Senders and Coroutines

Document Number:	P4007R0
Date:	2026-02-17
Audience:	LEWG

Reply-to:	Vinnie Falco vinnie.falco@gmail.com Mungo Gill mungo.gill@me.com

Table of Contents

Abstract

`std::execution` serves its domain well. Different asynchronous domains have different costs, and a single model cannot minimize all of them simultaneously. This paper identifies three structural gaps at the boundary where the sender model meets coroutines: in error reporting, error returns, and allocator propagation. Each gap is the cost of a property the sender model requires for compile-time analysis (P2300R10[1], D4014R0[26]). They are not design defects, they are tradeoffs. Mandating that standard networking be built on the sender model would force coroutine I/O users to pay these costs. A coroutine-native I/O exploration (P4003R0[25]) made the gaps visible by showing that partial results, error returns, cancellation, and allocator

propagation emerge naturally when I/O is designed for coroutines. The findings hold regardless of that exploration's specific design. This paper recommends: ship `std::execution` for C++26, defer `task` to C++29, and explore coroutine-native I/O designs alongside sender-based designs.

Revision History

R0: March 2026 (pre-Croydon mailing)

- Initial version.

---

1. Disclosure

The authors developed P4003R0[25] ("IoAwaitables: A Coroutines-Only Framework"), a coroutine-native I/O exploration. P4003 is one point in a design space, not a proposal. It asks: "what does I/O look like when designed exclusively for coroutines?" That question has its own tradeoffs. A coroutine-only design cannot express compile-time work graphs, does not support heterogeneous dispatch, and assumes a cooperative runtime. Those are real costs.

The point is not that coroutine-native I/O is universally superior. The point is that different domains have different costs, and a single model cannot minimize all of them simultaneously. The structural gaps documented in Sections 3-5 exist at the boundary where `std::execution` meets coroutines. They hold regardless of whether P4003 is the right design or any design at all. Code examples in Section 2 are drawn from P4003 to illustrate what coroutine-native I/O looks like - not to propose it as the only way forward.

---

2. Coroutine-Native I/O

Eric Niebler wrote in "Structured Concurrency"[33] (2020): ***"I think that 90% of all async code in the future should be coroutines simply for maintainability."***

What happens when I/O is designed from the ground up for coroutines? P4003R0[25] ("IoAwaitables: A Coroutines-Only Framework") explored this question. The code compiles on three toolchains with benchmarks and unit tests. Four properties emerged naturally.

2.1 Partial Results Are Natural

An I/O completion returns an error code and a result together. A composed `read` built from `read_some` shows this directly:

```
io_task<std::string> read(tcp::socket& sock)
{
    std::string result;
    char buf[1024];
    for (;;)
    {
        auto [ec, n] = co_await sock.read_some(buf);
        result.append(buf, n);              // n bytes arrived regardless of ec
        if (ec)
            co_return {ec, std::move(result)};
    }
}
```

The return type `io_task<std::string>` means the awaitable yields `(error_code, std::string)`. A caller writes `auto [ec, body] = co_await read(sock)`. A `read` that transferred 47 bytes before EOF gives the caller 47 bytes and `eof`. Error and data travel together in a single tuple. The coroutine decides what to do with both.

2.2 Errors Return Naturally

`co_return` handles success and failure the same way every production coroutine library has independently adopted:

```
route_task https_redirect(route_params& rp)
{
    std::string url = "https://";
    url += rp.req.at(field::host) + rp.url.encoded_path();
    rp.res.set(field::location, url);
    auto [ec] = co_await rp.status(status::found).send("redirect");
    if (ec)
        co_return route_error(ec);
    co_return route_done;
}
```

`co_return` handles both paths the same way it does in cppcoro, folly::coro, Boost.Cobalt, Boost.Asio, libcoro, and asyncpp.

2.3 Handle Errors, and Cancel is Free

On Windows, `CancelIoEx` completes the pending operation with `ERROR_OPERATION_ABORTED`. On POSIX, cancellation produces `ECANCELED`. The error code arrives through the same field as every other error, accompanied by the same byte count.

Code that already checks `ec` handles cancellation for free. No additional code path, no separate signal, no `set_stopped` handler. The `if (ec)` that handles `connection_reset` also handles `operation_aborted`. Cancellation is not a separate concept in I/O - it is another error code.

2.4 Allocators Propagate Transparently

Coroutine frame allocation overhead is measurable at high request rates. A recycling allocator eliminates it (Section 5 benchmarks a 3.1x speedup on MSVC). The allocator is set once at the launch site:

```
run_async( &pool )( do_connection(sock) );
```

The two-call syntax exists because of `operator new` timing. `run_async(&pool)` returns a wrapper that stores the allocator in thread-local storage ***before*** `do_connection(sock)` is invoked. By the time `do_connection`'s `promise_type::operator new` fires, the allocator is already available. C++17 guaranteed evaluation order makes this safe.

A promise opts in by reading the thread-local in `operator new`:

```
static void* operator new(std::size_t size) {
    auto* mr = get_current_frame_allocator();
    if (!mr)
        mr = std::pmr::new_delete_resource();
    auto total = size + sizeof(std::pmr::memory_resource*);
    void* raw = mr->allocate(total, alignof(std::max_align_t));
    std::memcpy(static_cast<char*>(raw) + size, &mr, sizeof(mr));
    return raw;
}
```

On every resume, the coroutine restores the thread-local from its heap-stable environment:

```
void await_resume() const noexcept {
    if (p_->frame_allocator())
        set_current_frame_allocator(p_->frame_allocator());
}
```

The window is narrow and deterministic: the thread-local is written when the coroutine resumes and read only in `operator new` of a child coroutine called during that execution. The technique is the same principle as `std::pmr::get_default_resource()`, scoped per-chain instead of per-process.

The result is that coroutine signatures express only their purpose:

```
route_task serve_api(route_params& rp)
{
    auto result = co_await db.query("SELECT ...");  // coroutine frame uses allocator here
    auto json = serialize(result);
    auto [ec] = co_await rp.send(json);             // and here
    if (ec) co_return route_error(ec);
    co_return route_done;
}
```

The allocator is set once at the launch site and reaches every frame automatically. We would welcome the opportunity to work with the P2300R10[1] and P3552R3[13] authors to explore whether a similar approach could deliver automatic allocator propagation to `std::execution::task`.

2.5 Two Models of Computation

The Sender Sub-Language expresses the same logic differently:

```
auto sndr = just(std::move(socket))
          | let_value([](tcp_socket& s) {
               return async_read(s, buf)
                    | then([](auto data) {
                         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);
start(op);
```

Niebler asked the question himself[33]: ***"Why would anybody write that when we have coroutines? You would certainly need a good reason."***

This is continuation-passing style[39] expressed as composable value types. P4014R0[26] ("The Sender Sub-Language") provides the full treatment. Coroutines inhabit direct-style C++ - values return to callers, errors propagate through the call stack, resources scope to lexical lifetimes. This paper calls that ***regular C++***.

Niebler characterized[33] the trade-off: ***"That style of programming makes a different tradeoff, however: it is far harder to write and read than the equivalent coroutine."***

SG4 polled at Kona (November 2023) on P2762R2[4] ("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.*

The poll presented two alternatives: the Networking TS's executor model and the sender model. A coroutine-native approach was not among the choices. This paper introduces a third option.

This paper recommends: ship `std::execution` for C++26, defer `task` to C++29, and explore coroutine-native I/O designs alongside sender-based designs. The following three sections explain why.

---

3. Where Do Errors Go?

How should an asynchronous operation report its outcome? The sender model has an elegant answer.

3.1 Senders Use Channels

The committee adopted three Sender completion channels: `set_value`, `set_error`, and `set_stopped`. These channels are the sender model's type system. Type-level routing is a requirement of CPS reification (Appendix A.1). From P2300R10 Section 1.3.1[1]:

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

The three channels enable compile-time routing: `upon_error` attaches to the error path at the type level, `let_value` chains successes, and algorithms like `when_all`[50] cancel siblings when a child completes through the error or stopped channel - all without runtime inspection of the payload.

Each channel has a fixed signature shape:

- `set_value(receiver, Ts...)` carries the success payload.
- `set_error(receiver, E)` carries a single, strongly-typed error.
- `set_stopped(receiver)` is a stateless signal.

3.2 What I/O Produces

The operating system is not a sender.

On POSIX, `read()` returns a byte count; errors arrive through `errno`. On Windows, `GetOverlappedResult` delivers a byte count and an error code. Boost.Asio[40] unified these into `void(error_code, size_t)` twenty-five years ago. The signature was correct then. P2762R2[4] ("Sender/Receiver Interface for Networking") preserves it because it is correct now.

A tuple accurately reflects what composed I/O operations physically produce. A `read_until` accumulates bytes across several receives, transferring 47 bytes before the connection resets. The error code says what happened. The byte count says how far it got. Both values are always meaningful.

Cancellation is an error code.

`CancelIoEx` on Windows completes the pending receive with `ERROR_OPERATION_ABORTED`. `close()` on POSIX produces `ECANCELED`. The error code arrives through the same field as every other error, accompanied by the same byte count. A composed `read_until` that has accumulated bytes across prior successful receives breaks the same way.

I/O completions are ***complicated success***. EOF with partial data, cancellation with accumulated progress, a reset after bytes already transferred - these are not failures. They are outcomes the caller must inspect. The sender model offers three ***simple channels***: one value, one error, one stop signal. Complicated success fits the value channel - but not the error channel or the stopped channel. For I/O, two of three channels go unused:

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

do_read(sock, buf)
  | then([](std::pair<error_code, std::size_t> result) {
        auto [ec, n] = result;
        // caller classifies ec at runtime
    })
  | upon_error([](auto e) {
        // NEVER CALLED - I/O errors travel in the pair
    })
  | upon_stopped([] {
        // NEVER CALLED - cancellation is error_code in the pair
    });
```

3.3 Does P2300R10 Guide Us?

P2300R10 Section 1.3.3[1] presents a composed read in its motivating examples. The operation reads a length-prefixed byte array: first the size, then the data.

```
using namespace std::execution;

sender_of<std::size_t> auto async_read(
    sender_of<std::span<std::byte>> auto buffer,
    auto handle);

struct dynamic_buffer {
  std::unique_ptr<std::byte[]> data;
  std::size_t size;
};

sender_of<dynamic_buffer> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(std::as_writeable_bytes(std::span(&buf.size, 1)))
                | async_read(handle)
                | then(
                    [&buf] (std::size_t bytes_read) {
                      assert(bytes_read == sizeof(buf.size));
                      buf.data = std::make_unique<std::byte[]>(buf.size);
                      return std::span(buf.data.get(), buf.size);
                    })
                | async_read(handle)
                | then(
                    [&buf] (std::size_t bytes_read) {
                      assert(bytes_read == buf.size);
                      return std::move(buf);
                    });
       });
}
```

Note: P2300R10 writes `as_writeable_bytes`. The standard library function is `std::as_writable_bytes`.

`async_read` is `sender_of<std::size_t>`. It completes through `set_value` with the byte count, or through `set_error` with the error code.

Consider the second `async_read` failing partway through. The first read succeeded. `buf.size` is valid. `buf.data` is allocated and correctly sized. The second read begins filling `buf.data`. Some bytes transfer. The connection resets.

`async_read` completes through `set_error`. The byte count - how far the second read got - has no channel to travel through. `set_error` accepts one argument: the error code.

`then` handles only `set_value`. The error propagates past it. The error exits `let_value`. The `dynamic_buffer` that `let_value` owns - with its valid `size`, its allocated `data`, its partial contents - is destroyed.

The caller receives the error code. The byte count from the failing read, the allocated buffer with its partial contents, and the successfully read size from the first operation are destroyed when the operation state is destroyed.

This is P2300R10 Section 1.3.3[1]'s motivating example of sender composition.

3.4 No Choice Is Correct

P2300R10[1] does not define `async_read`, but Section 1.4[1] presents `recv_sender` and `async_recv` - the primitive receive operation built on Windows IOCP. Any implementation of `async_read` loops the primitive until the buffer is full (Appendix A.2 shows the complete sender). The completion callback is where accumulated progress meets the channel model:

```
void completed(std::error_code ec, std::size_t n) {
    bytes_read_ += n;
    if (bytes_read_ == len_)
        return execution::set_value(std::move(rcvr_), bytes_read_);
    if (!ec)
        return start_recv();
    execution::set_error(std::move(rcvr_), ec);   // bytes_read_ bytes are lost
}
```

Each successful `async_recv` deposits bytes and advances `bytes_read_`. When a later receive fails, `set_error` carries the error code. `bytes_read_` - the accumulated progress from every prior successful receive - has no channel.

The mismatch is not a missing convention. It is a structural incompatibility between the three-channel type system 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 I/O.** Every generic error-handling algorithm in the sender model becomes inaccessible to I/O senders. `upon_error`, `let_error`, `upon_stopped` are unreachable. Algorithms that dispatch on channel - such as `when_all` cancelling siblings on error - cannot distinguish I/O failure from I/O success. The error channel and stopped channel both exist, and I/O never uses either.
- **Converge on `set_error` for I/O.** Partial success is inexpressible. The byte count is lost. A `read` that transferred 47 bytes before EOF is indistinguishable from a `read` that transferred zero.
- **Route cancellation through `set_stopped()`.** The error code and the byte count are both lost. `set_stopped` takes no arguments. The caller cannot distinguish a timeout from a user cancellation from a graceful shutdown - they all collapse to the same parameterless signal.
- **Write an adaptor.** The adaptor must know which convention the inner sender follows. This reintroduces the forced choice one level up.

The stdexec[52] repository contains exactly this adaptor. Robert Leahy's `use_sender` [60] (February 2026) bridges Asio's `(error_code, Args&&...)` completions into the three-channel model. Leahy is the author of P3373R2[12] and P3950R0[23], a contributor to the stdexec reference implementation, and an experienced practitioner in sender/receiver design. The adaptor receives the error code and byte count together from Asio. On cancellation, it dispatches:

```
::STDEXEC::set_stopped(static_cast<Receiver&&>(r_));
```

The byte count is gone. On error, the byte count is likewise discarded. Two of three paths lose the partial result. If an experienced sender/receiver practitioner cannot bridge these models without information loss, the limitation is in the model.

A committee member who has deployed sender/receiver in a production codebase (and who gave the authors permission to quote anonymously) independently confirmed the forced choice: ***"you can get partial results into S&R world with*** `asioexec::completion_token` ***but then you lose the error channel mapping that*** `asioexec::use_sender` ***does."***

The three-channel model assumes values, errors, and cancellation are distinct categories that can be separated at the type level. In I/O, they are not. EOF is information, not failure. Cancellation is an error code, not a separate signal. Partial success is the normal case, not an edge case. The model and the domain are incompatible. Appendix A.1 explains why.

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

3.5 The Paper Not Polled

Chris Kohlhoff identified this tension in 2021 in P2430R0[3] ("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."*

Kohlhoff published P2430R0 in August 2021, during the most intensive review period for P2300. The authors were unable to find minutes or polls addressing it in the published committee record. If the paper was reviewed, we would welcome a reference to the proceedings. Appendix A.3 traces the full timeline.

---

4. Where Is the `co_return`?

The three-channel completion model requires coroutines to deliver errors through a path that C++ coroutines do not provide. The workaround requires a language change.

4.1 The Mechanism

In a sender's operation state, signaling an error is direct: `set_error(std::move(rcvr), ec)`.

In a coroutine, `co_return expr` calls `promise.return_value(expr)`, which P3552R3[13] routes to `set_value`. There is no way to make `co_return` call `set_error`.

A coroutine promise can define `return_void` or `return_value`, not both ([dcl.fct.def.coroutine][50]). `task<void>` needs `return_void`. The `return_void` / `return_value` mutual exclusion blocks any `return_value` overload that could route to `set_error`.

`co_yield` is the only coroutine keyword that accepts an expression and passes it to the promise. Dietmar Kühl's `yield_value` overload for `with_error<E>` ([task.promise]/7[50]) is the best solution the language permits - and that is the problem. The constraints the sender model imposes are so narrow that even one of the most experienced C++ library engineers is left with a mechanism that reverses the established meaning of `co_yield` and requires a core language change to regularize. In every other coroutine context, `co_yield` means "produce a value and continue." Here it means "fail and terminate."

4.2 The Consequence

In regular C++, `co_return` delivers errors:

```
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);
    co_return n;
}
```

P3552R3[13] introduces a different mechanism:

```
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);
    co_return n;
}
```

For six years, every C++ coroutine library has taught the same two conventions: `co_return` for final values, `co_yield` for intermediate values that the coroutine continues to produce. The second example breaks both: `co_yield` does not produce a value, and the coroutine does not continue.

4.3 Established Practice

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

- cppcoro (Lewis Baker), Boost.Cobalt (Klemens Morgenstern), libcoro (Josh Baldwin): errors are exceptions or `co_return` values.
- folly::coro::Task (Facebook/Meta): errors are exceptions, propagated via `co_await`.
- Boost.Asio awaitable (Chris Kohlhoff): errors delivered as return values via `as_tuple(use_awaitable)`.

There is no third path. `std::execution::task` introduces one.

4.4 The Language Change

The `return_void` / `return_value` mutual exclusion has been part of the coroutine specification since N4499 (2015). In the decade since, cppcoro, folly::coro, Boost.Cobalt, Boost.Asio, libcoro, and asyncpp have shipped coroutine task types. None required the restriction to be lifted. All deliver errors through `co_return` or exceptions.

P1713R0[59] (Baker, 2019) proposed removing this restriction. It had no consensus in Cologne.

P3950R0[23] (Leahy, 2025) proposes the same change. The motivation is `std::execution`. The paper states: ***"Disallowing it either disadvantages coroutines vis-a-vis std::execution or necessitates library workarounds."*** The paper targets EWG - a core language change - to serve a library adopted by LEWG.

Müller's P3801R0[19] ("Concerns about the design of `std::execution::task`," 2025) confirms the cause: ***"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."***

A language rule that has served every coroutine library for a decade, and whose removal the committee previously declined, must now be reconsidered because one library requires it. The rule is not the problem.

---

5. Where Is the Allocator?

In the sender model, `connect` / `start` binds the continuation after the coroutine frame is already allocated. The allocator arrives too late.

Niebler identified the cost[33] in 2020:

> *"With coroutines, you have an allocation when a coroutine is first called, and an indirect function call each time it is resumed. The compiler can sometimes eliminate that overhead, but sometimes not."*

Each `task<T>` call invokes `promise_type::operator new`. At high request rates, frame allocations reach millions per second. A recycling allocator makes a measurable difference. P4003R0[25] ("IoAwaitables: A Coroutines-Only Framework") benchmarks a 4-deep coroutine call chain (2 million iterations):

Platform	Allocator	Time (ms)	Speedup
MSVC	Recycling	1265.2	3.1x
MSVC	mimalloc	1622.2	2.4x
MSVC	`std::allocator`	3926.9	-
Apple clang	Recycling	2297.08	1.6x
Apple clang	`std::allocator`	3565.49	-

5.1 Senders

The P2300R10[1] 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 sender algorithms propagate it automatically through every level of nesting:

```
auto work =
    just(std::move(socket))
  | let_value([](tcp_socket& s) {
        return async_read(s, buf)
             | let_value([&](auto data) {
                   return parse(data)
                        | let_value([&](auto doc) {
                              return async_write(
                                  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 allocator is set once at the launch site and reaches every operation automatically.

However...

Nothing here allocates. The operation state is a single concrete type with no heap allocation. The allocator propagates through a pipeline that does not need it.

5.2 Coroutines

Consider the standard usage pattern for spawning a connection handler:

```
namespace ex = std::execution;

ex::counting_scope scope;

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. 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.

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 );
```

P2300's elegant environment propagation is structurally unreachable for coroutine frames.

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

5.3 Coroutines Work for What Senders Get Free

Kühl's P3552R3[13] provides `std::allocator_arg_t` for the initial allocation. Propagation remains unsolved. The child's `operator new` fires before the parent can intervene. The only workaround is manual forwarding. 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 sender model forces coroutine users to propagate by hand.

C++ Core Guidelines F.7[37]: a function should not be coupled to the caller's ownership policy. Lampson (1983): ***"An interface should capture the minimum essentials of an abstraction."***

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.

5.4 Does Performance Matter?

The recycling allocator eliminates coroutine frame allocation overhead. It requires `allocator_arg` at every call site. One missed site falls back to `std::allocator` with no diagnostic. In production code, sites will be missed. Users profile, see heap allocation cost, and conclude coroutines are slow. Coroutines are not slow. The fast path is too hard to use.

5.5 Will HALO Save Us?

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:

```
namespace ex = std::execution;

ex::task<void> read_data(socket& s, buffer& buf)
{
    co_await s.async_read(buf);
}

void start_read(ex::counting_scope& scope, auto sch)
{
    ex::spawn(
        ex::on(sch, read_data(sock, buf)),
        scope.get_token());
}
```

The compiler cannot prove bounded lifetime, so HALO cannot apply and allocation is mandatory.

5.6 A Viral Signature?

Here is what allocator propagation looks like in a coroutine call chain under P3552R3[13]:

```
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;
}
```

5.7 Domain Freedom?

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

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. A networking task type can use thread-local propagation. A GPU task type can use device memory APIs. Solutions exist when the promise is free to serve its domain rather than forced to serve a model it does not participate in.

---

6. The Gaps Are The Tradeoff

Niebler wrote in 2024: ***"If your library exposes asynchrony, then returning a sender is a great choice: your users can await the sender in a coroutine if they like."***[34] The phrase "if they like" implies this is straightforward. The previous three sections suggest otherwise.

Each gap has the same shape. The sender model requires a property for compile-time analysis. That property forces a cost on coroutines at the boundary. The committee standardized the sender model for specific properties: compile-time routing, zero-allocation reification, type-level dispatch. Each gap documented in Sections 3-5 is the cost of one of those properties. Closing any gap requires removing the property it pays for.

6.1 The Channel Tradeoff

Section 3 showed the cost of compile-time channel routing. Choose one:

Compile-time channel routing	I/O tuple completion
`upon_error`, `let_error`, `upon_stopped` attach at the type level; algorithms dispatch on channel without runtime inspection	`(error_code, size_t)` returned together; error, cancellation, and byte count are all meaningful and inseparable

6.2 The `co_yield` Tradeoff

Section 4 showed the cost of a separate error channel for coroutines. The alternative, lifting the `return_void` / `return_value` mutual exclusion, is not part of C++26. Choose one:

Separate error channel from coroutines	Established `co_return` semantics
`co_yield with_error(ec);`	`co_return std::unexpected(ec);`

6.3 The Allocator Tradeoff

Section 5 showed the cost of deferred execution. Choose one:

Deferred execution via `connect` / `start`	Allocator reaches coroutine frames
Receiver environment available after `connect()` ; zero-allocation sender pipelines	`promise_type::operator new` fires at the call site with the right allocator

`await_transform` cannot help: the child's `operator new` fires before `co_await` processing begins. P3552R3[13] offers `allocator_arg` for the initial allocation, but propagation remains unsolved. P3826R3[20] offers five solutions for algorithm dispatch; none changes when the allocator becomes available.

6.4 ABI Makes the Choice Permanent

The first three tradeoffs are structural. This subsection is about timing.

Once shipped, the three-channel model and the `connect` / `start` protocol become ABI. Every sender algorithm's behavior is defined in terms of which channel fires. The relationship between the promise's `operator new` and `connect()` becomes fixed. Closing any of the three gaps after standardization requires changing these relationships - a breaking change to the sender protocol.

Choose one:

Ship `task` in C++26	Iterate the coroutine integration
The three tradeoffs above become ABI	The three tradeoffs remain open

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 6.3 show it is not. Each gap is the cost of a specific design property. Closing the gap means removing the property.

Shipping `task` is the risky choice, not the safe one. Three structural gaps locked in by ABI. A language change proposed to fix `co_yield with_error` for a channel I/O never uses. Allocator propagation still unsolved six months after adoption. Kühl himself filed sixteen open concerns. The sender model's environment leaks into the coroutine's public type: P3552R3[13] defines `task` with two template parameters where every production coroutine library has independently converged on one:

Library	Declaration	Params
asyncpp	`template<class T> class task`	1
Boost.Cobalt	`template<class T> class task`	1
cppcoro	`template<typename T> class task`	1
aiopp	`template<typename Result> class Task`	1
libcoro	`template<typename return_type> class task`	1
P3552R3 (std)	`template<class T, class Environment> class task`	2

Sources: cppcoro[42], libcoro[47], asyncpp[46], aiopp[48], P3552R3 ref. impl.[49]

Deferring `task` to C++29 costs nothing: no production user depends on it, every networking library already ships its own task type, and C++29 forwarding was unanimous.

The narrowest remedy is to ship `std::execution` without `task`. The sender pipeline is valuable and ready. The three gaps exist only at the coroutine boundary. Remove `task` from C++26, and no production user is affected. The cost falls on no one.

---

7. Why Wait To Ship?

The cost of shipping is documented above. What is the cost of waiting? The cost is real. C++26 ships without a standard task, and the committee's work on coroutine integration does not reach users in this cycle. The question is whether that cost exceeds the cost of locking in the three gaps documented above.

7.1 "C++ needs a standard task. Six years is long enough."

C++ needs `std::task`. The question is whether `std::execution::task` is that type.

Convention	P3552R3	Every existing library
Error signaling	`co_yield with_error(ec)`	`co_return` or exceptions
Type signature	`task<T, Environment>`	`task<T>`

A task bound to one model's conventions is a task for that model. The standard task for C++ has not been written yet.

7.2 "The gaps are manageable. Ship now, iterate later."

"Fix later" assumes the fix is a minor adjustment.

Each gap is the cost of a specific design property (Section 6). Closing the gap means removing the property. ABI lock-in makes the choice permanent (subsection 6.4). The committee deferred P2300 from C++23 for the same pattern of ongoing design changes.

The committee has been here before.

7.3 "A standard task enables library interop that no third-party type can."

Coroutine interop requires the awaitable protocol, not type identity.

P3552R3[13] Section 3:

- "different coroutine task implementations can live side by side: not all functionality has to be implemented by the same coroutine task."
- "it should be possible to `co_await` awaitables which includes both library provided and user provided ones."

Open task types ship in cppcoro[42], Boost.Cobalt[41], libunifex[43], folly::coro[44], QCoro[45], and asyncpp[46]. Each interoperates through the awaitable protocol.

7.4 “Without `task`, coroutine users cannot access standard networking.”

No standard networking exists in C++26. Every networking library that supports coroutines already ships its own task type:

Library	Coroutine Type
Boost.Asio	`awaitable<T>`
Boost.Cobalt	`task`, `promise`
folly::coro	`Task<T>`
libcoro	`task<T>`
COROIO	`TTask`
asyncpp	`task`

Coroutine users are not locked out of networking today and will not be locked out tomorrow.

Each argument assumes `std::execution::task` is the standard task C++ needs. The evidence suggests it is the execution model’s task - and that the standard task for C++ is still ahead.

---

8. Maturity of the Coroutine Integration

The coroutine integration is not ready for C++26. Three data points:

LEWG polled the allocator question directly (P3796R1[18], 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. Kühl himself identified the need for further iteration, reworking the allocator propagation model in D3980R0[24] (2026-01-25), six months after P3552R3[13]’s adoption at Sofia. LWG 4356 confirms the gap has been filed as a specification defect.

The task type’s forwarding polls (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 forwarding was unanimous. C++26 was conditional and weak. With the thoroughness that has defined his stewardship of `task`, Kühl catalogued sixteen open concerns in P3796R1[18] ("Coroutine Task Issues").

---

9. Conclusion

This paper recommends: ship `std::execution` for C++26, defer `task` to C++29, and explore coroutine-native I/O designs alongside sender-based designs.

`std::execution` has earned its place. Herb Sutter reported that Citadel Securities 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."***[36] Senders work well in their domain: compile-time work graph construction, GPU dispatch, high-frequency trading pipelines. The finding is not that `std::execution` failed. It is that its scope is specific, not universal. Narrowing the scope is not admitting failure - it is recognizing success where it exists and clearing the path where it does not.

The three gaps documented in Sections 3-5 are the cost of treating the sender model as the universal model of asynchronous computation. Each gap is the cost of a property the sender model requires. They are not design defects. They are tradeoffs - and coroutine I/O should not be forced to pay them.

1. **Is the sender model a universal model of asynchronous computation, or a domain-specific one?** The evidence suggests it serves GPU dispatch, high-frequency trading, and heterogeneous computing, but not I/O.
2. **Are coroutines the primary tool for asynchronous C++?** If yes, the coroutine integration deserves the same design investment as the sender pipeline itself.
3. **Should `task<T>` ship in C++26 with these costs?** Ship `std::execution` for the domains it serves. Let the coroutine integration iterate independently.

The SG4 poll (Kona 2023, SF:5/F:5/N:1/A:0/SA:1) presented two alternatives. A coroutine-native approach was not among the choices. The straw polls in Section 10 ask the committee whether the choice should remain binary.

---

10. Suggested Straw Polls

1. "`std::execution::task` concerns are addressable by C++29."
2. "Asynchronous C++ need not be limited to `std::execution`."
3. "WG21 should explore coroutine-native I/O alongside senders."

Appendix A - The Three-Channel Model

A.1 Why the Three-Channel Model Exists

The sender model constructs the entire work graph at compile time as a deeply-nested template type. `connect(sndr, rcvr)` [50] 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[50] declare three distinct type-level paths:

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

Algorithms dispatch 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. `upon_stopped` attaches to the stopped path. 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. Algorithms could not dispatch on error without runtime inspection of the payload. Every algorithm would need runtime branching logic to inspect the expected and route accordingly.

The three channels exist because the sender model 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 model 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. The three-channel model demands that the sender author choose one channel. No choice is correct because the compile-time type system cannot represent "both at once."

Eliminating the three-channel model would remove the type-level routing that makes the compile-time work graph possible. 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.

A.2 `async_read` as a Sender

Section 2.4 shows the completion callback where accumulated read progress is lost. Here is the complete sender implementation using P2300R10 Section 1.4[1]'s `recv_sender` and `async_recv`:

```
template<class Rcvr>
struct read_op
{
    struct recv_rcvr {
        read_op* self_;
        void set_value(std::size_t n) && noexcept { self_->completed({}, n); }
        void set_error(std::error_code ec) && noexcept { self_->completed(ec, 0); }
        void set_stopped() && noexcept { execution::set_stopped(std::move(self_->rcvr_)); }
    };

    Rcvr rcvr_;
    SOCKET sock_;
    std::byte* data_;
    std::size_t len_;
    std::size_t bytes_read_ = 0;
    std::optional<connect_result_t<recv_sender, recv_rcvr>> child_;

    void start() & noexcept { start_recv(); }

    void start_recv() {
        child_.emplace(execution::connect(
            async_recv(sock_, data_ + bytes_read_, len_ - bytes_read_),
            recv_rcvr{this}));
        execution::start(*child_);
    }

    void completed(std::error_code ec, std::size_t n) {
        bytes_read_ += n;
        if (bytes_read_ == len_)
            return execution::set_value(std::move(rcvr_), bytes_read_);
        if (!ec)
            return start_recv();
        execution::set_error(std::move(rcvr_), ec);   // bytes_read_ bytes are lost
    }
};
```

Every name is from P2300R10: `recv_sender`, `async_recv`, `connect`, `start`, `set_value`, `set_error`, `set_stopped`, `connect_result_t`. The `recv_rcvr` is the standard pattern for wiring a child sender back to a parent operation state.

A.3 Timeline of the Error Channel Question

The partial success problem was not raised once and set aside. It was raised independently, across multiple groups, by participants with different domain backgrounds, over a span of five years. The question was never polled.

- 2020 (Nov): Niebler's "Structured Concurrency"[33] blog post establishes the three-channel model publicly. Acknowledges CPS is harder to write and read than coroutines.

- **2020 (Feb, Prague):** During review of P1678R2[56] ("Callbacks and Composition"), a participant raised that asynchronous operations need not fail completely or succeed completely, and that no pattern in the proposal supports partial success. The response was that nothing would be different. The concern was not addressed. LEWG polled to encourage more work (SF:7/F:14/N:9/A:3/SA:0). No poll on partial success.

- **2021 (Feb, SG4 telecon):** During review of P0958R3, a participant stated that sender/receivers have a loss because they do not have success/partial-success. Another suggested adapting from success/error; the response was that the error does not carry information. No poll on the error channel design.

- **2021 (Jul-Oct, LEWG telecon series):** The partial success question was debated across at least five LEWG telecons during the P2300 review. Multiple participants raised incompatible positions. One argued that partial success calling `set_value` does not work for generic retry algorithms. Others described `set_error` as effectively `set_exception` - an error channel that does not serve error conditions. The August 17, 2021 LEWG outcome document explicitly listed "Better explain how partial success works with senders/receivers" as open guidance to the P2300 authors - an acknowledgment that the question was unresolved. During the October 4, 2021 LEWG telecon, a participant (who gave the authors permission to anonymously quote what they said) indicated that partial success could be addressed through async streams once the initial sender/receiver facilities were in place, with the expectation that such streams could then be standardized. The async streams facility was never proposed, never standardized, and is not in the C++26 working draft.

- **2021 (Aug):** Chris Kohlhoff published P2430R0[3] ("Partial success scenarios with P2300") identifying that partial results must be communicated down the `set_value` channel due to limitations of `set_error` and `set_done`. Published during P2300's most intensive review period. No committee response found in the published record.

- **2022-2024:** P2300R4[2] through P2300R10[1] published. Three-channel model unchanged across all revisions. P2430R0 not addressed.

- **2023:** P2762R2[4] (Dietmar Kühl, "Sender/Receiver Interface for Networking") preserves the proven `(error_code, size_t)` convention from Asio - implicitly acknowledging I/O needs both values together.

- **2023 (Nov, Kona):** SG4 polls that networking must use the sender model (SF:5/F:5/N:1/A:0/SA:1). Every future I/O operation must now navigate the channel mismatch.

- **2024 (Nov, Wroclaw):** The same three-channel question resurfaced during design of P0260 ("C++ Concurrent Queues"). LEWG debated whether a closed queue should complete through `set_error`, `set_value`, or `set_stopped`. One participant asked how popping from a closed queue could be an error when it represents the ultimate success of processing all items. Another noted that POSIX does not model EOF this way - reading from a socket at EOF succeeds with zero bytes. The debate consumed portions of two face-to-face meetings.

- **2025 (Feb, Hagenberg):** The concurrent queue channel question remained open. A poll to reopen the channel choice was withdrawn. A new problem was discovered: `try_push` cannot safely schedule an `async_pop` continuation because `std::execution` provides no way to express that a schedule operation is non-blocking - a concrete manifestation of the architectural mismatch in a producer-consumer API.

- **2025-2026:** P2300 adopted into the C++26 working draft. P3570R2[15] (Fracassi, "Optional variants in sender/receiver") documents the interface mismatch between `optional<T>` and the channel model. The error channel / partial success question remains open. No paper in the published record proposes a resolution that preserves compile-time channel routing.

The question has been open for five years. This duration is not due to neglect. Every proposed fix involves a tradeoff with real downsides (Section 3.4 enumerates five, all deficient). Five years of active iteration on P2300 - R0 through R10, plus dozens of follow-on papers - have not produced a resolution. The three-channel model is a property of the sender model, not a bug. The unresolved timeline is what structural friction looks like from the inside.

Appendix B - The Allocator Gap

B.1 The Full Ceremony for Allocator-Aware Coroutines

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

```
namespace ex = std::execution;

// 1. Define a custom environment that answers get_allocator
struct my_env
{
    using allocator_type = recycling_allocator<>;
    allocator_type alloc;

    allocator_type query(ex::get_allocator_t) const noexcept
    {
        return alloc;
    }
};

// 2. Alias the task type with the custom environment
using my_task = ex::task<void, my_env>;

// 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;
}

// 4. Construct the environment at the launch site
// 5. Inject it via write_env
void launch(ex::counting_scope& scope, auto sch)
{
    recycling_allocator<> alloc;                          // step 4
    my_env env{alloc};
    auto sndr =
        ex::write_env(
            level_one(0, std::allocator_arg, alloc), env)   // step 5
      | ex::continues_on(sch);
    ex::spawn(std::move(sndr), scope.get_token());
}
```

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

Appendix C - Direction of Change

The claim is not that the volume of changes is abnormal; it is that the direction is uniform. Every paper, LWG issue, and NB comment modifying `std::execution` since Tokyo (March 2024) falls into one of two categories.

Sender Sub-Language items address the CPS model's own machinery: algorithm customization, operation state lifetimes, completion signature constraints, removals of primitives that violated structured concurrency.

Sender Integration items address the boundary where the CPS model meets coroutines: the `task` type, allocator propagation into coroutine frames, `co_yield with_error` semantics.

Coroutine-Intrinsic items are specific to coroutines. There are none.

Origin	Items
Sender Sub-Language	P2855R1, P2999R3, P3175R3, P3187R1, P3303R1, P3373R2, P3557R3, P3570R2, P3682R0, P3718R0, P3826R3, P3941R1, LWG 4190, LWG 4206, LWG 4215, LWG 4368
Sender Integration	P3927R0, P3950R0, D3980R0, LWG 4356, US 255-384, US 253-386, US 254-385, US 261-391
Coroutine-Intrinsic	-

All data is gathered from the published WG21 paper mailings[54], the LWG issues list[55], and the C++26 national body ballot comments[53].

Acknowledgements

This document is written in Markdown and depends on the extensions in `pandoc` and `mermaid`, and we would like to thank the authors of those extensions and associated libraries.

The authors would also like to thank Peter Dimov, Klemens Morgenstern, Mohammad Nejati, and Michael Vandeberg for their valuable feedback in the development of this paper.

References

WG21 Papers

1. P2300R10 - "std::execution" (Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Eric Niebler, 2024). https://wg21.link/p2300r10
2. P2300R4 - "std::execution" (Michał Dominiak, et al., 2022). https://wg21.link/p2300r4
3. P2430R0 - "Partial success scenarios with P2300" (Chris Kohlhoff, 2021). https://wg21.link/p2430r0
4. P2762R2 - "Sender/Receiver Interface For Networking" (Dietmar Kühl, 2023). https://wg21.link/p2762r2
5. P2855R1 - "Member customization points for Senders and Receivers" (Ville Voutilainen, 2024). https://wg21.link/p2855r1
6. P2999R3 - "Sender Algorithm Customization" (Eric Niebler, 2024). https://wg21.link/p2999r3
7. P3149R11 - "async_scope" (Ian Petersen, Jessica Wong, Kirk Shoop, et al., 2025). https://wg21.link/p3149r11
8. P3164R4 - "Improving Diagnostics for Sender Expressions" (Eric Niebler, 2024). https://wg21.link/p3164r4
9. P3175R3 - "Reconsidering the std::execution::on algorithm" (Eric Niebler, 2024). https://wg21.link/p3175r3
10. P3187R1 - "Remove ensure_started and start_detached from P2300" (Kirk Shoop, Lewis Baker, 2024). https://wg21.link/p3187r1
11. P3303R1 - "Fixing Lazy Sender Algorithm Customization" (Eric Niebler, 2024). https://wg21.link/p3303r1
12. P3373R2 - "Of Operation States and Their Lifetimes" (Robert Leahy, 2025). https://wg21.link/p3373r2
13. P3552R3 - "Add a Coroutine Task Type" (Dietmar Kühl, Maikel Nadolski, 2025). https://wg21.link/p3552r3
14. P3557R3 - "High-Quality Sender Diagnostics with Constexpr Exceptions" (Eric Niebler, 2025). https://wg21.link/p3557r3
15. P3570R2 - "Optional variants in sender/receiver" (Fabio Fracassi, 2025). https://wg21.link/p3570r2
16. P3682R0 - "Remove std::execution::split" (Robert Leahy, 2025). https://wg21.link/p3682r0
17. P3718R0 - "Fixing Lazy Sender Algorithm Customization, Again" (Eric Niebler, 2025). https://wg21.link/p3718r0
18. P3796R1 - "Coroutine Task Issues" (Dietmar Kühl, 2025). https://wg21.link/p3796r1
19. P3801R0 - "Concerns about the design of std::execution::task" (Jonathan Müller, 2025). https://wg21.link/p3801r0
20. P3826R3 - "Fix Sender Algorithm Customization" (Eric Niebler, 2026). https://wg21.link/p3826r3
21. P3927R0 - "task_scheduler Support for Parallel Bulk Execution" (Lee Howes, 2026). https://wg21.link/p3927r0
22. P3941R1 - "Scheduler Affinity" (Dietmar Kühl, 2026). https://wg21.link/p3941r1
23. P3950R0 - "return_value & return_void Are Not Mutually Exclusive" (Robert Leahy, 2025). https://wg21.link/p3950r0
24. D3980R0 - "Task's Allocator Use" (Dietmar Kühl, 2026). https://isocpp.org/files/papers/D3980R0.html
25. P4003R0 - "IoAwaitables: A Coroutines-Only Framework" (Vinnie Falco, 2026). https://wg21.link/p4003r0
26. P4014R0 - "The Sender Sub-Language" (Vinnie Falco, 2026). https://wg21.link/p4014r0
27. N5028 - "Result of voting on ISO/IEC CD 14882" (2025). https://wg21.link/n5028

LWG Issues

28. LWG 4368 - "Potential dangling reference from `transform_sender`" (Priority 1). https://cplusplus.github.io/LWG/issue4368
29. LWG 4206 - "`connect_result_t` should be constrained with `sender_to`" (Priority 1). https://cplusplus.github.io/LWG/issue4206

30. LWG 4190 - "`completion-signatures-for` specification is recursive" (Priority 2). https://cplusplus.github.io/LWG/issue4190
31. LWG 4215 - "`run_loop::finish` should be `noexcept`". https://cplusplus.github.io/LWG/issue4215
32. LWG 4356 - "`connect()` should use `get_allocator(get_env(rcvr))`". https://cplusplus.github.io/LWG/issue4356

Blog Posts

33. Eric Niebler, "Structured Concurrency" (2020). https://ericniebler.com/2020/11/08/structured-concurrency/
34. Eric Niebler, "What are Senders Good For, Anyway?" (2024). https://ericniebler.com/2024/02/04/what-are-senders-good-for-anyway/
35. Herb Sutter, "Trip report: Summer ISO C++ standards meeting (St Louis, MO, USA)" (2024). https://herbsutter.com/2024/07/02/trip-report-summer-iso-c-standards-meeting-st-louis-mo-usa
36. Herb Sutter, "Living in the future: Using C++26 at work" (2025). https://herbsutter.com/2025/04/23/living-in-the-future-using-c26-at-work
37. C++ Core Guidelines - F.7, R.30 (Bjarne Stroustrup, Herb Sutter, eds.). https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html
38. Butler Lampson, "Hints for Computer System Design" (1983). https://bwlampson.site/33-Hints/Acrobat.pdf
39. Continuation-passing style - Wikipedia. https://en.wikipedia.org/wiki/Continuation-passing_style

Libraries

40. Boost.Asio - Asynchronous I/O library (Chris Kohlhoff). https://www.boost.org/doc/libs/release/doc/html/boost_asio.html
41. Boost.Cobalt - Coroutine task types for Boost (Klemens Morgenstern). https://www.boost.org/doc/libs/develop/libs/cobalt/doc/html/index.html
42. cppcoro - A library of C++ coroutine abstractions (Lewis Baker). https://github.com/lewissbaker/cppcoro
43. libunifex - Unified Executors library for C++ (Facebook/Meta, Eric Niebler). https://github.com/facebookexperimental/libunifex
44. folly::coro - Facebook's coroutine library. https://github.com/facebook/folly/tree/main/folly/experimental/coro
45. QCoro - Coroutine integration for Qt (Daniel Vrátil). https://qcoro.dev/
46. asyncpp - Async coroutine library (Péter Kardos). https://github.com/petiaccja/asyncpp
47. libcoro - C++20 coroutine library (Josh Baldwin). https://github.com/jbaldwin/libcoro
48. aiopp - Async I/O library (Joel Schumacher). https://github.com/pfirsich/aiopp
49. beman::task - P3552R3 reference implementation (Beman Project). https://github.com/bemanproject/task

Other

50. C++ Working Draft - (Richard Smith, ed.). https://eel.is/c++draft/
51. cppreference - C++ reference documentation. https://en.cppreference.com/
52. stdexec - NVIDIA reference implementation of std::execution. https://github.com/NVIDIA/stdexec

53. C++26 NB ballot comments - National body comments repository. https://github.com/cplusplus/nbballot
54. WG21 paper mailings - ISO C++ committee papers. https://open-std.org/jtc1/sc22/wg21/docs/papers/
55. LWG issues list - Library Working Group active issues. https://cplusplus.github.io/LWG/lwg-toc.html
56. P1678R2 - "Callbacks and Composition" (Kirk Shoop, 2020). https://wg21.link/p1678r2
57. cplusplus/sender-receiver #247 - "Add ability to know when computing completion-signatures whether the call to connect() will throw" (Lewis Baker, 2024). https://github.com/cplusplus/sender-receiver/issues/247
58. P3388R2 - "Provide nothrow connect guarantee for P2300" (Lewis Baker, 2025). https://wg21.link/p3388r2
59. P1713R0 - "Allowing both co_return; and co_return value; in the same coroutine" (Lewis Baker, 2019). https://wg21.link/p1713r0
60. Robert Leahy, `use_sender.hpp` - Asio-to-sender bridge in stdexec (2026). https://github.com/NVIDIA/stdexec/blob/9d5836a634a21ecb06d17352905d04f99f635be6/include/asioexec/use_sender.hpp