WG21 PROPOSAL

Senders and C++

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

Blog Posts

Programming Language Theory

Other

Abstract

With `std::execution` (P2300R10), the committee adopted the Sender Sub-Language, a continuation-passing style (CPS) programming model with its own control flow, variable binding, error handling, and type system (D4014R0). Programming language theory generates specific predictions about friction when CPS meets direct-style. This paper tests those predictions against the coroutine integration: three structural gaps in the integration and eight additional friction points in the Sub-Language's type system, semantics, and specification. The gaps are not design defects. They are the cost of the Sub-Language.

Revision History

R0: March 2026 (pre-Croydon mailing)

- Initial version.

---

1. Two Models of Computation

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

```
task<error_code> handle_request(tcp::socket&& sock)
{
    char buf[1024];
    auto [ec, n] = co_await sock.async_read(buf, sizeof(buf));
    if (ec) co_return ec;
    std::expected<document, error_code> doc = parse_json(buf, n);
    if (!doc) co_return doc.error();
    co_return co_await sock.async_write(build_response(*doc));
}
```

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

The Sender Sub-Language expresses the same logic differently:

```
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
```

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

Herb Sutter called `std::execution` ***"the biggest usability improvement yet to use the coroutine support we already have."***

This is continuation-passing style expressed as composable value types. P4014R0 ("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++***.

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.

Every future networking operation in the C++ standard must be a sender. Coroutines access those operations through `std::execution::task`. What happens when the integration layer has structural gaps?

Regular C++ language features - structured bindings, `if` with initializer, range-for - do not apply to sender pipelines. Niebler characterized the trade-off: ***"That style of programming makes a different tradeoff, however: it is far harder to write and read than the equivalent coroutine."***

The committee chose to standardize both. This paper examines what follows.

---

2. Three Predictions

The committee placed a CPS-based model alongside an imperative language. Programming language theory offers specific predictions about what happens at the boundary. Sections 3-5 test three of them.

Danvy, "Back to Direct Style" (1992): ***"Not all lambda-terms are CPS terms, and not all CPS terms encode a left-to-right call-by-value evaluation."***

The CPS transform is asymmetric. The Sub-Language's three completion channels partition the completion space at the type level. I/O operations return a single tuple whose meaning is determined at runtime. The two shapes are incompatible. ***Prediction: the three-channel completion model will force I/O sender authors into a choice where neither channel is correct.*** Section 3 tests this.

Plotkin, "Call-by-Name, Call-by-Value and the lambda-Calculus" (1975): ***"Operational equality is not preserved by either of the simulations."***

You can simulate one model in the other, but the simulation changes program behavior. `task<T>` is that simulation. A coroutine that returns `std::expected<size_t, error_code>` through `co_return` is operationally different from a sender that routes `error_code` through `set_error`. The first is invisible to `upon_error`. The second loses the byte count.

The simulation works. It does not preserve what the original meant. ***Prediction: the bridge between models will require coroutines to express CPS concepts they have no native syntax for.*** Section 4 tests this.

Strachey & Wadsworth, "Continuations: A Mathematical Semantics for Handling Full Jumps" (1974), as cited in later transformation work: ***"Transforming the representation of a direct-style semantics into continuation style usually does not yield the expected representation of a continuation-style semantics (i.e., one written by hand)."***

A hand-written sender allocates inside `connect()`, after the receiver's environment is available. A coroutine's `promise_type::operator new` fires at the function call, before any sender machinery runs. ***Prediction: resource propagation designed for the CPS model will not reach coroutine frames correctly.*** Section 5 tests this.

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."*** The phrase "if they like" implies this is straightforward. The predictions above suggest otherwise.

Wherever CPS and direct-style meet, we can expect friction. Coroutines are the first boundary, but not the last. LWG 4368 (dangling reference from `transform_sender`) is already one consequence. Sections 3-5 test whether the predictions hold.

---

3. Where Do Errors Go?

How should an asynchronous operation report its outcome? The Sub-Language 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 Sub-Language's type system. Type-level routing is a requirement of CPS reification (Appendix A.5). From P2300R10 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 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` 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 unified these into `void(error_code, size_t)` twenty-five years ago. The signature was correct then. P2762R2 ("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 47 bytes across prior successful receives breaks the same way.

3.3 Does P2300R10 Guide Us?

P2300R10 Section 1.3.3 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);
                    });
       });
}
```

`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's motivating example of sender composition.

3.4 No Choice Is Correct

P2300R10 does not define `async_read`, but Section 1.4 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.1 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 Sub-Language's 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 Sub-Language 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.
- **Add a fourth channel.** This would be a breaking change to the completion signature model.

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.5 explains why.

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.

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 ("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.

---

4. Where Is the `co_return`?

The Sub-Language requires errors to reach a separate channel. Regular C++ coroutines have no native path to one. `co_yield with_error` is one model of computation being asked to express a concept that belongs to the other.

4.1 Senders

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

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

The call is direct.

4.2 Coroutines

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 introduces a new 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);           // not co_return
        // never gets here?
    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 above 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.

`co_return expr` calls `promise.return_value(expr)`, which P3552R3 routes to `set_value`. There is no way to make `co_return` call `set_error`. A coroutine promise can only define `return_void` or `return_value`, not both, and `task<void>` needs `return_void`.

Dietmar Kühl needed a different mechanism. The `return_void` / `return_value` mutual exclusion and the Sub-Language's separate error channel leave exactly one path: `co_yield` is the only coroutine keyword that accepts an expression and passes it to the promise. Kühl's `yield_value` overload for `with_error<E>` ([task.promise]/7) is the best solution the language permits:

> *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)))`.

In every other coroutine context, `co_yield` means "produce a value and continue." Here it means "fail and terminate." The keyword's established meaning predicts the wrong behavior.

The committee is aware of this problem. Müller's P3801R0 ("Concerns about the design of `std::execution::task`," 2025) 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.

The `return_void` / `return_value` mutual exclusion has existed since C++20 and has not prevented any other coroutine library from delivering errors through `co_return`. The constraint becomes a Sub-Language problem: errors must reach a separate channel.

Does `co_yield with_error` serve coroutines, or the Sub-Language?

5. Where Is the Allocator?

In the committee's Sub-Language, `connect` / `start` binds the continuation after the coroutine frame is already allocated. The allocator arrives too late.

Niebler identified the cost 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 ("IoAwaitables: A Coroutines-Only Framework") 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	-

5.1 Senders

The P2300R10 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))                          // 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)
                        | let_value([&](auto doc) {  // third level bind
                              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

P3552R3 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 Sub-Language 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 Sub-Language forces coroutine users to propagate by hand.

C++ Core Guidelines F.7: 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 A 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;
}
```

5.6 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 Sub-Language it does not participate in.

---

6. Friction Beyond Coroutines

The following friction points were identified through a survey of active papers and LWG issues, but not analyzed to the depth of the three gaps above. The question is whether the pattern is better explained by scale or by structure.

6.1 Type-System Friction

Algorithm customization broken (P3826R3, P3718R0, P2999R3, P3303R1). A sender cannot know its completion domain until it knows where it will start. Early customization is, in Eric Niebler's words, "irreparably broken."[20] `starts_on(gpu, just()) | then(fn)` silently uses the CPU implementation for GPU work. Four papers and counting. In regular C++, calling `f(x)` on a thread pool just works.

Incomprehensible diagnostics (P3557R3, P3164R4). Type checking deferred to `connect` time means `just(42) | then([](){})` (nullary lambda, wrong arity) produces dozens of lines of template backtrace far from the source. In regular C++, `g(f(42))` with a type mismatch produces a one-line error at the call site. **`connect_result_t` SFINAE breakage (LWG 4206, Priority 1).** Changing `connect`'s constraints to `Mandates` made `connect` unconstrained, but `let_value` uses `connect_result_t` in SFINAE contexts expecting substitution failure. It gets hard errors instead. The Sub-Language's layered constraint model violates normal template metaprogramming conventions.

6.2 Semantic Friction

`split` and `ensure_started` removed (P3682R0, P3187R1). The two most natural ways to share or reuse an async result violate structured concurrency. Both removed by plenary vote. In regular C++, `auto x = f(); use(x); use(x);` is trivial. The Sub-Language's structured concurrency guarantees make this pattern unsafe.

Optional vs. variant interface mismatch (P3570R2). A concurrent queue's `async_pop` naturally returns `optional<T>` in regular C++ ("value or nothing"). The Sub-Language requires the same operation to complete through `set_value(T)` or `set_stopped()`, two channels dispatched at the type level. API designers must choose which world to serve.

Partial success in pure sender context (P2430R0). This is the same error channel mismatch documented in Section 3, viewed from the sender side rather than the coroutine side. The forced choice between `set_value(ec, n)` and `set_error(ec)` exists for raw sender authors writing operation states, independent of coroutines. We include it here to show that the friction is not specific to the coroutine boundary.

6.3 Specification Friction

`transform_sender` dangling reference (LWG 4368, Priority 1). The specification itself has a use-after-free. The layered transform architecture creates a lifetime hazard: `default_domain::transform_sender` forwards a temporary as an xvalue after the temporary is destroyed. The reference implementation (stdexec) works around it by non-conformantly returning prvalues. This hazard does not exist in direct function calls.

Circular `completion_signatures_for` (LWG 4190, Priority 2). The spec for `completion_signatures_for<Sndr, Env>` tests `sender_in<Sndr, Env>`, which requires `get_completion_signatures(sndr, env)` to be well-formed, the very thing being defined. Circular specifications are a symptom of the type-level machinery's self-referential complexity.

6.4 Viral Friction

Environment leaks into `task` (P3552R3). In a sender pipeline, the receiver's environment propagates through `connect` and never appears in a public type. When the Sub-Language meets coroutines, the environment has nowhere to hide: a coroutine's return type is its public interface. P3552R3 defines `task` with two template parameters: `template<class T, class Environment>`. Every production coroutine library has independently converged on a single parameter:

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, libcoro, asyncpp, aiopp, P3552R3 ref. impl.

The two-parameter form appears only in P3552R3. A default `Environment` resolves the spelling but not the composability: a library returning `task<T>` with the default environment cannot participate in sender pipelines that depend on it. The pattern is the same as Sections 3-5: a property internal to the Sub-Language becomes a cost at the boundary with regular C++.

Is the pattern coincidental? If the committee determines these items are routine specification work, that finding does not affect the three structural gaps in Sections 3-5. Those gaps stand on their own evidence.

---

7. The Gaps Are The Tradeoff

The committee standardized the Sender Sub-Language 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.

7.1 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

7.2 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);`

7.3 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 offers `allocator_arg` for the initial allocation, but propagation remains unsolved. P3826R3 offers five solutions for algorithm dispatch; none changes when the allocator becomes available (Appendix A.3).

7.4 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 7.1 through 7.3 show it is not. Each gap is the cost of a specific design property. Closing the gap means removing the property.

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.

---

8. Why Wait To Ship?

The cost of shipping is documented above. What is the cost of waiting?

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

C++ needs a standard task. The question is whether this task unifies or fragments.

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

Convention divergence is the opposite of unification.

8.2 "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 7). Closing the gap means removing the property. ABI lock-in makes the choice permanent (subsection 7.4). The committee deferred P2300 from C++23 for the same pattern of ongoing design changes.

The committee has been here before.

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

Coroutine interop requires the awaitable protocol, not type identity.

P3552R3 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."
- `as_awaitable` converts any sender into this protocol. `affine_on` wraps any co_awaited sender to reschedule the coroutine onto its original scheduler. It exists because `task` co_awaits senders it does not own. If interop required type identity, `affine_on` would have nothing to operate on.

Open task types ship in cppcoro, Boost.Cobalt, libunifex, folly::coro, QCoro, and asyncpp. Each interoperates through the awaitable protocol.

8.4 "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 the costs documented in Sections 3-6 are temporary. They are structural.

---

9. 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. Dietmar Kühl returned to the problem in D3980R0 (2026-01-25), reworking the allocator propagation model six months after P3552R3's adoption 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.*

The earlier design approval poll for P3552R1 was notably soft: SF:5 / F:6 / N:6 / A:1 / SA:0, six neutral votes matching six favorable votes. C++29 forwarding was unanimous. C++26 was conditional and weak. Dietmar's P3796R1 ("Coroutine Task Issues") catalogues sixteen open concerns about `task` - a candid assessment from the task author himself. P3801R0 ("Concerns about the design of `std::execution::task`," Müller, 2025) was filed in July 2025. P2300 was previously deferred from C++23 for maturity concerns; the same pattern of ongoing design changes is present again.

---

10. Working With the Grain

The three tradeoffs in Section 7 are the cost of treating the Sender Sub-Language as the universal model of asynchronous computation. If that assumption is relaxed, the design space opens for coroutine I/O and for senders alike.

10.1 Senders in Their Element

Herb Sutter reported that Citadel Securities uses `std::execution` 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."*** This confirms senders work well in their own domain: compile-time work graph construction, GPU dispatch, high-frequency trading pipelines.

10.2 What If The Design Space Opens?

Section 10.3 shows what coroutine I/O gains. But senders gain freedom too. Much of the specification friction documented in Section 6 - broken algorithm customization across four papers, removed primitives, the `transform_sender` dangling reference - exists at the boundary with domains the Sub-Language was not built to serve. Narrow the scope, and the boundary recedes. Its authors would be free to optimize for what senders do well, without carrying the weight of what they do not.

P4003R0 ("IoAwaitables: A Coroutines-Only Framework") is not proposed for standardization. The code is real, compiles on three toolchains, and comes with benchmarks and unit tests. We show it as evidence that when the universality constraint relaxes, the three tradeoffs become avoidable - not necessarily through this particular

framework, but through approaches like it.

10.3 All Problems Become Solvable

Here is what a coroutine I/O 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 at launch site
    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);
    buf.commit(n);                          // partial success: use what arrived
    if (ec) co_return;                      // then check the status

    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);
}
```

Every tradeoff from Section 7 resolves - not by choosing one column over the other, but by getting both. Errors and byte counts return together, and the coroutine branches on the error at runtime (7.1: both columns). `co_return` handles all completions, matching established practice across every production coroutine library (7.2: both columns). The allocator is set once at the launch site and reaches every frame automatically - function signatures express only their purpose (7.3: both columns). The task type is `template<class T> class task`, one parameter.

P4003R0 achieves this by using `thread_local` propagation to deliver the allocator to `promise_type::operator new` before `connect()`.

P4003R0 does not provide compile-time work graph construction, zero-allocation pipelines, or vendor extensibility. It does not need to. Those properties belong to senders. The three tradeoffs are the cost of universality, not the cost of asynchronous C++.

---

11. Suggested Straw Polls

Diagnosis:

> *"The friction between senders and coroutines comes from the design, not the implementation."*

Task type:

> *"Error signaling in coroutines should use `co_return`, not `co_yield`."*

> *" `std::execution::task` should continue iterating for C++29 rather than shipping in C++26."*

Framework:

> *"Completion channels should preserve partial results alongside errors."*

> *"Coroutines should receive the allocator as cleanly as senders do."*

Broader question:

> *" `std::execution` does not have to extend to all asynchronous C++."*

> *"There is room for a coroutine-native I/O model alongside `std::execution` ."*

---

12. Conclusion

The decision to adopt `std::execution` has consequences for coroutines. This paper asks the committee to address them explicitly.

1. **Is the Sender Sub-Language 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 coroutines be required to work through the Sender Sub-Language to access asynchronous I/O?** The SG4 poll (Kona 2023, SF:5/F:5/N:1/A:0/SA:1) answers implicitly. This paper asks the committee to answer explicitly.
4. **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 straw polls in Section 11 offer the committee a way to record its answers. The deeper question is whether there is room in the standard for `std::io` alongside `std::execution`, each serving its own domain.

Appendix A - Code Examples

A.1 `async_read` as a Sender

Section 3.4 shows the completion callback where accumulated read progress is lost. Here is the complete sender implementation using P2300R10 Section 1.4'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.2 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:

```
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(
        ex::on(sch, read_data(sock, buf)),
        scope.get_token());
}
```

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

A.3 The Full Ceremony for Allocator-Aware Coroutines

The Sub-Language requires five layers of machinery to propagate a custom allocator through a coroutine call chain:

```
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.4 P3826R3 and Algorithm Dispatch

P3826R3 addresses 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. This restructuring could enable future allocator solutions, but none has been proposed.

A.5 Why the Three-Channel Model Exists

The Sub-Language 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
```

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 Sub-Language 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 Sub-Language 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.

Appendix B - 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, the LWG issues list, and the C++26 national body ballot comments.

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

LWG Issues

28. LWG 4368 - "Potential dangling reference from `transform_sender`" (Priority 1)
29. LWG 4206 - "`connect_result_t` should be constrained with `sender_to`" (Priority 1)
30. LWG 4190 - "`completion-signatures-for` specification is recursive" (Priority 2)
31. LWG 4215 - "`run_loop::finish` should be `noexcept`"
32. LWG 4356 - "`connect()` should use `get_allocator(get_env(rcvr))`"

Blog Posts

33. Eric Niebler, "Structured Concurrency" (2020)
34. Eric Niebler, "What are Senders Good For, Anyway?" (2024)
35. Herb Sutter, "Trip report: Summer ISO C++ standards meeting (St Louis, MO, USA)" (2024)
36. Herb Sutter, "Living in the future: Using C++26 at work" (2025)

Programming Language Theory

37. Olivier Danvy, "Back to Direct Style" (1992)
38. Gordon Plotkin, "Call-by-Name, Call-by-Value and the lambda-Calculus" (1975)
39. Christopher Strachey & Christopher Wadsworth, "Continuations: A Mathematical Semantics for Handling Full Jumps" (1974)
40. Eugenio Moggi, "Notions of Computation and Monads" (1991)
41. John Reynolds, "The Discoveries of Continuations" (1993)
42. Guy Steele & Gerald Sussman, The Lambda Papers (1975-1980)

Other

43. C++ Core Guidelines - F.7, R.30 (Bjarne Stroustrup, Herb Sutter, eds.)
44. Butler Lampson, "Hints for Computer System Design" (1983)
45. Continuation-passing style - Wikipedia. https://en.wikipedia.org/wiki/Continuation-passing_style
46. C++ Working Draft - (Richard Smith, ed.). https://eel.is/c++draft/
47. cppreference - C++ reference documentation. https://en.cppreference.com/
48. stdexec - NVIDIA reference implementation of std::execution. https://github.com/NVIDIA/stdexec
49. C++26 NB ballot comments - National body comments repository. https://github.com/cplusplus/nbballot
50. WG21 paper mailings - ISO C++ committee papers. https://open-std.org/jtc1/sc22/wg21/docs/papers/
51. LWG issues list - Library Working Group active issues. https://cplusplus.github.io/LWG/lwg-toc.html