WG21 PROPOSAL

Are Senders Replacing Coroutines?

Document Number:	D4007R0
Date:	2026-02-15
Audience:	SG1, LEWG
Reply-to:	Vinnie Falco vinnie.falco@gmail.com Mungo Gill mungo.gill@me.com

Table of Contents

Abstract

Three structural gaps exist in the coroutine integration of `std::execution`: the three-channel completion model makes correct I/O composition impossible, the standard's own task type introduces risky new syntax to signal an error, and coroutines are denied the allocator that senders receive. In each case, coroutines bear a cost that

sender pipelines avoid. Since approval for C++26, the committee has processed 50 modifications to `std::execution` in 22 months at an accelerating rate - and every one originates in sender/receiver machinery or in the cost of making sender/receiver work with coroutines. Not a single item represents a coroutine-intrinsic problem. Taken together, the evidence reveals a recurring pattern: a design conceived for sender pipelines that disadvantages coroutines at every turn. Freezing this API without addressing these gaps risks permanent harm to the programming model most C++ developers will use.

---

1. Introduction

1.1 The Value of Coroutines

Coroutines let developers write asynchronous code that reads like sequential code:

```
task<void> handle_request(tcp::socket sock)
{
    auto [ec, n] = co_await sock.async_read(buf);
    if (ec) co_return;
    auto doc = co_await parse_json(buf);
    co_await sock.async_write(build_response(doc));
}
```

Coroutines are the programming model most C++ developers will reach for when writing asynchronous code. Concurrent programming exhibits irreducible complexity: thread safety, cancellation, lifetime across suspension points. Coroutines are the vehicle C++ chose to make concurrency as accessible as possible while acknowledging that complexity. For most developers they are not merely a preference, they are the only asynchronous programming model within practical reach.

Coroutines inhabit the same syntactic world as the rest of C++. The `auto [ec, n] = co_await ...` line uses structured bindings (C++17) naturally. The result is a value; it works with `if` with initializer, range-for, and every other direct-style feature the language committee has invested in over the last decade. Coroutines do not require a parallel syntax - they participate in the one C++ already has.

1.2 The Value of Sender/Receiver

Sender/receiver brings a different kind of power to C++. Where coroutines optimize for the programmer's experience, sender/receiver optimizes for the compiler's visibility. A sender pipeline is a lazy description of work - nothing executes until `connect` and `start` are called. The entire work graph is available as a compile-time data structure. Sender/receiver pipelines are continuation-passing style (CPS) expressed as composable value types - a technique with deep roots in the theory of computation. These properties - compile-time work graph construction, zero-allocation steady-state operation, full visibility into the continuation structure - make sender/receiver a powerful tool for domains where those properties matter most: high-frequency trading, quantitative finance, GPU dispatch, and heterogeneous computing.

Here is what this looks like in practice - a tree search from the stdexec examples repository:

```
auto search_tree(auto                    test,
                 tree::NodePtr<int>      tree,
                 stdexec::scheduler auto sch,
                 any_node_sender&&       fail)  // the continuation
    -> any_node_sender {                        // returns a monadic value
    if (tree == nullptr) {
        return std::move(fail);                 // invoke the continuation
    }
    if (test(tree)) {
        return stdexec::just(tree);             // pure/return: lift into context
    }
    return stdexec::on(sch, stdexec::just())    // schedule on executor
      | stdexec::let_value(                     // monadic bind (>>=)
            [=, fail = std::move(fail)]() mutable {
                return search_tree(             // recursive Kleisli composition
                    test, tree->left(), sch,
                    stdexec::on(sch, stdexec::just())
                      | stdexec::let_value(
                            [=, fail = std::move(fail)]() mutable {
                                return search_tree(
                                    test, tree->right(), sch,
                                    std::move(fail));
                            }));
            });
}
```

Herb Sutter, Technical Fellow at Citadel Securities, called `std::execution` ***"the biggest usability improvement yet to use the coroutine support we already have"*** and reported that his firm already uses it in production: ***"We already use C++26's*** `std::execution` ***in production for an entire asset class, and as the foundation of our new messaging infrastructure."***

Note the syntactic divergence. Structured bindings, `if` with initializer, range-for over results - the direct-style features C++ has been building for a decade - have no purchase on a sender pipeline. Values flow forward into continuations as arguments, not backward to callers as returns. There is no aggregate to destructure at intermediate stages. Sender pipelines occupy a parallel syntactic world where these language features do not apply.

SG4 polled at Kona (November 2023) on P2762R2 ("Sender/Receiver Interface For Networking"):

> *"Networking should support only a sender/receiver model for asynchronous operations; the Networking TS's executor model should be removed"*
>
> | *SF* | *F* | *N* | *A* | *SA* |
> |---|---|---|---|---|
> | *5* | *5* | *1* | *0* | *1* |
>
> *Consensus.*

This means every future networking operation in the C++ standard must be a sender. Coroutines access those operations through `std::execution::task` - the integration layer documented in P3552R3. If that integration layer has structural gaps, every C++ developer who writes networked code with coroutines will encounter them.

The question this paper asks is not whether sender/receiver has value. It does. The question is whether coroutines - the programming model that most C++ developers will actually use, and the one that inhabits the same syntactic world as the rest of C++ - receive the same level of support from `std::execution`.

---

2. Where Does the Error Code Go?

P2300R10 provides three completion channels: `set_value`, `set_error`, and `set_stopped`. Only one can be called for a given operation - the outcome is either a value, an error, or a cancellation. The authors describe the model in Section 1.3.1:

> *"A sender describes asynchronous work and sends a signal (value, error, or stopped) to some recipient(s) when that work completes."*

The design priorities in Section 1.2 state:

> *"Errors must be propagated, but error handling must not present a burden."*
>
> *"Support cancellation, which is not an error."*

These are clear principles. The three channels implement them: values for success, errors for failure, stopped for cancellation. In a sender pipeline, the channels route through the continuation structure:

```
auto sndr = just(std::move(socket))                           // pure/return
          | let_value([](tcp_socket& s) {                      // monadic bind (>>=)
                return async_read(s, buf)                      // Kleisli arrow
                    | then([](auto data) {                     // fmap/functor lift
                          return parse(data);                  // VALUE -> success path
                      });
            })
          | upon_error([](auto e) {                            // ERROR -> error path
                log(e);
            })
          | upon_stopped([] {                                  // STOPPED -> cancellation path
                log("cancelled");
            });
auto op = connect(std::move(sndr), rcvr);                     // reify continuation
start(op);                                                    // begin execution
```

2.1 Senders

Every I/O operation in Boost.Asio completes with `void(error_code, size_t)` - an error code and a byte count, delivered together:

```
void(boost::system::error_code ec, std::size_t bytes_transferred)
```

In a coroutine, the caller receives both values naturally:

```
auto [ec, n] = co_await socket.async_read(buf);
```

A `read` may transfer 47 bytes and then encounter EOF. A `write` may transfer 1000 bytes out of 4096 before the connection resets. The error code and the byte count are not alternatives - they are returned together, because partial success is the normal case in I/O. P2762R2 (“Sender/Receiver Interface for Networking”) preserves this pattern: Dietmar Kuhl’s networking sender completes with `set_value(receiver, error_code, size_t)`.

A developer implementing an async read sender must write an operation state whose `start()` initiates the I/O and whose completion handler signals the receiver. The three channels present a forced choice. The natural instinct is to route the error code through `set_error`:

```
template<class Receiver>
struct read_op
{
    tcp_socket& socket_;
    buffer buf_;
    Receiver rcvr_;

    void start() & noexcept
    {
        socket_.async_read(buf_,
            [this](std::error_code ec, std::size_t n) {
                if (!ec)
                    set_value(std::move(rcvr_), n);
                else
                    set_error(std::move(rcvr_), ec); // bytes lost :(
            });
    }
};
```

But this loses the byte count on error. A `read` that transferred 47 bytes before EOF is now indistinguishable from a `read` that transferred zero. Partial success - the normal case in I/O - becomes inexpressible.

The alternative is to put the error code into `set_value`, delivering both values together as P2762R2 does:

```
socket_.async_read(buf_,
    [this](std::error_code ec, std::size_t n) {
        set_value(std::move(rcvr_), ec, n);
    });
```

This preserves the byte count. It matches twenty years of Asio practice. But now the developer is reporting an error through the success channel. The name says "value." They are sending an error code. Chris Kohlhoff identified this tension in 2021 in P2430R0 ("Partial success scenarios with P2300"):

> *"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."*

The naming is misleading. `set_error` looks right for an `error_code`, and for non-I/O senders it probably is. But for I/O - the domain with the broadest user base - it is a trap. The developer who follows the API's own naming convention produces code that silently misbehaves under composition. The developer who does the right thing must override that naming convention and put an error into the value channel. There is no guidance in the standard for which choice to make. Every I/O library author must independently discover and resolve this tension, and the ecosystem has no mechanism to converge on an answer.

2.2 Coroutines

The same forced choice reaches coroutine authors through `std::execution::task`. P3552R3 provides two mechanisms for completing a `task`:

- `co_return value` - the operand becomes the argument to `set_value` on the receiver
- `co_yield with_error(e)` - suspends the coroutine and calls `set_error` on the receiver

An unhandled exception is caught by `unhandled_exception()` and also routes to `set_error`. A coroutine author performing I/O must choose between three return types, each with consequences:

Option A: Return both values together.

```
std::execution::task<std::pair<std::error_code, std::size_t>>
do_read(tcp_socket& s, buffer& buf)
{
    auto [ec, n] = co_await s.async_read(buf);
    co_return {ec, n};
}
```

The error code and byte count are delivered through `set_value` as a pair. Partial success is expressible. But the error is invisible to `when_all`, `upon_error`, `let_error`, and every generic error-handling algorithm in `std::execution`. A `retry_on_error` adaptor will never fire, because the error never reaches the error channel.

Option B: Signal the error through the error channel.

```
std::execution::task<std::size_t>
do_read(tcp_socket& s, buffer& buf)
{
    auto [ec, n] = co_await s.async_read(buf);
    if (ec)
        co_yield with_error(ec);                      // bytes lost :(
    co_return n;
}
```

The byte count is lost when `co_yield with_error(ec)` fires. A `read` that transferred 47 bytes before EOF is now indistinguishable from a `read` that transferred zero.

Option C: Return only the byte count, discard the error.

```
std::execution::task<std::size_t>
do_read(tcp_socket& s, buffer& buf)
{
    auto [ec, n] = co_await s.async_read(buf);
    co_return n;                                      // error silently discarded
}
```

The error code is lost entirely. This is obviously wrong, but the return type `task<std::size_t>` provides no compile-time indication that an error code was discarded.

None of these options is correct. Option A hides errors from the framework's own error-handling algorithms. Option B discards partial results. Option C discards the error entirely. The coroutine author must choose which kind of wrong they prefer - the same impossible choice that raw sender authors face, expressed through `co_return` and `co_yield with_error` instead of `set_value` and `set_error`.

2.3 `when_all` Is Broken for Coroutines

`when_all` is a sender algorithm in P2300R10 that launches multiple child operations concurrently and waits for all of them to complete. If any child completes with `set_error` or `set_stopped`, `when_all` requests cancellation of the remaining children and propagates the error or stopped signal. The channel the sender author chose now determines the behavior of every algorithm that consumes it. Consider two concurrent reads:

```
auto result = co_await when_all(
    socket_a.async_read(buf_a),
    socket_b.async_read(buf_b));
```

The developer is presented with an impossible choice. Under `when_all`, no convention produces correct behavior:

If the sender author chose `set_error` for EOF, `when_all` cancels `socket_b` and propagates the error. The 47 bytes already transferred are discarded. Data loss.

If the sender author chose `set_value` for EOF, `when_all` treats the error as success. The sibling is not cancelled. A failure goes undetected.

There is no third option. The developer chooses which kind of wrong they prefer: lose the data, or lose the error.

2.3.1 Composition Is Broken for Coroutines

The problem is sharpest when two libraries are composed in the same `when_all`. Suppose Library A follows the P2762R2 convention (error in `set_value`) and Library B follows P2300R10's own naming convention (error in `set_error`, as the channel name suggests):

```
// Library A (P2762R2 convention): EOF delivered through set_value
auto read_a = lib_a::async_read(socket_a, buf_a);
// completes: set_value(receiver, error_code::eof, 47)

// Library B (P2300 naming convention): EOF delivered through set_error
auto read_b = lib_b::async_read(socket_b, buf_b);
// completes: set_error(receiver, error_code::eof)
```

```
auto result = co_await when_all(read_a, read_b);
```

This is worse than the single-library case. Previously, the developer could choose which kind of wrong they prefer. The mixed case removes even that choice: the outcome is random.

If `read_b` hits EOF first: `when_all` sees `set_error`, cancels `read_a`, propagates the error. The 47 bytes already transferred are discarded. Data loss.

If `read_a` hits EOF first: `when_all` sees `set_value`, keeps waiting for `read_b`. No cancellation. The failure goes undetected until `read_b` eventually completes.

Same two EOF events. Same `when_all`. The program is correct on some runs and incorrect on others, determined by which socket completes first.

2.4 The Problem Is Unfixable

This is not a missing convention waiting to be supplied. It is a structural mismatch between the three-channel model and I/O completion semantics. No convention, no adaptor, and no additional algorithm can resolve it without changing the model:

- **Converge on `set_value` for error codes** (the Asio/P2762R2 approach). Every generic error-handling algorithm in `std::execution` - `upon_error`, `let_error`, any future `retry` - is dead code for I/O. The error channel exists but I/O never uses it. Half the framework's error handling machinery becomes vestigial for the largest async use case.

- **Converge on `set_error` for error codes.** Partial success is inexpressible. The byte count is lost. Twenty years of I/O practice says that is wrong. A `retry_on_error` algorithm that intercepts `set_error` will retry, but it cannot know how many bytes were already transferred - that information was discarded at the channel boundary.

- **Write an adaptor that converts between conventions.** The adaptor must know which convention the inner sender follows. This reintroduces the same forced choice one level up.

- **Add a fourth channel for partial success.** This would be a breaking change to the completion signature model that `std::execution` is built on.

The three-channel model assumes a clean separation: values are values, errors are errors. I/O does not have a clean separation. An `error_code` is a status report. EOF is not a failure - it is information. A partial write is not an error - it is a progress report. The model and the domain are structurally incompatible. Appendix A.4 explains why: the three channels are a structural requirement of the compile-time work graph, and eliminating them would collapse the sender pipeline's type-level routing into runtime dispatch.

Kohlhoff identified the channel routing tension in P2430R0 (2021) but did not analyze the composition consequences. What is new in this paper is the `when_all` proof: regardless of which convention a sender author chooses, correct behavior under `when_all` is impossible by design. When two libraries choose differently, the result is a race condition over correctness determined by completion order. This is not the ambiguity Kohlhoff identified - it is a concrete correctness defect that follows from it.

2.5 I/O Should Adapt

One might argue that the channels are generic and I/O should simply adopt the `set_value` convention - treating error codes as domain values rather than framework errors.

Here is what that costs. Every error-handling sender algorithm - `upon_error`, `let_error`, any future `retry` - becomes dead code for I/O. And `when_all` still breaks: when every I/O sender delivers `(error_code, size_t)` through `set_value`, I/O failure is indistinguishable from I/O success. Sibling cancellation on failure is impossible. All current and future sender algorithms that depend on channel routing are unusable for a domain that `std::execution` claims to universally serve.

The completion signature `void(error_code, size_t)` predates senders by 25 years. It is not a quirk of one library. It is the completion signature of every I/O operation in POSIX, Win32, Asio, and every networking library built on them. The three-channel model is optimized for type-level routing in sender pipelines (Appendix A.4 examines the design rationale). The friction with I/O completion semantics is a consequence of that optimization.

Note that `upon_error` cannot recover from an error; it transforms the error but stays in the error channel. Only `let_error` can switch from the error channel back to the value channel, because the callable returns a new sender that may complete with `set_value`. As of this writing, neither P2300R10, cppreference, nor the stdexec repository contains a published example of `let_error` being used to recover from an error.

2.6 An Unforced Error?

Kohlhoff published P2430R0 in August 2021, during the most intensive review period for P2300. He demonstrated that partial results must go through `set_value` and that the channel model does not accommodate I/O completion semantics. The paper was targeted at LEWG and SG1. The authors of this paper were unable to find minutes or polls addressing P2430R0 in the published committee record. If the paper was reviewed, we would welcome a reference to the proceedings.

Are coroutines paying for a feature they did not ask for?

---

3. The `co_yield` Workaround

When an asynchronous operation fails, the framework must deliver an error to the caller. The sender/receiver protocol and coroutines provide different mechanisms for this - and the gap between them is stark.

3.1 Senders

In a sender's operation state, signaling an error is one line:

```
set_error(std::move(rcvr), ec);
```

The call is direct. The name matches the intent. No special syntax is required.

3.2 Coroutines

Returning errors from a coroutine is customarily accomplished through `co_return`:

```
my_task<std::expected<std::size_t, std::error_code>>
do_read(tcp_socket& s, buffer& buf)
{
    auto [ec, n] = co_await s.async_read(buf);
    if (ec)
        co_return std::unexpected(ec);     // return the error
    co_return n;                           // return the value
}
```

Yet P3552R3 innovates:

```
std::execution::task<std::size_t>
do_read(tcp_socket& s, buffer& buf)
{
    auto [ec, n] = co_await s.async_read(buf);
    if (ec)
        co_yield with_error(ec);           // not co_return
        // never gets here?
    co_return n;
}
```

3.3 Established Practice

No production C++ coroutine library uses `co_yield` for error signaling:

- **cppcoro** (Lewis Baker): errors are exceptions or values. `co_return` delivers both.
- **folly::coro::Task** (Facebook/Meta): errors are exceptions, propagated automatically through the coroutine chain via `co_return` and `co_await`.
- **Boost.Cobalt** (Klemens Morgenstern): errors are exceptions or values. `co_return` delivers both.
- **Boost.Asio awaitable** (Chris Kohlhoff): `co_return` for values; errors delivered via `as_tuple(use_awaitable)` as return values.
- **libcoro** (Josh Baldwin): errors are exceptions or values. `co_return` delivers both.

The established convention is universal: errors are either exceptions (caught by `unhandled_exception`) or values (delivered by `co_return`). There is no third path.

`std::execution::task` introduces one. The sender/receiver protocol requires errors to reach the receiver through a separate channel (`set_error`), not through `set_value`. The C++ language specification defines `co_return expr` as calling `promise.return_value(expr)` ([dcl.fct.def.coroutine]/7), and P3552R3's `return_value` routes to `set_value`. There is no way to make `co_return` call `set_error` on the receiver. The reason is a language constraint: a coroutine promise can only define `return_void` or `return_value`, but not both ([dcl.fct.def.coroutine]/6). Since `std::execution::task<void>` needs `co_return;` (which requires `return_void`), a `task<void>` coroutine cannot also have `return_value` - and therefore cannot use `co_return with_error(e)`.

P3552R3 needed a different mechanism to reach `set_error`. The only other coroutine keyword that accepts an expression and passes it to the promise is `co_yield`. The language defines `co_yield expr` as equivalent to `co_await promise.yield_value(expr)` ([expr.yield]/1). P3552R3 exploits this by providing a special `yield_value` overload for `with_error<E>` ([task.promise]/7):

> *Returns: An awaitable object of unspecified type whose member functions arrange for the calling coroutine to be suspended and then completes the asynchronous operation associated with STATE(*this) by invoking `set_error(std::move(RCVR(*this)), Cerr(std::move(err.error)))`.*

`co_yield with_error(e)` is interesting - a creative use of `co_yield` to reach a code path that the coroutine language feature does not natively support. In every other coroutine context - generators, async generators, async streams - `co_yield` means "produce a value and suspend, then resume when the consumer is ready." A developer reading `co_yield with_error(ec)` would reasonably expect the coroutine to continue after the yield point. It does not. The coroutine is suspended and then destroyed. The semantic meaning of `co_yield` is inverted: instead of "produce and continue," it means "fail and terminate."

The committee is aware of this problem. Jonathan Wakely's P3801R0 ("Concerns about the design of `std::execution::task`," 2025) identifies `co_yield with_error(x)` as "clunky" and explains:

> *"The reason `co_yield` is used, is that a coroutine promise can only specify `return_void` or `return_value`, but not both. If we want to allow `co_return;`, we cannot have `co_return with_error(error_code);`. This is unfortunate, but could be fixed by changing the language to drop that restriction."*

The proposed fix - a language change to the `return_void` / `return_value` mutual exclusion - is not part of C++26 and has no paper proposing it. The syntax ships as-is.

This is not mere unfamiliarity. Unfamiliar syntax is learnable - `co_await` was unfamiliar in 2020 and developers adapted. The problem with `co_yield with_error(e)` is that it actively misleads. A developer who has learned what `co_yield` means - produce a value and continue - will read `co_yield with_error(ec)` and conclude that the coroutine continues after the yield point. It does not. The keyword's established meaning predicts the wrong behavior. Every other coroutine library on GitHub uses `co_return` for both values and errors. `std::execution::task` is the only coroutine type in the C++ ecosystem that requires `co_yield` to signal an error - not because it is a better design, but because the three-channel sender/receiver model requires a path that the C++ coroutine language does not provide.

Does **`co_yield with_error`** serve coroutines or senders?

---

4. Where Is the Allocator?

Everyone has concerns about the frame allocation that I/O coroutines cannot avoid. They are right - HALO cannot elide frames that outlive their callers, and the coroutine's implementation is often not visible (see Appendix A.1). Each `task<T>` call performs `operator new`. In a server handling thousands of connections at high request rates, frame allocations can reach the order of millions per second. A recycling allocator tuned for coroutine frames can outperform even state-of-the-art general-purpose allocators. P4003R0 benchmarks a 4-deep coroutine call chain (2 million iterations):

Platform	Allocator	Time (ms)	vs std::allocator
MSVC	Recycling	1265.2	+210.4%
MSVC	mimalloc	1622.2	+142.1%
MSVC	`std::allocator`	3926.9	-
Apple clang	Recycling	2297.08	+55.2%
Apple clang	`std::allocator`	3565.49	-

Stateful allocators are not a luxury for coroutine-based I/O; they are a prerequisite for competitive performance.

The question is how the allocator reaches `operator new`.

4.1 Senders

The P2300 authors solved the allocator propagation problem for senders with care and precision. The receiver's environment carries the allocator via `get_allocator(get_env(rcvr))`, and the framework's sender algorithms propagate it automatically through every level of nesting. P2300R4 describes the design:

> *"contextual information is associated with the receiver and is queried by the sender and/or operation state after the sender and the receiver are connected."*

The mechanism is elegant. Each sender algorithm - `let_value`, `then`, `when_all` - creates an internal receiver whose `get_env()` forwards queries to the parent receiver's environment via `forwarding-query`. Since `get_allocator` satisfies `forwarding-query`, the allocator propagates automatically through every level of the operation tree without any sender in the chain mentioning it:

```
auto work =
    just(std::move(socket))                         // pure/return
  | let_value([](tcp_socket& s) {                   // monadic bind
        return async_read(s, buf)                   // Kleisli arrow
             | let_value([&](auto data) {           // nested bind
                   return parse(data)               // another Kleisli arrow
                        | let_value([&](auto doc) { // third level bind
                              return async_write(   // another Kleisli arrow
                                  s, build_response(doc));
                          });
               });
    });

recycling_allocator<> alloc;                        // set once here
auto op = connect(
    write_env(std::move(work),
        prop(get_allocator, alloc)),
    rcvr);
start(op);
// The recycling_allocator set at the launch site is
// available via get_allocator(get_env(rcvr)) at every
// level of nesting. No function signature mentions it.
// No intermediate operation forwards it.
```

The allocator is set once at the launch site and reaches every operation in the tree - four levels deep in this example - without any intermediate sender mentioning it in a parameter list, a template argument, or a forwarding expression. This is genuinely good design. The P2300 authors invested substantial effort in making environment propagation automatic, invisible, and correct.

However...

None of this allocates. The standard sender algorithms compose as value types - the operation state produced by `connect` is a single concrete type, a deeply nested template instantiation with no heap allocation. The allocator propagates elegantly through a pipeline that does not allocate.

4.2 Coroutines

Coroutines are not afforded the same level of care. Consider the standard usage pattern for spawning a connection handler with `async_scope` (P3149R11):

```
namespace ex = std::execution;

ex::counting_scope scope;

// Spawn a handler for each accepted connection
ex::spawn(
    ex::on(sch, handle_connection(std::move(conn))),
    scope.get_token());
```

Two independent problems prevent the allocator from reaching the coroutine frame:

First, there is no API. Neither `spawn` nor `on` accept an allocator parameter. There is nowhere to put it. The sender algorithms that launch coroutines have no mechanism to forward an allocator into the coroutine's `operator new`.

Second, even if there were, the timing is wrong. The expression `handle_connection(std::move(conn))` is a function call. The compiler evaluates it - including `promise_type::operator new` - before `spawn` or `on` execute. The frame is already allocated by the time any sender algorithm runs. P3826R3 (Eric Niebler, 2026-01-05) confirms the consequence:

> *"The receiver is not known during early customization. Therefore, early customization is irreparably broken."*

The only way to get an allocator into the coroutine frame is through the coroutine's own parameter list:

```
// What users expect to write:
std::execution::task<>
handle_connection( tcp_socket conn );

// What they must actually write:
template<class Allocator>
std::execution::task<>
handle_connection( std::allocator_arg_t, Allocator alloc, tcp_socket conn );
                                                   // user's needs come last :(
```

And the call site becomes:

```
ex::spawn( ex::on( sch,
    handle_connection( std::allocator_arg, alloc, std::move(conn))),
    scope.get_token());                         // user's needs come last again
```

The environment propagation that the P2300 authors designed so carefully is structurally unreachable for coroutine frame allocation. The allocator must be manually threaded through every coroutine signature instead.

Senders get the allocator they do not need. Coroutines need the allocator they do not get.

4.3 Coroutines Work for What Senders Get Free

P3552R3 provides `std::allocator_arg_t` as a creation-time mechanism: the caller passes the allocator explicitly at the call site, and `promise_type::operator new` can use it. This solves the initial allocation.

Propagation remains unsolved. When a coroutine calls a child coroutine, the child's `operator new` fires during the function-call expression - before the parent's promise has any opportunity to intervene. The only workaround is manual forwarding: every coroutine in the chain must accept the allocator, query its environment, and forward explicitly. Three properties distinguish this from a minor inconvenience:

1. **Composability loss.** Generic sender algorithms like `let_value` and `when_all` launch child operations without knowledge of the caller's allocator. Manual forwarding cannot cross algorithm boundaries.

2. **Silent fallback.** Omitting `allocator_arg` from one call does not produce a compile error - the child silently falls back to the default heap allocator with no diagnostic.

3. **Protocol asymmetry.** Schedulers and stop tokens propagate automatically through the receiver environment. Allocators are the only execution resource that the protocol forces coroutine users to propagate by hand. See Appendix A for allocation cost benchmarks demonstrating why this matters for performance.

Senders receive the allocator through the environment - automatically, at every level of nesting, with no signature pollution. Coroutines receive it through `allocator_arg` - manually, at every call site, with silent fallback on any mistake. The same framework, the same resource, two completely different levels of support.

Should coroutines inherit the cost of a protocol they do not use?

4.4 Let the Domain Choose?

No workaround - global PMR, thread-local registries, or otherwise - can bypass the promise. Every path to `operator new` runs through `promise_type::operator new`. If the promise does not cooperate, the allocator does not reach the frame.

Our research in P4003R0 suggests that promise-level cooperation is the only viable path - but the right cooperation depends on the domain. A networking task needs per-connection recycling allocators. A GPU dispatch task needs device memory. A compute task might need arena allocation with bulk reclamation. These are different problems with different solutions.

The escape hatch is to stop searching for a universal allocator model in one promise type. Let each domain's task type cooperate with its own allocator strategy. Solutions exist when the promise is free to serve its domain rather than forced to serve all of them.

5. Elegant Senders, Viral Coroutines

Every production coroutine library has converged on a single template parameter:

Library	Declaration	Params
cppcoro	`template<typename T> class task`	1
libcoro	`template<typename return_type> class task`	1
asyncpp	`template<class T> class task`	1
aiopp	`template<typename Result> class Task`	1
Boost.Cobalt	`template<class T> class task`	1
Capy	`template<class T> class task`	1

P3552R3 innovates again:

Library	Declaration	Params
P3552R3 (std)	`template<class T, class Environment> class task`	2

The `Environment` parameter exists because sender/receiver needs it to carry the receiver's environment type. A coroutine-native design does not need it. Every library that provides coroutine-based APIs must either pick a concrete environment (limiting reuse) or template everything on the environment type (preventing separate compilation and ABI stability). Hyrum's Law predicts what happens next: any observable behavior of an API will be depended upon. An exposed template parameter invites creative use, and different communities settling on different environments risk fragmenting the ecosystem along type boundaries that need not exist.

5.1 The Viral Signature

Here is what allocator propagation looks like in a coroutine call chain under P3552R3:

```
template<typename Allocator>
task<void> level_three(
    std::allocator_arg_t = {},
    Allocator alloc = {})
{
    co_return;
}

template<typename Allocator>
task<void> level_two(
    int x,
    std::allocator_arg_t = {},
    Allocator alloc = {})
{
    co_await level_three(std::allocator_arg, alloc);
}

template<typename Allocator>
task<int> level_one(
    int v,
    std::allocator_arg_t = {},
    Allocator alloc = {})
{
    co_await level_two(42, std::allocator_arg, alloc);
    co_return v;
}
```

Every function signature carries allocator forwarding machinery that has nothing to do with the function's purpose. Forgetting to forward at any single call site silently breaks the chain with no compile error. Adding a new function in the middle requires updating every caller. The error code must be manually routed into a channel at every I/O boundary.

None of this appears in a sender pipeline. The environment propagates automatically. The signatures are clean.

5.2 A Known Anti-Pattern

The pattern of threading infrastructure concerns through every function signature has a name in every major software engineering tradition.

Gregor Kiczales et al. identified the problem in "Aspect-Oriented Programming" (ECOOP 1997) - the foundational AOP paper. Certain design decisions cannot be cleanly captured in procedural or OO decomposition, forcing implementations to be ***"scattered throughout the code, resulting in 'tangled' code that is excessively difficult to develop and maintain."***

The recognized code smell is called tramp data - parameters passed through intermediate functions that do not use them. Tramp data ***"increases code rigidity, constraining how you can refactor methods in the call chain"*** and ***"distributes architectural knowledge to functions that do not need it."***

The React ecosystem calls the same problem prop drilling (Kent C. Dodds) - passing data through intermediate components that do not need it. When it becomes burdensome, React Context provides an implicit channel. That implicit channel is architecturally identical to what P2300 environments do for senders, and to what thread-local propagation does for coroutines.

5.3 What the Core Guidelines Say

C++ Core Guidelines F.7 (Stroustrup & Sutter, eds.): ***"For general use, take `T*` or `T&` arguments rather than smart pointers."*** A function that uses an object should not be coupled to the caller's ownership or lifetime policy.

Herb Sutter, GotW #91: ***"Don't pass a smart pointer as a function parameter unless you want to use or manipulate the smart pointer itself, such as to share or transfer ownership."***

Alisdair Meredith & Pablo Halpern, "Getting Allocators out of Our Way" (CppCon 2019): Two senior committee members presented on making allocator support ***"transparent in the majority of cases"*** through language-level changes. They identified ***"enlarged interfaces and contracts"*** as a real-world cost of allocator integration.

Butler Lampson, "Hints for Computer System Design" (1983): ***"An interface should capture the minimum essentials of an abstraction. Don't generalize; generalizations are generally wrong."***

The parallel is direct: if `shared_ptr<widget>` does not belong in `f(widget&)` because lifetime management is not `f`'s job, then `allocator_arg` does not belong in `write(buf)` because allocation policy is not `write`'s job. The principle is the same. The Core Guidelines already state it. The committee's own allocator experts are working to eliminate it. Yet P3552R3 introduces it into every coroutine signature in a chain.

The practical consequence is predictable. The parameter burden means that in practice, the recycling allocator will not get used consistently. Developers will forget to forward at one call site, or decline to pollute a clean interface, and silently fall back to `std::allocator`. Performance will suffer. Benchmarks will blame coroutines. The perception that coroutines are not fit for high-performance I/O will become self-fulfilling - not because coroutines are slow, but because the framework made the fast path too hard to use.

Is this how the committee intended coroutines to be used?

---

6. The Gaps Cannot Be Fixed Later

`operator new` runs before `connect()`. No library-level change known to the authors can alter this sequencing - it is determined by the C++ language specification for coroutines interacting with the sender/receiver protocol. Closing the gap appears to require changing the sequencing - which means changing the API.

6.1 `await_transform` Cannot Help

In `co_await child_coro(args...)`, the function call `child_coro(args...)` is evaluated first - the child's `operator new` fires and the frame is allocated before `co_await` processing begins. By the time `await_transform` sees the returned `task<T>`, the child's frame has already been allocated without the parent's allocator.

6.2 No Path to `operator new`

P3552R3 establishes a two-tier model: the allocator is a creation-time concern passed at the call site, while the scheduler and stop token are connection-time concerns from the receiver. Propagating the allocator through a chain of nested coroutine calls remains the harder and unsolved problem. With standard containers, the allocator type is part of the type signature (`vector<T, Alloc>`), and ***uses_allocator*** construction gives generic code a standard way to propagate. With coroutines, the return type is `task<T>` - it does not carry the allocator type, and there is no ***uses_allocator*** equivalent for coroutine frame allocation.

6.3 P3826R3 Targets a Different Problem

P3826R3 proposes important fixes for sender algorithm customization. Its five proposed solutions all target algorithm dispatch - i.e., which implementation of `then`, `let_value`, or `bulk` should run. Four of the five do not change when the allocator becomes available. The fifth - remove all `std::execution` - resolves the gap by deferral. See Appendix A.3 for an analysis of each solution.

6.4 ABI Lock-In Makes the Gap Permanent

Once standardized, the relationship between `operator new` and `connect()` becomes part of the ABI. The standard does not break ABI on standardized interfaces. A fix would likely need `connect()` to propagate allocator context before the coroutine frame is allocated - a structural change to the sender protocol. The committee would then face a choice: accept the gap permanently, or introduce a parallel framework that serves coroutine-based I/O alongside the original.

6.5 The Three-Channel Model Is the ABI

The three-channel completion model is the foundation of `std::execution`'s completion signature machinery. Every sender algorithm's behavior is defined in terms of which channel fires. Adding a fourth channel, changing `when_all` semantics, or redefining the relationship between error codes and channels would be a breaking change to the model. The channel routing problem identified by Kohlhoff in 2021 has no resolution within the current model, and the ABI freeze would make the model permanent.

The natural compromise - ship `task` with known limitations and fix via DR or C++29 addendum - assumes the fix is a minor adjustment. Sections 6.1 through 6.4 show it is not. The allocator sequencing gap requires changing the relationship between `operator new` and `connect()`. The channel routing gap requires changing the completion model. Both are structural, and both become ABI once shipped. A DR that changes ABI is not a DR - it is a new framework.

Should the committee ship an API with acknowledged gaps?

---

7. The Design Has Not Converged

Since `std::execution` was approved for C++26 at Tokyo in March 2024, the committee has processed 50 items modifying this single feature in 22 months. All data is gathered from the published WG21 paper mailings, the LWG issues list, and the C++26 national body ballot comments.

We are not criticizing the amount of churn, or its direction. `std::execution` is a big feature; big features generate proportionate review activity. What we are asking is whether the unidirectional nature of the churn - every item flowing from sender/receiver machinery into the coroutine integration, never the reverse - harms an important language feature.

7.1 Items by Meeting Period

Period	Months	Removals	Reworks	Wording	Missing	LWG	Total
Pre-Wroclaw (Mar-Oct 2024)	8	1	5	0	1	0	7
Pre-Hagenberg (Nov 2024-Feb 25)	4	1	0	2	2	3	8
Pre-Sofia (Mar-Jun 2025)	4	0	2	0	7	1	10
Pre-Kona (Jul-Nov 2025)	5	0	3	3	1	7	14
Pre-London (Dec 2025-Feb 2026)	3	0	5	1	0	0	6
Total	24	2	15	6	11	11	45

Five NB comments on `task` allocator support and signal safety bring the total to 50 items.

7.2 Rate of Change

Normalizing by period duration reveals the acceleration:

Period	Items	Months	Items/Month
Pre-Wroclaw (Mar-Oct 2024)	7	8	0.88
Pre-Hagenberg (Nov 2024-Feb 25)	8	4	2.00
Pre-Sofia (Mar-Jun 2025)	10	4	2.50
Pre-Kona (Jul-Nov 2025)	14	5	2.80
Pre-London (Dec 2025-Feb 2026)	6	3	2.00

The rate rose steadily from 0.88 items/month to 2.80 items/month over the first four complete periods. The Pre-London period, only three months old, already matches Pre-Hagenberg - and is far from complete. The rate itself is not the concern - it is what the expanding categories of change (Section 7.3) and their unidirectional flow

(Section 7.4) reveal about the structural relationship between senders and coroutines.

7.3 The Subjects Are Not Converging

Early periods focused on API reworks (replacing `tag_invoke`, renaming algorithms). Later periods introduced new categories: safety defects (Priority 1 LWG issues in mid-2025), allocator concerns (NB comments at Kona in late 2025), and task-type design changes (D3980R0 in January 2026 changing the allocator model relative to the text adopted at Sofia in June 2025). The 2026-01 mailing added still more categories: operation state lifetime semantics (P3373R2), scheduler affinity redesign (P3941R1), and coroutine promise type correctness (P3950R0). Switzerland's CD ballot comment identifying signal-safety as "a serious defect" opens yet another front. The design surface under active modification is expanding, not contracting.

7.4 Coroutines Always Lose Out

Every one of the 50 items falls into one of two categories:

Category	Items	Examples
Sender/receiver machinery	~35	Algorithm customization (4 papers), removals, operation states, environments
Sender-to-coroutine bridge	~15	`task`, allocator model, `connect-awaitable`, `as.awaitable`, NB comments
Coroutine-intrinsic issues	0	(none)

Every item is either a sender/receiver design issue or a consequence of making sender/receiver work with coroutines. Not a single item represents a coroutine-intrinsic design problem propagating into the sender pipeline. The complexity flows in one direction: from sender/receiver into the coroutine integration, never the reverse.

The three structural gaps documented in this paper - the error channel mismatch, the `co_yield` workaround, and the allocator sequencing gap - are consistent with this pattern. All three arise because the sender/receiver model was designed for sender pipelines, and coroutines bear the cost. The churn data shows the committee investing heavily in the sender pipeline while the coroutine integration is repeatedly reworked to chase a moving target.

Discovering issues is a sign of thorough review, not careless design. But the unidirectional flow is not just a description of the past - it is predictive. If every one of 50 items originates in sender/receiver machinery and propagates cost into the coroutine integration, the structural relationship is clear: senders are the design center and coroutines are the adaptation layer. An adaptation layer absorbs the cost of every future change to the thing it adapts. As `std::execution` continues to evolve - and the expanding categories in Section 7.3 show it will - coroutines will continue to bear the integration cost. The three gaps documented in this paper are not the last. They are the first we found. The pattern predicts more.

What are the implications when a language feature only ever absorbs cost from a library feature, and never the reverse?

---

8. The Committee's Own Record

The committee's own proceedings confirm the gaps are known and unresolved.

LEWG polled the allocator question directly (P3796R1, September 2025):

> *"We would like to use the allocator provided by the receivers env instead of the one from the coroutine frame"*
>
> | *SF* | *F* | *N* | *A* | *SA* |
> |---|---|---|---|---|
> | *0* | *0* | *5* | *0* | *0* |
>
> *Attendance: 14. Outcome: strictly neutral.*

The entire room abstained. Without a mechanism to propagate allocator context through nested coroutine calls, the committee had no direction to endorse. D3980R0 (Kuhl, 2026-01-25) subsequently reworked the allocator propagation model relative to P3552R3 - adopted only six months earlier at Sofia. LWG 4356 confirms the gap has been filed as a specification defect.

The task type itself was contested. The forwarding poll (LEWG, 2025-05-06):

> *"Forward P3552R1 to LWG for C++29"*
>
> *SF:5 / F:7 / N:0 / A:0 / SA:0 - unanimous.*
>
> *"Forward P3552R1 to LWG with a recommendation to apply for C++26 (if possible)."*
>
> *SF:5 / F:3 / N:4 / A:1 / SA:0 - weak consensus, with "if possible" qualifier.*

C++29 was unanimous. C++26 was conditional and weak. P3796R1 catalogues sixteen distinct open concerns about `task`. P3801R0 (Jonathan Wakely, "Concerns about the design of `std::execution::task`") was filed in July 2025.

9. The Design Space Is Larger Than One Model

We are not proposing P4003R0 ("IoAwaitables: A Coroutines-Only Framework") for standardization. The paper is not yet ready for that step. But the code is real. It is extracted from a production library under active development, compiles on three major toolchains, and ships with benchmarks and unit tests. The code examples below are not sketches - they run today. We present them to inform the committee that the three gaps documented in this paper are not inherent to asynchronous programming. They are specific to the sender/receiver integration. The design space is larger than one model.

P4003R0 is not a solution for everyone. It is a possible solution for networking. There are likely others, also not sender/receiver, for other domains. We do not believe one model fits all. The fact that P4003R0 does not serve GPU dispatch or heterogeneous computing is not a deficiency - it is evidence that different domains need different models. Here is what a coroutine-only networking design looks like when it is free to serve its own domain:

```cpp
// main.cpp - the launch site decides allocation policy
int main()
{
    io_context ioc;
    pmr::monotonic_buffer_resource pool;

    // allocator set once here - never appears in a coroutine signature
    run_async(ioc.get_executor(), &pool)(accept_connections(ioc));

    ioc.run();
}

// server.cpp - coroutines just do their job
task<> accept_connections(io_context& ioc)
{
    auto stop_token = co_await this_coro::stop_token;
    tcp::acceptor acc(ioc, {tcp::v4(), 8080});
    while (! stop_token.stop_requested())
    {
        auto [ec, sock] = co_await acc.accept();
        if (ec) co_return;
        run_async(ioc.get_executor())(
            handle_request(std::move(sock)));
    }
}

task<> handle_request(tcp::socket sock)
{
    auto [ec, n] = co_await sock.read(buf);
    if (ec) co_return;

    auto doc = co_await parse_json(buf);
    auto resp = co_await build_response(doc);

    pmr::unsynchronized_pool_resource bg_pool;
    auto [ec2] = co_await run(bg_executor, &bg_pool)(
        write_audit_log(resp));

    co_await sock.write(resp);
}
```

No `allocator_arg` in any signature. No forwarding. No `Environment` template parameter. No error channel routing decision. The task type is `template<class T> class task` - one parameter, matching established practice. None of the three gaps exist.

This is not an argument that P4003R0 is the answer. It is an argument that answers exist - if the committee is willing to look beyond a single model. GPU dispatch, heterogeneous computing, and coroutine-based I/O have different requirements. A model designed for compile-time work graph construction may not be the right foundation for the programming model most developers will use. The three gaps in this paper are evidence that one model may not serve both audiences equally. Perhaps the right answer is not a better integration between senders and coroutines, but an honest acknowledgment that they serve different audiences, with interoperation at the boundary rather than forced unification.

Does the evidence support one model for all of asynchronous C++?

---

10. Conclusion

Two independent lines of evidence support the conclusion that `std::execution`'s coroutine integration has not received the same level of design investment as its sender pipeline:

Three structural gaps. The three-channel model forces an impossible choice about where the error code goes - and no choice produces correct behavior under `when_all`. The standard's own task type requires `co_yield with_error(e)` to signal an error - a repurposed keyword that no other coroutine library uses. The allocator arrives after the coroutine frame is allocated. Each gap is independently unfixable without model or language changes. Each originates in the integration between sender/receiver and coroutines, not in coroutines themselves.

Empirical churn data. Fifty modifications in 22 months at an accelerating rate, with all complexity flowing from sender/receiver machinery into the coroutine integration. Not a single item represents a coroutine-intrinsic problem. The sender pipeline is still under active development; the coroutine integration is being repeatedly reworked to chase it.

We ask the committee to choose a path forward. In order of descending preference:

1. **Fix the three gaps before shipping C++26.** Address the error channel, the `co_yield` workaround, and the allocator sequencing gap. We welcome a solution from the P2300 authors and are eager to collaborate on one.

2. **Ship `std::execution` without `task`.** Iterate on the coroutine integration for C++29. This preserves the sender/receiver protocol, avoids ABI lock-in on the coroutine integration, and keeps the CPU/GPU use cases on track.

3. **Defer `std::execution` to C++29.** Give P4003R0 and other alternatives time to explore whether the gaps can be closed - or whether coroutine-based I/O is better served by a different execution model entirely.

4. **Consider that coroutines and senders may need to part ways.** Each serves the use cases it was designed for. Coroutines serve developers who write asynchronous code. Senders serve architects who build execution frameworks. The current integration forces coroutines to bear costs that originate in sender/receiver design choices. Perhaps the answer is not a better integration, but an honest separation.

Appendix A - Code Examples

A.1 Why HALO Cannot Help

HALO allows compilers to elide coroutine frame allocation when the frame's lifetime is provably bounded by its caller. When an I/O coroutine is launched onto an execution context, the frame must outlive the launching function:

```cpp
namespace ex = std::execution;

task<size_t> read_data(socket& s, buffer& buf)
{
    co_return co_await s.async_read(buf);
}

void start_read(ex::counting_scope& scope, auto sch)
{
    ex::spawn(                                  // Frame must outlive this function
        ex::on(sch, read_data(sock, buf)),
        scope.get_token());
}                                               // Caller returns immediately
```

The compiler cannot prove bounded lifetime, so HALO cannot apply and allocation is mandatory. In the nested case - where coroutine A `co_await`s coroutine B - B's frame lifetime is bounded by A's. In principle, a sufficiently advanced compiler could elide B's allocation. No current compiler does this for `task<T>` coroutines, and the optimization becomes infeasible once any frame in the chain is transferred to an execution context.

A.2 The Full Ceremony for Allocator-Aware Coroutines

The current sender/receiver model requires five layers of machinery to propagate a custom allocator through a coroutine call chain:

```cpp
namespace ex = std::execution;

// 1. Define a custom environment with the allocator
struct my_env
{
    using allocator_type = recycling_allocator<>;
    allocator_type alloc;

    friend auto tag_invoke(
        ex::get_allocator_t, my_env const& e) noexcept
    {
        return e.alloc;
    }
};

// 2. Alias the task type with the custom allocator
using my_task = ex::basic_task<
    ex::task_traits<my_env::allocator_type>>;

// 3. Every coroutine accepts and forwards the allocator
template<typename Allocator>
my_task level_two(
    int x,
    std::allocator_arg_t = {},
    Allocator alloc = {})
{
    co_return;
}

template<typename Allocator>
my_task level_one(
    int v,
    std::allocator_arg_t = {},
    Allocator alloc = {})
{
    co_await level_two(42, std::allocator_arg, alloc);
    co_return;
}

// At the launch site: inject the allocator via write_env
void launch(ex::io_context& ctx)
{
    my_env env{recycling_allocator<>{}};
    auto sndr =
        ex::write_env(level_one(0), env)
      | ex::continues_on(ctx.get_scheduler());
    ex::spawn(std::move(sndr), ctx.get_token());
}
```

Forgetting any one of the five steps silently falls back to the default allocator. The compiler provides no diagnostic.

A.3 P3826R3 and Algorithm Dispatch

P3826R3 proposes important fixes for sender algorithm customization. P3826 offers five solutions. All target algorithm dispatch:

Solution 4.1: Remove all `std::execution`. Resolves the allocator sequencing gap by deferral.

Solution 4.2: Remove customizable sender algorithms. Does not change when the allocator becomes available.

Solution 4.3: Remove sender algorithm customization. Does not change when the allocator becomes available.

Solution 4.4: Ship as-is, fix via DR. Defers the fix. Does not change when the allocator becomes available.

Solution 4.5: Fix algorithm customization now. Restructures `transform_sender` to take the receiver's environment, changing information flow at `connect()` time. This enables correct algorithm dispatch but does not change when the allocator becomes available - the receiver's environment is still only queryable after `connect()`.

A.4 Why the Three-Channel Model Exists

The sender/receiver model constructs the entire work graph at compile time as a deeply-nested template type. `connect(sndr, rcvr)` collapses the pipeline into a single concrete type. For this to work, every control flow path must be distinguishable at the type level, not the value level.

The three completion channels provide exactly this. Completion signatures declare three distinct type-level paths:

```
using completion_signatures =
    stdexec::completion_signatures<
        set_value_t(int),                          // value path
        set_error_t(error_code),                   // error path
        set_stopped_t()>;                          // stopped path
```

`when_all` dispatches on which channel fired without inspecting payloads. `upon_error` attaches to the error path at the type level. `let_value` attaches to the value path at the type level. The routing is in the types, not in the values.

If errors were delivered as values (for example, `expected<int, error_code>` through `set_value`), the compiler would see one path carrying one type. `when_all` could not cancel siblings on error without runtime inspection of the payload. `upon_error` could not exist as a type-level construct. Every algorithm would need runtime branching logic to inspect the expected and route accordingly, rather than having separate continuations selected at compile time.

The three channels exist because the sender pipeline is a compile-time language.

Compile-time languages route on types. Runtime languages route on values. A coroutine returns `auto [ec, n] = co_await read(buf)` and branches with `if (ec)` at runtime. The sender pipeline encodes `set_value` and `set_error` as separate types in the completion signature and routes at compile time. The three-channel model is not an arbitrary design choice; it is a structural requirement of the compile-time work graph.

A compile-time language cannot express partial success.

I/O operations return `(error_code, size_t)` together because partial success is normal. A `read` that transfers 47 bytes before EOF is simultaneously a success (47 bytes delivered) and an error (EOF). The three-channel model demands that the sender author choose one channel. Choosing `set_error` loses the byte count. Choosing `set_value` hides the error from every type-level routing algorithm. No choice is correct because the compile-time type system cannot represent "both at once."

The problem is fundamental.

Eliminating the three-channel model would remove the type-level routing that makes the compile-time work graph possible. The zero-allocation pipelines, the full type visibility, and the deterministic execution documented in the sender/receiver design all depend on type-level dispatch through completion channels. The three channels are not a design flaw. They are the price of compile-time analysis. I/O cannot pay that price because I/O's completion semantics are inherently runtime: whether a read succeeded, partially succeeded, or failed is determined by the operating system, not by the type system.

Appendix B - Complete Churn Catalogue

The following tables list every item identified in the survey, organized by category.

Removals

Paper	Title	Date	Status
P3187R1	Remove `ensure_started` and `start_detached` from P2300	2024-10-15	Adopted
P3682R0	Remove `std::execution::split`	2025-02-04	Adopted

Major Design Reworks

Paper	Title	Date	Status
P2855R1	Member customization points for Senders and Receivers	2024-03-18	Adopted
P2999R3	Sender Algorithm Customization	2024-04-16	Adopted
P3303R1	Fixing Lazy Sender Algorithm Customization	2024-10-15	Adopted
P3175R3	Reconsidering the `std::execution::on` algorithm	2024-10-15	Adopted
P3557R3	High-Quality Sender Diagnostics with Constexpr Exceptions	2025-06-10	Adopted
P3570R2	Optional variants in sender/receiver	2025-06-14	Adopted
P3718R0	Fixing Lazy Sender Algorithm Customization, Again	2025-07-24	In Progress
P3826R3	Fix Sender Algorithm Customization	2025-11-14	In Progress
P3927R0	`task_scheduler` Support for Parallel Bulk Execution	2026-01-15	In Progress
D3980R0	Task's Allocator Use	2026-01-25	In Progress

Paper	Title	Date	Status
P3373R2	Of Operation States and Their Lifetimes	2025-12-29	In Progress
P3941R1	Scheduler Affinity	2026-01-14	In Progress
P3950R0	`return_value` & `return_void` Are Not Mutually Exclusive	2025-12-21	In Progress

LWG Defects

Issue	Title	Priority/Status
LWG 4368	Potential dangling reference from `transform_sender`	Open - Priority 1
LWG 4206	`connect_result_t` should be constrained with `sender_to`	Open - Priority 1
LWG 4215	`run_loop::finish` should be `noexcept`	Open
LWG 4190	`completion-signatures-for` specification is recursive	Open
LWG 4356	`connect()` should use `get_allocator(get_env(rcvr))`	Open

NB Comments

NB Comment	Title	Status
US 255-384	Use allocator from receiver's environment	Wording in D3980R0
US 253-386	Allow use of arbitrary allocators for coroutine frame	Wording in D3980R0
US 254-385	Constrain `allocator_arg` argument position	Wording in D3980R0
US 261-391	Bad specification of parameter type	Wording in D3980R0

NB Comment	Title	Status
CH	Signal-safety defect	Needs resolution

References

1. P2300R10 - std::execution (Michal Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Eric Niebler)
2. P2300R4 - std::execution (Michal Dominiak, et al., 2022)
3. P2430R0 - Partial success scenarios with P2300 (Chris Kohlhoff, 2021)
4. P2762R2 - Sender/Receiver Interface For Networking (Dietmar Kuhl, 2023)
5. P3149R11 - async_scope (Ian Petersen, Jessica Wong, Kirk Shoop, et al., 2025)
6. P3175R3 - Reconsidering the std::execution::on algorithm (Eric Niebler, 2024)
7. P3187R1 - Remove ensure_started and start_detached from P2300 (Lewis Baker, Eric Niebler, 2024)
8. P3373R2 - Of Operation States and Their Lifetimes (Robert Leahy, 2025)
9. P3552R3 - Add a Coroutine Task Type (Dietmar Kuhl, Maikel Nadolski, 2025)
10. P3682R0 - Remove std::execution::split (Eric Niebler, 2025)
11. P3796R1 - Coroutine Task Issues (Dietmar Kuhl, 2025)
12. P3801R0 - Concerns about the design of std::execution::task (Jonathan Wakely, 2025)
13. P3826R3 - Fix Sender Algorithm Customization (Eric Niebler, 2026)
14. P3941R1 - Scheduler Affinity (Dietmar Kuhl, 2026)
15. P3950R0 - return_value & return_void Are Not Mutually Exclusive (Robert Leahy, 2025)
16. D3980R0 - Task's Allocator Use (Dietmar Kuhl, 2026)
17. P4003R0 - IoAwaitables: A Coroutines-Only Framework (Vinnie Falco, 2026)
18. C++ Core Guidelines - F.7, R.30 (Bjarne Stroustrup, Herb Sutter, eds.)
19. Herb Sutter, GotW #91: Smart Pointer Parameters (2013)
20. Alisdair Meredith & Pablo Halpern, "Getting Allocators out of Our Way" (CppCon 2019)
21. Gregor Kiczales et al., "Aspect-Oriented Programming" (ECOOP 1997)
22. Butler Lampson, "Hints for Computer System Design" (1983)
23. Herb Sutter, "Trip report: Summer ISO C++ standards meeting (St Louis, MO, USA)" (2024)
24. Herb Sutter, "Living in the future: Using C++26 at work" (2025)
25. N5028 - Result of voting on ISO/IEC CD 14882 (2025)