3 Compound Results in Practice

The compound-result pattern is how systems report outcomes when the result carries more than a boolean. io_uring delivers `(res, flags)` in one CQE. [6] IOCP delivers `(BOOL, lpNumberOfBytesTransferred, lpOverlapped)` in one call. [7] POSIX `read()` returns `ssize_t` with `errno`. [8] `std::from_chars` returns `{ptr, ec}`. Three OS families and the C++ standard library converge on the same shape: status and data arrive as a pair because they are a single result.

---

4. The Infrastructure Error Model

The sender three-channel model (P2300R10 [9]) assigns semantic meaning to each channel:

- `set_value(args...)` - the postcondition was satisfied; here are the results
- `set_error(err)` - the postcondition was violated; here is why
- `set_stopped()` - the operation was cancelled before attempting the postcondition

For infrastructure operations, this mapping is unambiguous:

```
// Timer sender: binary outcome
auto timer_sender = schedule_at(scheduler, deadline);
// Success: set_value()       - timer expired
// Failure: set_error(ec)     - resource exhaustion
// Cancel:  set_stopped()     - cancelled before expiry
```

```
// Thread pool submit: binary outcome
auto submit_sender = on(pool.get_scheduler(), work);
// Success: set_value(result) - work executed
// Failure: set_error(ec)     - pool rejected
// Cancel:  set_stopped()     - cancelled before dispatch
```

Channel assignment is deterministic. No information is lost. `then` sees the result. `upon_error` sees the error. `upon_stopped` sees the cancellation. The channels compose with generic algorithms: `retry` retries on `set_error`, `when_all` cancels siblings on failure, `upon_error` routes errors to handlers.

The three-channel model is correct for this class. `std::execution` serves it well.

Throughout this paper, *composition algebra* refers specifically to the channel-dispatching subset of `std::execution` algorithms - those whose behavior depends on which channel a result arrives on. `retry` re-executes on `set_error`. `when_all`

cancels siblings when one completes with `set_error` or `set_stopped`. `upon_error` and `upon_stopped` route to handlers by channel. Algorithms that sequence work without inspecting the channel - `then`, `let_value`, `bulk` - compose but do not dispatch. The distinction is between *sequencing* and *routing*.

---

5. The Compound-Result Error Model

The status code is not a failure signal. It is a vocabulary:

- `ECONNRESET` - the peer reset the connection
- `EPIPE` - the write end is closed
- `ETIMEDOUT` - the peer did not respond within the timeout
- `EWOULDBLOCK` - nothing available now; try again
- `EOF` - the peer closed the connection gracefully

Each requires different application handling. None indicates that the operation failed to operate. `ECONNRESET` means "fatal, abort the transaction" in an HTTP request handler, but "done, expected closure" in a long-polling connection that the server intentionally drops. The same error code is classified differently depending on the protocol state the application holds.

Partial results are normal. A `write` that sends 500 of 1,000 bytes before `ECONNRESET` produces `(ECONNRESET, 500)`. Both values are needed.

Chris Kohlhoff identified this in P2430R0 [10] ("Partial success scenarios with P2300," 2021):

> *"Due to the limitations of the `set_error` channel (which has a single 'error' argument) and `set_done` channel (which takes no arguments), partial results must be communicated down the `set_value` channel."*

Two approaches emerged in the reflector discussion: route compound results through the error channel and handle them before they reach `task`, or keep everything on the value channel. Maurer framed the choice:

> *"That's all part of the design spectrum, I'd say, and depends on your particular situation. As you correctly observe, you either need to handle all errors before they reach 'task', or decide that certain errors should 'bubble up', causing an exception, maybe because they're fatal enough to terminate the world. If you wish to handle complicated success results, your surroundings must be prepared to deal with that somehow." - Jens Maurer [5]*

Two roads. What does each cost?

---

6. The Channel Split

When a sender-based I/O operation completes, it must choose a channel. The byte count and the status code - which the OS delivered as a pair - are now split:

```
async_read(socket, buffer)
    | then([](size_t n) {
          // byte count visible
          // error_code invisible
      })
    | upon_error([](std::error_code ec) {
          // error_code visible
          // byte count invisible
      });
```

`then` and `upon_error` are mutually exclusive execution paths. The programmer cannot inspect both values at the same call site.

Voutilainen did not leave this as an observation. He sketched a concrete sender adapter - a `dispatch` algorithm that routes the result to different channels based on a runtime choice:

> *"I'm describing a sender adapter that takes a predecessor sender, a predicate function that returns a bool based on the completion result of the predecessor sender, and two sender factory functions that produce different senders based on the completion result of the predecessor sender. [...] That then allows you to decide what data to push to the chosen completion channel, and allows avoiding the problems of having to somehow drop/lose any data." - Ville Voutilainen* [5]

The adapter preserves data because both factory functions receive the full result. A natural concern is that routing data to the error channel forces the programmer to discard part of the result. Voutilainen demonstrated otherwise:

> *"You can keep the payload on both channels if you wish. You can drop it if you wish, too. But you don't have to drop it." - Ville Voutilainen* [1]

The data can survive the channel crossing. The question is whether it survives the generic algorithms downstream.

A natural response is that the `dispatch` adapter, if standardized, would close the gap within the sender model. The adapter solves the *local* problem: at the point of dispatch, both fields are visible and the programmer chooses a channel with full context. But the downstream algebra is channel-typed, not value-typed. Once `dispatch` routes `ec` to `set_error`, the byte count must travel separately - inside the error object (position 11.6), in a captured lambda (position 11.5), or not at all (position 11.1). `retry` does not receive the byte count through the channel; it receives `error_code`. The adapter moves the classification to the right layer. It does not change what the channels carry after classification.

The local problem is solved. The downstream problem remains.

6.1 Known Mappings

Two known attempts to solve the channel assignment problem for I/O illustrate the structural difficulty.

The default mapping. beman.net [11]:

```
if (!ec)
    set_value(std::move(rcvr), n);
else
    set_error(std::move(rcvr), ec);
```

All non-zero error codes go to `set_error`. The byte count is discarded unconditionally.

The refined mapping. Peter Dimov proposed a convention, documented in P4007R3 [12] (Section 3.6), that preserves partial results by discriminating the channel based on the byte count and error code category:

Completion	Channel
`(eof, n)` for any n	`set_value(eof, n)`
`(canceled, n)` for any n	`set_stopped()`
`(ec, 0)` where ec is not eof	`set_error(ec)`
`(ec, n)` where n > 0	`set_value(ec, n)`

This is the only known mapping that preserves partial results on the error path. Dimov characterized it as "ad hoc." [12] The classification is context-free: the I/O layer decides which channel before the application can inspect the result.

Both mappings demonstrate the same structural problem: any function from `(error_code, size_t)` to `{set_value, set_error, set_stopped}` must either lose data, embed domain-specific classification at the wrong layer, or bypass the channels entirely.

6.2 Prior Engagement

Dietmar Kühl enumerated five channel-routing options for I/O in P2762R2 [13] (Section 4.2) and acknowledged the partial-success problem directly: "some of the error cases may have been partial successes. In that case, using the set_error channel taking just one argument is somewhat limiting."

Kirk Shoop identified the same heuristic difficulty in P2471R1 [14], observing that completion tokens translating to senders "must use a heuristic to type-match the first arg."

To the author's knowledge, no published paper resolves the compound-result channel-routing problem identified in P2430R0 [10]. The problem has been identified by Kohlhoff (2021), Kühl (2023), and Shoop (2021). It remains open.

6.3 The P2300 Authors' Position

P2300R10 [9] addresses compound results directly in Section 4.14, "Senders can represent partial success." The authors identify two cases.

For the first case:

> *"Often in the case of partial success, the error condition is not fatal nor does it mean the API has failed to satisfy its post-conditions. It is merely an extra piece of information about the nature of the completion. In those cases, 'partial success' is another way of saying 'success'. As a result, it is sensible to pass both the error code and the result (if any) through the value channel."* [9]

This is position 11.2 (Section 11). The composition algebra does not participate.

For the second case:

> *"In other cases, the partial success is more of a partial failure. [...] It's possible that bundling the error and the incomplete results into an object and passing it through the error channel makes more sense. In that way, generic algorithms will not miss the fact that a post-condition has not been met and react inappropriately."* [9]

This is position 11.6 (Section 11). The composition algebra participates, but every `upon_error` handler in the pipeline must understand the bundled type.

An earlier paper by several of the same authors, P1525R1 [15] ("One-Way execute is a Poor Basis Operation," 2020), characterized the error channel's intended scope:

> *"The `set_error()` channel of a receiver, like C++ exceptions, is for exceptional circumstances: things like dropped network connections, resource allocation failure, or inability to create an execution agents."* [15]

Section 5 of this paper observes that `ECONNRESET` - a dropped network connection - means "fatal, abort the transaction" in one protocol and "done, expected closure" in another.

---

7. The Equivalence

In the same reflector discussion, Petersen constructed four working sender implementations [16] that dispatch compound I/O results onto different channels: one using `exec::variant_sender`, one using `exec::any_sender_of`, one using `std::execution::task`, and one using a custom sender. Each implements a different policy for splitting `(error_code, buffer)` into the three channels.

The author asked whether all four were equivalent to:

```
auto [ec, buf] = co_await read(socket, buffer);
switch (ec) { ... }
```

Petersen confirmed:

> *"In one word, yes. In more words, there's some nuance but basically yes." - Ian Petersen [1]*

The nuance matters. The implementations are equivalent in result - both paradigms handle compound results as values. The coroutine destructures a pair and branches with `switch`. The sender pipes `set_value(ec, buffer)` into `let_value` and branches with `if`. The dispatch logic is identical. But the four sender implementations each use a different construction (`variant_sender`, `any_sender_of`, `execution::task`, custom sender), and a programmer encountering the problem for the first time must evaluate all four before choosing. The coroutine version has one construction. The equivalence is in what the code does, not in what the programmer must know to write it.

This equivalence raises a question the discussion had not yet asked.

---

8. The Symmetry

Petersen read the companion papers and identified a structural asymmetry in the argument. An earlier draft of this paper argued that the sender error channel (`set_error`) cannot carry compound results without data loss. True. But the coroutine examples in that draft and its companions never use the coroutine error channel (`throw`). They keep compound results as values: `co_return pair{ec, n}`. If senders do the same - `set_value(ec, n)` - the structural symmetry is exact.

> *"If we're going to compare and contrast coroutines with senders, we should compare and contrast them on equal footing. Your coroutine examples don't use the coroutine error channel, but your assertions about the deficiencies of senders insist that sender code must use the sender error channel. I haven't been able to figure out why the two paradigms should be held to different standards like this." - Ian Petersen [1]*

The observation is correct. This revision accounts for it.

The coroutine error channel is `throw`. The sender error channel is `set_error`. Both destroy compound data. The industry advice - use the value channel for compound results - applies to both paradigms equally. The coroutine examples in this paper and its companions follow that advice. The question is whether senders should follow it too, and what is lost when they do.

When a coroutine keeps `(ec, n)` on the value channel, the programmer has `if`, `switch`, `for`, and every other C++ statement available for dispatch. When a sender keeps `(ec, n)` on the value channel, the programmer has `let_value`. The generic algorithms - `retry`, `when_all`, `upon_error` - do not participate. They were designed for channel-based dispatch, not value-based dispatch.

---

9. The Boundary

When the discussion turned to whether the byte count could remain visible to generic algorithms like `retry` during their execution - not just before and after - Voutilainen gave a precise characterization:

> *"Certainly, if the successor can eat a function object, and you then capture the count in that function object. Or use some actual shared state. [...] The short answer is, 'yes, intrusively.' Not fully-generically." - Ville Voutilainen* [5]

The data can be preserved. The generic algorithms can participate. But the connection between them requires application-specific wiring - a lambda capture, shared state, or a function object that carries the byte count alongside the error code. The generic algorithms do not see the byte count through the channel. They see it through a side channel the programmer constructs.

For compound-result I/O, the application-specific wiring point arrives at every operation. P4090R0 [17] provides the concrete measurement. Five echo-server implementations - four sender-based constructions and one coroutine-native - implement identical protocol logic. The sender constructions require between 2x and 3.5x the line count of the coroutine construction, with the additional lines concentrated in channel-routing and type-erasure machinery.

Domain	Sync/Async	Error model	Three-channel model fits?
`malloc`	Sync	Infrastructure	Yes
`fopen`	Sync	Infrastructure	Yes
GPU dispatch	Async	Infrastructure	Yes
Thread pool	Async	Infrastructure	Yes
Timer	Async	Infrastructure	Yes
Connect	Either	Infrastructure	Yes
`from_chars`	Sync	Compound-result	Via `let_value`
`recvmsg`	Either	Compound-result	Via `let_value`
Read / Write	Either	Compound-result	Via `let_value`
Accept	Either	Compound-result	Via `let_value`
DNS resolve	Either	Compound-result	Via `let_value`

The "Via `let_value`" entries work. The compound result stays intact inside the `let_value` handler. The composition algebra does not see it.

---

10. The Abstraction Floor

The abstraction floor is the boundary where compound results are reduced to a single value - where one field of the result is destroyed because the path can only carry one thing.

Both paradigms have one.

The coroutine floor is `throw`. Below it: `auto [ec, n] = co_await read(...)` - both values visible, dispatch with `if`/`switch`. Above it: `catch (std::system_error const& e)` - the byte count is gone.

The sender floor is `set_error`. Below it: `set_value(ec, n)` piped into `let_value` - both values visible, dispatch with `if`. Above it: `upon_error([](error_code ec) {...})` - the byte count is gone.

Paradigm	Floor location	Below the floor	Above the floor
Coroutine	`throw`	`(ec, n)` both visible	exception alone
Sender	`set_error`	`(ec, n)` both visible	`error_code` alone

The floor is not a sender problem or a coroutine problem. It is a property of any system with mutually exclusive error and value paths. Giving it a name helps us see it. Being honest about where it sits helps us design around it.

The floors are structurally analogous but operationally opposite. Coroutines default to below-floor composition: the programmer stays below the floor unless an explicit `throw` crosses it. Below the floor, the programmer has `if`, `switch`, `for`, and every other C++ statement for dispatch. Senders default to above-floor composition: the composition algebra - `retry`, `when_all`, `upon_error` - lives above the floor. To use the algebra, the result must cross the floor. To preserve compound data, the result must stay below it. The programmer cannot do both simultaneously without application-specific wiring (Section 9).

Any async facility that separates error and value paths has an abstraction floor. Identifying where it sits relative to composition is a design decision, not an accident.

P4093R0 [18] uses the abstraction floor as a design constraint. P4090R0 [17] shows the floor in each of four echo server constructions.

---

11. The Trade-Off Space

Six positions are available for handling compound I/O results in a sender pipeline. Each is a legitimate design choice. Each trades something. The first three sacrifice one property cleanly. The last three attempt to preserve all three and pay in complexity.

11.1 Accept the Information Loss

Route errors through `set_error`, discard the byte count. Composition works. Data is lost. POSIX `write()` returns `ssize_t` with `errno`. Asio completion handlers receive `(error_code, size_t)`. Both preserve the byte count alongside the status.

11.2 Keep Everything on the Value Channel

Call `set_value(error_code, bytes)` for all outcomes. The pair stays intact. `set_error` and `set_stopped` serve no purpose for I/O senders. `retry`, `when_all`, `upon_error` do not participate. The composition algebra is bypassed. This is the sender equivalent of what coroutines do when they avoid `throw` - and it is a legitimate choice.

11.3 Classify Errors Into Channels

Route "routine" errors through `set_value` and "exceptional" errors through `set_error`. This requires a taxonomy that does not exist in any OS API and forces the classification at the I/O layer - before the application has seen the result. `ECONNRESET` is fatal in one protocol and expected in another.

11.4 Decompose With `let_value`

Route everything through `set_value(error_code, bytes)`, pipe into `let_value`, inspect both values, re-route the error code to `set_error`. The handler sees both values. Classification happens with full application context. But `set_error` takes a single argument. The byte count is destroyed when the error code crosses the floor.

11.5 Preserve Data With Application-Specific Wiring

Keep the full tuple on the error channel, or capture the byte count in a lambda and restore it after the successor runs. The data survives. The generic algorithms participate. The wiring is intrusive and not fully generic (Section 9).

11.6 Bundle Compound Data Into the Error Type

Call `set_error(io_result{ec, n})` where `io_result` carries both the error code and the byte count. The composition algebra participates: `retry` sees the error, `upon_error` sees the error. The data survives the floor crossing because it is inside the error object. P2300R10 [9] suggests this approach for partial failure (Section 6.3). The cost: every `upon_error` handler in the pipeline must accept `io_result` as a variant alongside `error_code`, `exception_ptr`, and any other error type in the completion signatures. `retry` must understand `io_result`. The error type proliferates across the pipeline. No published library implements this convention.

11.7 The Observable Trade-Offs

Position	Data preserved	Composition algebra	Generic	Floor crossed
Accept the loss (11.1)	No	Yes	Yes	Yes
Value channel only (11.2)	Yes	No	Yes	No
Classify errors (11.3)	Partial	Partial	No	Partial
Decompose (11.4)	Yes (below floor)	Yes (above floor)	Yes	Yes
Application wiring (11.5)	Yes	Yes	No	Yes
Bundle into error type (11.6)	Yes	Yes	Partial	Yes

Not all compound-result operations are equally affected. Operations without partial-success semantics - `accept()`, where the peer socket is meaningless on failure - degrade naturally to the three-channel model: the error routes through `set_error` with no data loss because there is no meaningful data to preserve. The compound-result tension is sharpest for operations like `read()` and `write()`, where the byte count carries meaningful data on every path, including failure. A design that handles the degradation case well does not automatically handle the partial-success case. Peter Dimov observed in the reflector

discussion cited in Section 6.2 that the mapping "should degrade to the natural mapping when no partial successes are present," and that the value-channel-only approach (11.2) does not achieve this. An operation-level classification based on the presence of partial-success semantics would reduce the trade-off space for operations that lack it, while preserving the full six-position analysis for operations where partial success is real.

---

12. Anticipated Objections

Q: The analysis compares coroutine sequential composition to sender concurrent composition. Is that fair?

Compound results arise at individual I/O operations, which are sequential by nature - a single `read` returns one `(ec, n)`. The comparison is fair for this problem. Coroutine structured concurrency - fan-out, join, cancellation propagation - requires its own framework (`when_all` over tasks, nurseries, task groups), and those frameworks face analogous routing decisions. The coroutine advantage for concurrent composition is smaller, and this paper does not claim otherwise.

Q: The `dispatch` adapter (Section 6) would close the gap if standardized.

Section 6 documents that the adapter solves the local problem: at the point of dispatch, both fields are visible. The downstream algebra remains channel-typed. Once `dispatch` routes `ec` to `set_error`, the byte count must travel separately - inside the error object (position 11.6), in a captured lambda (position 11.5), or not at all (position 11.1).

Q: The three-channel model serves a real need.

Section 4 attests to this. The three-channel model is correct for infrastructure operations. The paper does not argue that the model is useless. It documents the structural consequence when compound results meet channel-based dispatch.

---

13. Conclusion

The partition is real. The abstraction floor exists in both paradigms. Senders can carry compound results - through `set_value`. Coroutines can discard them - through `throw`. The composition algebra that distinguishes senders from coroutines lives above the floor. Compound I/O results live below it. Any system that separates error and value paths inherits this tension. The question is not which paradigm is correct, but where the floor belongs for the domain at hand.

---

14. Acknowledgments

The author thanks Andrzej Krzemieński for the question that launched the discussion; Ville Voutilainen for broadening the problem beyond I/O, constructing the dispatch adapter, demonstrating data preservation on both channels, and characterizing the boundary between generic and application-specific composition with precision; Ian Petersen for four working sender implementations, for confirming the equivalence between sender and coroutine dispatch, and for identifying the symmetry between coroutine and sender error channels that prompted this revision; and Jens Maurer for framing the design spectrum.

The author also thanks Chris Kohlhoff for identifying the partial-success problem in P2430R0 [10], Dietmar Kühl for the channel-routing enumeration in P2762R2 [13] and for `beman::execution`, Kirk Shoop for the completion-token heuristic analysis in P2471R1 [14], Fabio Fracassi for P3570R2 [19], Peter Dimov for the refined channel mapping, Michał Dominiak, Eric Niebler, and Lewis Baker for `std::execution`, Maikel Nadolski for work on `execution::task`, and Steve Gerbino for co-developing the constructed comparison.

Any quoted participant who wishes a passage retracted or revised may contact the author, who is happy to edit.

References

[1] Lib-ext reflector, "std::execution -- dynamically selecting a channel," March 2026

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

[3] cppalliance/capy - Coroutine I/O primitives library.

[4] cppalliance/corosio - Coroutine-native networking library.

[5] Lib-ext reflector, "Complicated success at coroutine/sender composition boundaries (from SG14 Mar 11)," March 2026

[6] io_uring completion queue entry (`struct io_uring_cqe`), Linux kernel 5.1+

[7] `GetQueuedCompletionStatus`, Windows I/O Completion Ports

[8] IEEE Std 1003.1-2024 - POSIX `read()` / `write()` specification

[9] P2300R10 - "std::execution" (Michał Dominiak et al., 2024).

[10] P2430R0 - "Partial success scenarios with P2300" (Chris Kohlhoff, 2021).

[11] bemanproject/net - Sender/receiver networking library.

[12] P4007R3 - "Open Issues in `std::execution::task`" (Vinnie Falco, Mungo Gill, 2026).

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

[14] P2471R1 - "NetTS, ASIO and Sender Library Design Comparison" (Kirk Shoop, 2021).

[15] P1525R1 - "One-Way execute is a Poor Basis Operation" (Eric Niebler, Kirk Shoop, Lewis Baker, Lee Howes et al., 2020).

[16] Ian Petersen, four sender implementations of channel dispatch, March 2026 (accessed 2026-03-15)

[17] P4090R0 - "Sender I/O: A Constructed Comparison" (Vinnie Falco, Steve Gerbino, 2026).

[18] P4093R0 - "Producing Senders from Coroutine-Native Code" (Vinnie Falco, Steve Gerbino, 2026).

[19] P3570R2 - "Optional variants in sender/receiver" (Fabio Fracassi, 2025).