WG21 PROPOSAL

Is the Sender Sub-Language C++?

Document Number:	P4007R1
Date:	2026-02-16
Audience:	SG1, LEWG

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

Table of Contents

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.

---

2. Two Models of Computation

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

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

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, Technical Fellow at Citadel Securities, called `std::execution` ***"the biggest usability improvement yet to use the coroutine support we already have."***

`just` is Haskell's `return` / `pure`, lifting a value into a monadic context. `let_value` is monadic bind (`>>=`), threading a value into the next computation. `then` is fmap, applying a function inside a context. The three completion channels are a fixed algebraic effect system. `connect` reifies the continuation into a concrete operation state. This is continuation-passing style expressed as composable value types. P4014R0 ("The Sender Sub-Language") provides the full treatment.

Coroutines inhabit direct-style C++, code where values return to callers, errors propagate through the call stack, and resources are scoped to lexical lifetimes. This paper will call that ***regular C++*** for the remainder.

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`. If that integration layer has structural gaps, every C++ developer who writes networked code with coroutines will encounter them.

Niebler described what regular C++ means for async programming: ***"We sprinkle `co_await` in our code and we get to continue using all our familiar idioms: exceptions for error handling, state in local variables, destructors for releasing resources, arguments passed by value or by reference, and all the other hallmarks of good, safe, and idiomatic Modern C++."***

Regular C++ language features such as structured bindings, `if` with initializer, and range-for do not apply to sender pipelines, which pass results through continuations rather than returning them.

For most C++ developers, coroutines are already at the limit of complexity they can reasonably manage; the Sender Sub-Language is beyond it. The committee chose to standardize both. This paper examines what follows.

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."***

---

3. 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 4-6 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. `when_all` dispatches on which channel fired at the type level, cancelling siblings on error without inspecting the payload. A coroutine has no way to express this. The transform goes one way. ***Prediction: the three-channel completion model will force I/O sender authors into a choice where neither channel is correct.*** Section 4 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 5 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 6 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.

Bob Nystrom, "What Color is Your Function?" (2015): ***"You still have divided your entire world into asynchronous and synchronous halves and all of the misery that entails."*** The Sub-Language is the "red" world. Regular C++ is the "blue" world.

The committee put a CPS sub-language into C++. Wherever the two meet, we can expect friction. Coroutines are the first boundary, but not the last. Any C++ construct that assumes values return to callers will encounter the same mismatch. Sections 4-6 test whether these predictions hold.

Haskell's `do` -notation bridges CPS and direct-style, but Haskell is pure and `do` -notation is a language feature designed for that purpose. C++ coroutines are neither.

C++ is the only language strongly typed enough to host a genuine CPS sub-language and memory unsafe enough that the boundary creates lifetime hazards no other language faces. LWG 4368 (dangling reference from `transform_sender`) is one.

Are the predictions accurate?

---

4. Where Does the Error Code Go?

The Sub-Language the committee adopted has three completion channels: `set_value`, `set_error`, and `set_stopped`. These channels are its type system. Type-level routing is a requirement of CPS reification (Appendix A.4). 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."*

4.1 Senders

Every I/O operation in Boost.Asio completes with `void(error_code, size_t)`. Partial success is the normal case: a `read` may transfer 47 bytes before EOF. The error code and byte count are returned together. P2762R2 ("Sender/Receiver Interface for Networking") preserves this pattern.

A developer implementing an async read sender must choose. The natural instinct is to route the error code through `set_error`:

```
if (!ec)
    set_value(std::move(rcvr_), n);
else
    set_error(std::move(rcvr_), ec); // bytes lost
```

But this loses the byte count on error. A `read` that transferred 47 bytes before EOF is now indistinguishable from a `read` that transferred zero.

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

```
set_value(std::move(rcvr_), ec, n);
```

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

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

For I/O, `set_error` is the natural choice and the wrong one. The developer who follows the API's naming convention produces code that silently misbehaves under composition.

4.2 Coroutines

The same forced choice reaches coroutine authors through `std::execution::task`. P3552R3 ("Add a Coroutine Task Type") provides two mechanisms for completing a `task`:

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

An I/O coroutine author must choose between three return types, each with consequences:

Option A: Return both values together.

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

The error is invisible to `when_all`, `upon_error`, `let_error`, and every generic error-handling algorithm in `std::execution`.

Option B: Signal the error through the error channel.

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

The byte count is lost when `co_yield with_error(ec)` fires.

Option C: Return only the byte count.

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

None of these options is correct. The developer chooses which kind of wrong they prefer.

4.3 Is `when_all` Problematic?

`when_all` launches multiple child operations concurrently. If any child completes with `set_error` or `set_stopped`, `when_all` requests cancellation of the remaining children. The channel the sender author chose now determines the behavior of every algorithm that consumes it:

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

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

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

When two libraries choose different conventions and are composed in the same `when_all`, the outcome depends on completion order. Same two EOF events. Same `when_all`. The program is correct on some runs and incorrect on others.

4.3.1 Mixed Conventions

Suppose Library A follows the P2762R2 convention (error in `set_value`) and Library B follows P2300R10's own naming convention (error in `set_error`):

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

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

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

If `read_b` hits EOF first: `when_all` sees `set_error`, cancels `read_a`. The 47 bytes already transferred are discarded. If `read_a` hits EOF first: `when_all` sees `set_value`, keeps waiting for `read_b`. Same two EOF events. Correctness determined by which socket completes first.

4.4 The Problem Is Structural

The mismatch is not a missing convention waiting to be supplied. 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 error codes.** Every generic error-handling algorithm in the Sub-Language is inaccessible to I/O senders. The error channel exists but I/O never uses it.
- **Converge on `set_error` for error codes.** Partial success is inexpressible. The byte count is lost.
- **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 and errors are distinct. In I/O they are not. EOF is information, not failure. The model and the domain are incompatible. Appendix A.4 explains why.

4.5 Always Use `set_value`?

One might argue that the channels are generic and I/O should simply adopt the `set_value` convention.

But `when_all` still breaks: when every I/O sender delivers `(error_code, size_t)` through `set_value`, I/O failure is indistinguishable from I/O success. Sibling cancellation on failure is impossible.

The completion signature `void(error_code, size_t)` predates senders by 25 years. Composed I/O operations like Asio's `async_read` accumulate bytes across multiple system calls; when a connection resets mid-transfer, the error code and the byte count are both meaningful.

Kohlhoff identified the tension in P2430R0 (2021). We extend his analysis to composition (Sections 4.3 and 4.3.1).

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.

4.6 P2430R0

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.

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

---

5. 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.

5.1 Senders

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

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

The call is direct.

5.2 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;
}
```

Yet P3552R3 innovates:

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

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.

5.3 Established Practice

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

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

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`.

P3552R3 needed a different mechanism. The only other coroutine keyword that accepts an expression and passes it to the promise is `co_yield`. P3552R3 exploits `yield_value` with a special overload for `with_error<E>` ([task.promise]/7):

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

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. Jonathan Wakely'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?

---

6. 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.

Eric Niebler wrote in 2021: ***"The overwhelming benefit of coroutines in C++ is its ability to make your async scopes line up with lexical scopes."***

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	-

6.1 Senders

The P2300 authors solved the allocator propagation problem for senders with care and precision. The receiver's environment carries the allocator via `get_allocator(get_env(rcvr))`, and the 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 in the tree without any intermediate sender mentioning it. This is a clean design.

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.

6.2 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.

6.3 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.

Software engineering calls this tramp data. The React ecosystem calls it prop drilling.

C++ Core Guidelines F.7 says: a function should not be coupled to the caller's ownership policy.

Meredith & Halpern, "Getting Allocators out of Our Way" (CppCon 2019), identified "enlarged interfaces and contracts" as the cost. 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. The same framework, the same resource, two completely different levels of support.

6.4 Does Performance Matter?

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

6.5 A Viral Signature?

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

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

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

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

Every function signature carries allocator forwarding machinery that has nothing to do with the function's purpose.

6.6 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.

Is this how the committee intended coroutines to be used?

7. Friction Beyond Coroutines

Sections 4-6 demonstrate friction at the coroutine boundary. The same mismatch generates friction within the Sub-Language's own machinery.

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. Large features generate proportionate review activity. The question is whether the classification pattern is better explained by scale or by structure.

7.1 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." `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.

7.2 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 4, 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.

7.3 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.

7.4 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 4-6: a property internal to the Sub-Language becomes a cost at the boundary with regular C++.

Is the pattern coincidental? If the committee determines that these items are routine specification work unrelated to the model of computation, that finding does not affect the three structural gaps documented in Sections 4-6. Those gaps stand on their own evidence. The question this section poses is narrower: does the pattern, nine items from four independent categories, each clustering at the boundary between the Sub-Language and regular C++, suggest a cause, or is it coincidence? The authors leave the classification to the committee.

---

8. 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 4-6 is the cost of one of those properties. Closing any gap requires removing the property it pays for.

8.1 The Error Channel Tradeoff

The three-channel model enables compile-time routing: `when_all` cancels siblings on error, `upon_error` attaches at the type level, `let_value` chains successes - all without runtime inspection. Section 4 showed the cost: I/O operations that return `(error_code, size_t)` together cannot be expressed in the model.

Choose one:

Compile-time error routing	I/O partial success
`when_all` cancels siblings at the type level; `upon_error` attaches without runtime inspection	`(error_code, size_t)` returned together; 47 bytes before EOF is distinguishable from zero

Removing the three-channel model eliminates the cost but also eliminates compile-time routing. The model and the gap are the same mechanism viewed from different sides.

8.2 The `co_yield` Tradeoff

The separate error channel requires a separate delivery mechanism from coroutines. `co_return` maps to `return_value` / `return_void` , which maps to `set_value` . The language provides no second return path. Section 5 showed the cost: `co_yield with_error` repurposes a keyword with semantics no other coroutine library has used. The alternative is a language change (lifting the `return_void` / `return_value` mutual exclusion), which is not part of C++26 and has no paper proposing it.

Choose one:

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

8.3 The Allocator Tradeoff

CPS reification (`connect` / `start`) defers execution until the receiver's environment is available. This enables zero-allocation sender pipelines and automatic scheduler propagation. Section 6 showed the cost: the coroutine's `promise_type::operator new` fires at the function call, before `connect()` runs. The allocator the receiver carries is structurally unavailable.

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: in `co_await child_coro(args...)` , the child's `operator new` fires before `co_await` processing begins. P3552R3 offers `allocator_arg` for the initial allocation, but propagation remains unsolved - no `uses_allocator` equivalent exists for coroutines. P3826R3 offers five solutions for algorithm dispatch; none changes when the allocator becomes available (see Appendix A.3).

8.4 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 8.1 through 8.3 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.

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. D3980R0 (Kuhl, 2026-01-25) subsequently reworked the allocator propagation model relative to P3552R3, adopted only six months earlier at Sofia. LWG 4356 confirms the gap has been filed as a specification defect.

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

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

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. P3796R1 ("Coroutine Task Issues") catalogues sixteen distinct open concerns about `task`. P3801R0 ("Concerns about the design of `std::execution::task`," Jonathan Wakely, 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 8 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 already 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. Working with the grain of CPS, the model delivers exactly what it was designed to deliver.

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 7 - 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 8 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 (8.1: both columns). `co_return` handles all completions, matching established practice across every production coroutine library (8.2: both columns). The allocator is set once at the launch site and reaches every frame automatically - function signatures express only their purpose (8.3: 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()`. The timing gap is solvable when the promise cooperates with a non-receiver mechanism.

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, and senders have them. The three tradeoffs are the cost of universality, not the cost of asynchronous C++. Relax the constraint, and the cost disappears.

---

11. Suggested Straw Polls

We ask the committee to consider the following straw polls, which address the consequences documented in this paper.

Diagnosis:

> *"The pattern documented in Section 7 and Appendix B, friction points clustering at the boundary between the Sender Sub-Language and regular C++, is best explained as a structural consequence of integrating a CPS-based model with direct-style C++, rather than typical end-of-cycle specification polish."*

Task type:

> *"The ergonomic cost of `co_yield with_error` is an acceptable trade-off for `std::execution::task` in C++26."*

> *" `std::execution::task` should not ship in C++26. The coroutine integration should iterate independently for C++29."*

Framework:

> *"The behavior of `when_all` under mixed channel conventions is acceptable for I/O use cases."*

> *"The allocator sequencing gap, where the receiver's environment is structurally unavailable at coroutine frame allocation time, is acceptable for C++26."*

Broader question:

> *" `std::execution` should be considered a domain-specific model for compile-time work graph construction, not a universal foundation for all asynchronous C++."*

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

---

12. Conclusion

The committee has adopted the Sender Sub-Language into C++26. This paper does not argue otherwise. The Sub-Language has real value for the domains it serves, and the committee made that decision with good reason.

But the decision has consequences for coroutines, and this paper asks the committee to address those consequences explicitly.

1. **Is the Sender Sub-Language still considered a universal model of asynchronous computation?** The evidence in this paper suggests it serves specific domains, GPU dispatch, high-frequency trading, heterogeneous computing, but not I/O. The three gaps are structural consequences of applying the Sub-Language to a domain whose completion semantics are inherently runtime.

2. **Are coroutines the primary tool for writing asynchronous C++ that most developers will use?** If yes, the coroutine integration deserves the same level of design investment as the sender pipeline itself, not an adaptation layer absorbing every cost of a model it does not use.

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 this implicitly. This paper asks the committee to answer it explicitly, with full knowledge of the costs.

4. Should `task<T>` ship in C++26 with these structural costs, or should the coroutine integration iterate independently? Ship `std::execution` for the domains it serves. Let the coroutine integration develop on its own timeline.

The suggested straw polls in Section 11 offer the committee a way to record its answers.

The namespace `std::execution` claims universality. The evidence suggests the claim is too broad. Coroutine-based I/O is not in the Sub-Language's domain. Perhaps the right question is not how to make `std::execution::task` serve I/O, but whether there is room in the standard for `std::io` alongside `std::execution`, each serving its own domain, with interoperation at the boundary.

Appendix A - Code Examples

A.1 Why HALO Cannot Help

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

```
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.2 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:

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

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

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

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

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

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

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

```
    ex::spawn(std::move(sndr), ctx.get_token());
}
```

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

A.3 P3826R3 and Algorithm Dispatch

P3826R3 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.4 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
```

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

If errors were delivered as values (for example, `expected<int, error_code>` through `set_value`), the compiler would see one path carrying one type. `when_all` could not cancel siblings on error without runtime inspection of the payload. 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.

References

WG21 Papers

1. P2300R10 - "std::execution" (Michal Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Eric Niebler, 2024)
2. P2300R4 - "std::execution" (Michal Dominiak, et al., 2022)
3. P2430R0 - "Partial success scenarios with P2300" (Chris Kohlhoff, 2021)
4. P2762R2 - "Sender/Receiver Interface For Networking" (Dietmar Kuhl, 2023)
5. 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” (Lewis Baker, Eric Niebler, 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 Kuhl, 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” (Eric Niebler, 2025)
17. P3718R0 - “Fixing Lazy Sender Algorithm Customization, Again” (Eric Niebler, 2025)
18. P3796R1 - “Coroutine Task Issues” (Dietmar Kuhl, 2025)
19. P3801R0 - “Concerns about the design of std::execution::task” (Jonathan Wakely, 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 Kuhl, 2026)
23. P3950R0 - “return_value & return_void Are Not Mutually Exclusive” (Robert Leahy, 2025)
24. D3980R0 - “Task’s Allocator Use” (Dietmar Kuhl, 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, “Ranges, Coroutines, and React: Early Musings on the Future of Async in C++” (2017)
34. Eric Niebler, “Structured Concurrency” (2020)
35. Eric Niebler, “Asynchronous Stacks and Scopes” (2021)
36. Eric Niebler, “What are Senders Good For, Anyway?” (2024)
37. Herb Sutter, “Trip report: Summer ISO C++ standards meeting (St Louis, MO, USA)” (2024)
38. Herb Sutter, “Living in the future: Using C++26 at work” (2025)

39. Bob Nystrom, "What Color is Your Function?" (2015)

Programming Language Theory

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

Other

46. C++ Core Guidelines - F.7, R.30 (Bjarne Stroustrup, Herb Sutter, eds.)
47. Herb Sutter, GotW #91: Smart Pointer Parameters (2013)
48. Alisdair Meredith & Pablo Halpern, "Getting Allocators out of Our Way" (CppCon 2019)
49. Gregor Kiczales et al., "Aspect-Oriented Programming" (ECOOP 1997)
50. Butler Lampson, "Hints for Computer System Design" (1983)