WG21 PROPOSAL

Symmetric Transfer and Sender Composition

Document Number:	D2583R1
Date:	2026-02-24
Audience:	LEWG
Reply-to:	Mungo Gill mungo.gill@me.com Vinnie Falco vinnie.falco@gmail.com

Table of Contents

Abstract

C++20 provides symmetric transfer (P0913R1[1]) - a mechanism where `await_suspend` returns a `coroutine_handle<>` and the compiler resumes the designated coroutine as a tail call. Coroutine chains execute in constant stack space. `std::execution` (P2300R10[5]) composes asynchronous operations through sender algorithms. These algorithms create receivers that are structs, not coroutines. No `coroutine_handle<>` exists at any intermediate point in a sender pipeline. When a coroutine `co_await`s a sender that completes

synchronously, the stack grows by one frame per completion. P3552R3[2]'s `std::execution::task` inherits this property. A protocol-level fix exists: completion functions and `start()` return `coroutine_handle<>` instead of `void`, enabling struct receivers to propagate handles from downstream without becoming coroutines. The fix preserves zero-allocation composition but requires changing the return type of every completion function, every `start()`, and every sender algorithm in P2300R10[5]. This paper documents the gap, describes the fix, and enumerates the changes required.

Revision History

R1: February 2026 (Croydon)

- Corrected Section 5's characterization: struct receivers cannot produce a `coroutine_handle<>`, but they can propagate one from downstream. The gap is a protocol choice, not a structural impossibility.
- Added handle propagation fix (Section 10).
- Added scope of changes analysis (Section 11).
- Added ABI considerations (Section 12).
- Corrected date of R0 revision (March -> February).

R0: February 2026 (pre-Croydon mailing)

- Initial version.

---

1. Disclosure

The authors developed P4003R0[3] ("Coroutines for I/O") and P4007R0[4] ("Senders and Coroutines"). A coroutine-only design cannot express compile-time work graphs, does not support heterogeneous dispatch, and assumes a cooperative runtime. Those are real costs. We do not claim coroutines are the answer to all problems. The limitation documented here exists independently of any alternative design.

P4007R0[4] documents three costs at the boundary where the sender model meets coroutines: error reporting, error returns, and frame allocator propagation. Each is an interface mismatch. This paper documents a cost inside the composition mechanism. Sender algorithms are structs. The completion protocol is void-returning. These are not boundary properties. They are what sender algorithms are.

---

2. Symmetric Transfer in C++20

Prior to P0913R1[1], coroutines could only return control to their caller or resumer. Gor Nishanov described the problem:

> *"Recursive generators, zero-overhead futures and other facilities require efficient coroutine to coroutine control transfer. Involving a queue and a scheduler makes coroutine to coroutine control transfer inefficient. Coroutines need direct and efficient way of expressing the desired behavior."*

The solution was a third form of `await_suspend`. C++20 provides three:

```
void await_suspend(coroutine_handle<>);                // unconditional suspend
bool await_suspend(coroutine_handle<>);                // conditional suspend
coroutine_handle<> await_suspend(coroutine_handle<>);  // symmetric transfer
```

The third form returns a `coroutine_handle<>`. The compiler resumes the designated coroutine as if by a tail call. The note in [expr.await][16] states:

> *"Any number of coroutines can be successively resumed in this fashion, eventually returning control flow to the current coroutine caller or resumer."*

The stack does not grow. One coroutine suspends and another resumes in constant stack space. This is the mechanism C++20 provides to prevent stack overflow in coroutine chains.

Lewis Baker, author of cppcoro[9] and co-author of P2300R10[5], implemented symmetric transfer in cppcoro and documented the motivation in the source:

> *"We can eliminate the use of the std::atomic once we have access to coroutine_handle-returning await_suspend() on both MSVC and Clang as this will provide ability to suspend the awaiting coroutine and resume another coroutine with a* ***guaranteed tail-call to resume()****."*

The canonical implementation is cppcoro's `final_awaitable`:

```cpp
struct final_awaitable
{
    bool await_ready() const noexcept { return false; }

    template<typename PROMISE>
    std::coroutine_handle<> await_suspend(
        std::coroutine_handle<PROMISE> coro) noexcept
    {
        return coro.promise().m_continuation;
    }

    void await_resume() noexcept {}
};
```

When the coroutine reaches `final_suspend`, `await_suspend` returns the continuation's handle. The compiler performs a tail call to the continuation. No stack frame is added.

3. Production Libraries

Every major C++ coroutine library the authors surveyed uses symmetric transfer in its task type. The following table shows the `await_suspend` return type in each library's `final_suspend` awaiter:

Library	`await_suspend` Return	Mechanism
cppcoro	`coroutine_handle<>`	Returns `m_continuation`
folly::coro	`coroutine_handle<>`	Returns `continuation_.getHandle()`
Boost.Cobalt	`coroutine_handle<>`	Returns `awaited_from` or `noop_coroutine()`
libcoro	`coroutine_handle<>`	Returns `m_continuation` or `noop_coroutine()`
Boost.Capy	`coroutine_handle<>`	Returns `p_->continuation()`
asyncpp	`void`	Event-based notification

Five of six libraries converge on the same mechanism: `await_suspend` returns a `coroutine_handle<>`. This is independent replication. The libraries were developed by different authors, for different platforms, with different design goals. They converged on symmetric transfer because it is the only guaranteed zero-overhead mechanism C++20 provides for preventing stack overflow in coroutine chains.

4. `std::execution` 's Coroutine Bridges

P2300R10[5] achieves zero-allocation composition of asynchronous operations, compile-time work graph construction, and heterogeneous dispatch across execution contexts. It provides two bridges between coroutines and the sender model. Neither uses symmetric transfer.

4.1 Coroutine Completing as Sender

When an awaitable is used where a sender is expected, P2300R10 bridges it via `connect-awaitable` . The completion uses `suspend-complete` :

```
template<class Fun, class... Ts>
auto suspend-complete(Fun fun, Ts&&... as) noexcept {
    auto fn = [&, fun]() noexcept { fun(std::forward<Ts>(as)...); };

    struct awaiter {
        decltype(fn) fn;

        static constexpr bool await_ready() noexcept { return false; }
        void await_suspend(coroutine_handle<>) noexcept { fn(); }
        [[noreturn]] void await_resume() noexcept { unreachable(); }
    };
    return awaiter{fn};
}
```

`await_suspend` returns `void` . The coroutine is unconditionally suspended. The completion function `fn()` - which calls `set_value` , `set_error` , or `set_stopped` on the receiver - runs as a nested function call on the current stack. This is not symmetric transfer.

4.2 Sender Co-Awaited by Coroutine

When a coroutine `co_await` s a sender, P2300R10 bridges it via `sender-awaitable` :

```
class sender-awaitable {
    // ...
    connect_result_t<Sndr, awaitable-receiver> state;

public:
    static constexpr bool await_ready() noexcept { return false; }
    void await_suspend(coroutine_handle<Promise>) noexcept { start(state); }
    value-type await_resume();
};
```

`await_suspend` returns `void`. The sender is started. When the sender completes, the `awaitable-receiver` resumes the coroutine:

```
// awaitable-receiver::set_value (exception handling elided)
rcvr.result-ptr->template emplace<1>(vs...);
rcvr.continuation.resume();
```

The `.resume()` call is a function call within whatever context invokes `set_value`. If the sender completes synchronously inside `start`, the call chain is:

```
await_suspend -> start(state) -> ... -> set_value -> continuation.resume()
```

The stack grows with each synchronous completion. Symmetric transfer would return the continuation handle from `await_suspend`, allowing the compiler to arrange a tail call. The sender model calls `.resume()` as a function call inside `set_value`.

Both bridges use `void await_suspend`. Neither can perform symmetric transfer.

5. Why Sender Algorithms Do Not Provide a Handle

Symmetric transfer requires `await_suspend` to return a `coroutine_handle<>`. The question is where that handle would come from. The answer has two parts.

5.1 The Composition Layer

Sender algorithms (`then` , `let_value` , `when_all`) compose operations by wrapping one receiver in another. The wrapping receiver is a struct. It has `set_value` , `set_error` , and `set_stopped` member functions that invoke the wrapped callable and forward to the next receiver. It is not a coroutine. It has no `coroutine_handle<>` .

A sender pipeline with three algorithms produces a chain: coroutine -> struct -> struct -> struct -> coroutine. The structs are sender algorithm receivers. Under the current protocol, no `coroutine_handle<>` exists at any intermediate point. These struct receivers have void-returning completion functions. No return channel exists. A `coroutine_handle<>` at the end of the chain cannot propagate back through the intermediate structs to `await_suspend` .

Making each sender algorithm a coroutine would produce a handle at every intermediate point. It would also produce a heap-allocated frame at every intermediate point. A struct receiver cannot ***produce*** a `coroutine_handle<>` - it is not a coroutine and has no frame. However, it could ***propagate*** one received from further down the chain, if the protocol provided a return channel. Under the current protocol, no such channel exists. Section 10 describes a protocol-level fix that provides one.

5.2 The Protocol

Even when a receiver IS backed by a coroutine - as `awaitable-receiver` is, holding a `coroutine_handle<Promise> continuation` member - the completion protocol does not help. `set_value` is void-returning:

```
// awaitable-receiver::set_value (exception handling elided)
rcvr.result-ptr->template emplace<1>(vs...);
rcvr.continuation.resume();
```

The handle exists inside the receiver. The protocol calls `.resume()` on it as a function call. It does not return the handle to the caller. The caller of `set_value` - which is the sender algorithm's receiver, which is a struct - cannot receive a `coroutine_handle<>` back from a void-returning function.

Four facts:

1. Sender algorithms create receivers that are structs, not coroutines. These structs have no `coroutine_handle<>` .
2. Even coroutine-backed receivers complete through void-returning `set_value` .
3. The handle exists inside the receiver but the protocol provides no way to return it.
4. `await_suspend` cannot return what neither the composition layer nor the protocol provides.

Could the protocol be changed? A struct receiver cannot produce a `coroutine_handle<>`, but it does not need to. If `set_value` returned `coroutine_handle<>`, each struct receiver could forward the handle it receives from the next receiver downstream. The handle originates from the coroutine-backed receiver at the end of the chain and propagates back through every intermediate struct to `await_suspend`. Under the current protocol, no such return channel exists. `set_value` returns `void`.

6. `std::execution::task` and `affine_on`

P3552R3[2]'s `task` wraps every `co_await`ed sender in `affine_on` for scheduler affinity. The `await_transform` is defined in [task.promise] p9[16]:

```
as_awaitable(affine_on(std::forward<Sender>(sndr), SCHED(*this)), *this)
```

Even when one `task` `co_await`s another `task`, the inner task is wrapped in `affine_on`, producing a different sender type. Dietmar Kühl, a co-author of P3552R3, documented this in P3796R1[6] ("Coroutine Task Issues"), Section 3.2.3:

> *"The specification doesn't mention any use of symmetric transfer. Further, the `task` gets adapted by `affine_on` in `await_transform` ([task.promise] p10[16]) which produces a different sender than `task` which needs special treatment to use symmetric transfer."*

Jonathan Müller demonstrated the consequence in P3801R0[7] ("Concerns about the design of std::execution::task"), Section 3.1:

```
ex::task<void> f(int i);

ex::task<void> g(int total) {
    for (auto i = 0; i < total; ++i) {
        co_await f(i);
    }
}
```

> *"Depending on the value of `total` and the scheduler used to execute `g` on, this can lead to a stack overflow. Concretely, if the `ex::inline_scheduler` is used, each call to `f` will execute eagerly, but, because `ex::task` does not support symmetric transfer, each schedule back-and-forth will add additional stack frames."*

Müller added:

> *"Having iterative code that is actually recursive is a potential security vulnerability."*

7. The Proposed Fix and Its Limits

Kühl suggested a fix direction in P3796R1[6], Section 3.2.3:

> *"To address the different scheduling problems (schedule on `start`, schedule on completion, and symmetric transfer) it may be reasonable to mandate that `task` customises `affine_on` and that the result of this customisation also customises `as_awaitable`."*

This would allow a `task` `co_await`ing another `task` to bypass the sender machinery and use a direct awaiter with symmetric transfer. The fix addresses one case: task-to-task.

It does not address the general case. Müller stated this directly in P3801R0[7], Section 3.1:

> *"A potential fix is adding symmetric transfer to `ex::task` with an `operator co_await` overload. However, while this would solve the example above, it would not solve the general problem of stack overflow when awaiting other senders. A thorough fix is non-trivial and requires support for guaranteed tail calls."*

Three cases remain after the proposed fix:

1. A `task` `co_await`ing a sender that is not a `task` - no symmetric transfer. The sender completes through the receiver, which calls `.resume()` as a function call.

2. A `task` `co_await`ing a `task` whose body `co_await`s a synchronously completing sender (e.g. `co_await just()`) - the inner task's `co_await` goes through the `sender-awaitable` bridge, which uses `void await_suspend`. Stack frames accumulate inside the inner task.

3. Any sender chain where coroutines are composed through sender algorithms (`then` , `let_value` , `when_all`) - each sender algorithm is not a coroutine. It is a struct with `set_value` member functions. No coroutine handle exists as a transfer target.

Kühl acknowledged the general case requires a different mitigation:

> *"There is a general issue that stack size needs to be bounded when operations complete synchronously. The general idea is to use a trampoline scheduler which bounds stack size and reschedules when the stack size or the recursion depth becomes too big."*

A trampoline scheduler is a runtime mitigation. It detects excessive stack depth and reschedules. This is the runtime overhead in the completion path that P0913R1[1] was specifically adopted to eliminate.

Müller identifies a language-level alternative: guaranteed tail calls. If C++ adopted such a feature, `set_value` could transfer control without growing the stack, potentially closing the gap without changing the receiver abstraction. No such feature exists in C++.

7.1 The Cost

A coroutine that `co_await` s N synchronously-completing senders in a loop accumulates O(N) stack frames. With symmetric transfer, the same loop executes in O(1) stack space. The difference is the composition mechanism.

The stack growth is not specific to `inline_scheduler` . Any sender that completes synchronously grows the stack: `just()` , `then()` on a ready value, a cached result, an in-memory lookup.

Two mitigations exist. Both have costs:

Mitigation	Mechanism	Cost
Real scheduler	Every `co_await` round-trips through the scheduler queue	Latency per iteration, even when the work completes immediately
Trampoline	Detect stack depth at runtime, reschedule when threshold is reached	Runtime check on every completion, plus occasional rescheduling latency

Neither mitigation is zero-cost. Symmetric transfer is zero-cost. The difference is the price of composing coroutines through sender algorithms rather than through the awaitable protocol.

Symmetric transfer does not prevent all stack overflow. Infinite recursion exhausts any finite stack regardless. What symmetric transfer prevents is the specific case where a finite coroutine chain overflows due to accumulated frames from non-tail calls. Sender composition creates exactly this case.

8. Failure To Launch

Section 7 showed that the proposed task-to-task fix does not reach the general case of `co_await` ing arbitrary senders. This section shows that the gap extends further: no launch mechanism in P3552R3[2] avoids the sender composition layer.

8.1 Two Entry Points

P3552R3[2] provides two ways to start a `task` from non-coroutine code:

- `sync_wait(task)` [16] - blocks the calling thread until the task completes.
- `spawn(task, token)` [8] ("async_scope") - launches the task into a `counting_scope` .

The paper's only complete example uses `sync_wait` :

```
int main() {
    return std::get<0>(*ex::sync_wait([]->ex::task<int> {
        std::cout << "Hello, world!\n";
        co_return co_await ex::just(0);
    }()));
}
```

8.2 `spawn` Does Not Compile

`task` is scheduler-affine by default. The scheduler is obtained from the receiver's environment when `start` is called. P3552R3[2], Section 4.5:

> *"An example where no scheduler is available is when starting a task on a `counting_scope` . The scope doesn't know about any schedulers and, thus, the receiver used by `counting_scope` when `connect` ing to a sender doesn't support the `get_scheduler` query, i.e., this example doesn't work:"*

```
ex::spawn([]->ex::task<void> { co_await ex::just(); }(), token);
```

The paper acknowledges this is not a corner case:

> *"Using `spawn()` with coroutines doing the actual work is expected to be quite common."*

The working pattern wraps the task in `on`:

```
ex::spawn(ex::on(sch, my_task()), scope.get_token());
```

8.3 `on` Is a Sender Algorithm

`on(sch, sndr)`[16] is a sender algorithm. Its receiver is a struct with void-returning `set_value`. This is the composition layer that Section 5 showed does not support symmetric transfer under the current protocol. The task's completion traverses the `on` algorithm's receiver before reaching the scope's receiver. No `coroutine_handle<>` exists at any point in this path.

`sync_wait` connects to its own receiver - a struct with void-returning `set_value`. The `run_loop` scheduler inside `sync_wait` resumes the coroutine through `.resume()` as a function call.

Both entry points route through sender algorithms. Both use void-returning completions. Neither can perform symmetric transfer.

8.4 Implementation Experience

A coroutine-native launcher avoids the sender pipeline entirely. Boost.Capy[13] starts a task directly on an executor:

```
corosio::io_context ioc;
run_async(ioc.get_executor())(do_session());
ioc.run();
```

The launcher creates a trampoline coroutine that owns the task. The task chain uses symmetric transfer throughout. No sender algorithm participates. The gap documented in Sections 5 through 7 does not arise because the composition layer is the awaitable protocol, not the sender protocol.

This is not an argument for one design over another. It is evidence that the symmetric transfer gap is specific to the sender launch path, not inherent to launching coroutines.

Every path into `std::execution::task` enters the sender composition layer. No path out preserves symmetric transfer.

9. The Symmetric Transfer Gap

The prior papers document the symptom. Kühl's P3796R1[6] states that the specification does not mention symmetric transfer and suggests a partial fix through domain customization of `affine_on`. Müller's P3801R0[7] confirms the partial fix does not reach the general case and identifies guaranteed tail calls as a language-level direction. Neither paper claims the gap is architectural. Both frame it as a missing feature - absent, difficult to add, but not structurally precluded.

This paper makes a stronger claim. The analysis in Section 5 identifies two structural causes. Sender algorithms create receivers that are structs, not coroutines - no `coroutine_handle<>` exists at the composition layer. Even coroutine-backed receivers complete through void-returning `set_value` - the protocol does not return the handle. The gap is not a missing feature. It is a consequence of the design choices that define the sender model.

The distinction matters. If symmetric transfer is merely missing, it can be added independently. If it is a consequence of protocol choices, it can only be closed by changing one of those choices - as Section 10 demonstrates by changing the return type while preserving struct receivers - or by a language feature that does not exist. The committee should know which characterization is correct.

Symmetric transfer requires `await_suspend` to return a `coroutine_handle<>`. Sender algorithms create receivers that are structs. These structs are not coroutines and have no handle. Even when the final receiver is coroutine-backed, `set_value` is void-returning and does not propagate the handle to the caller.

Sender composition	Symmetric transfer
Sender algorithms compose through non-coroutine structs	`await_suspend` must return a `coroutine_handle<>`
Completion is a void-returning call to `set_value`	Completion is a tail jump to the continuation
Zero-allocation pipelines, compile-time work graphs	Constant-stack coroutine chains

Symmetric transfer requires a handle. The sender model provides a function call. Under the current protocol, no bridge exists between the two.

Section 10 describes a protocol-level fix. Completion functions and `start()` return `coroutine_handle<>` instead of `void`. Struct receivers propagate handles from downstream without becoming coroutines. Zero-allocation composition is preserved. The fix closes the gap - but the scope of changes enumerated in Section 11 is pervasive, and Section 12 shows that the change becomes ABI-breaking once `std::execution` ships. If the fix is not adopted, the characterization above applies: the gap is permanent.

Three directions could be explored if the fix is not adopted: domain customization that short-circuits the receiver abstraction for known coroutine types, a language feature for guaranteed tail calls, or coroutine task types that use the awaitable protocol directly for coroutine-to-coroutine composition and reserve the sender model for the boundaries where it is needed.

This is a tradeoff, not a defect. P2300R10[5] was designed for GPU dispatch and heterogeneous computing - domains where symmetric transfer provides no benefit. The cost appears only when the same model is applied to coroutine composition, where symmetric transfer is the primary stack-overflow prevention mechanism that C++20 provides.

The fix is pervasive. Without it, the gap is permanent.

---

10. Handle Propagation

The fix changes the return type of completion functions and `start()` from `void` to `coroutine_handle<>` . A null handle means no transfer is needed. The `coroutine_handle<>` type already provides null semantics: a default-constructed handle is null, and `operator bool` tests for non-null.

10.1 The Mechanism

Three changes to the protocol:

1. `set_value` , `set_error` , and `set_stopped` return `coroutine_handle<>` instead of `void` .
2. `start()` returns `coroutine_handle<>` instead of `void` .
3. The `sender-awaitable` bridge uses the symmetric transfer form of `await_suspend` .

The coroutine-backed receiver at the end of a sender pipeline returns its continuation handle instead of calling `.resume()` :

```
// current awaitable-receiver
void set_value(Args&&... args) noexcept {
    result_ptr_->template emplace<1>(std::forward<Args>(args)...);
    continuation_.resume();
}

// proposed awaitable-receiver
coroutine_handle<> set_value(Args&&... args) noexcept {
    result_ptr_->template emplace<1>(std::forward<Args>(args)...);
    return continuation_;
}
```

The `sender-awaitable` bridge uses the third form of `await_suspend` from P0913R1[1] ("Add symmetric coroutine control transfer"):

```
// current sender-awaitable
void await_suspend(coroutine_handle<Promise>) noexcept {
    start(state_);
}

// proposed sender-awaitable
coroutine_handle<> await_suspend(coroutine_handle<Promise>) noexcept {
    auto h = start(state_);
    return h ? h : noop_coroutine();
}
```

If `start()` completes synchronously and the downstream receiver returns a handle, the handle propagates through `start()` to `await_suspend`, which returns it for symmetric transfer. If the operation is pending, `start()` returns a null handle, and `await_suspend` returns `noop_coroutine()` to suspend the coroutine.

10.2 `then`: Propagation

A `then` receiver calls its callable, then forwards the result to the next receiver. The handle originates downstream and propagates upward:

```
// current then_receiver
void set_value(Args&&... args) noexcept {
    auto result = invoke(f_, std::forward<Args>(args)...);
    execution::set_value(next_, std::move(result));
}

// proposed then_receiver (exception handling elided)
coroutine_handle<> set_value(Args&&... args) noexcept {
    auto result = invoke(f_, std::forward<Args>(args)...);
    return execution::set_value(next_, std::move(result));
}
```

Exception handling is elided for clarity. If `invoke(f_, args...)` throws, the implementation catches the exception and routes it through `set_error(next_, current_exception())`, which also returns `coroutine_handle<>`. The protocol is self-consistent across error boundaries: the handle from `set_error` propagates identically to one from `set_value`.

The `then` receiver does not produce a handle. It returns whatever the next receiver returned. The handle originates from the coroutine-backed receiver at the end of the chain and propagates through every intermediate struct receiver to `start()`, to `await_suspend`, and into the compiler's symmetric transfer machinery.

10.3 `when_all`: Partial Completion

A `when_all` receiver stores its values and decrements a counter. Only the last completing sub-sender propagates the handle:

```
coroutine_handle<> set_value(Args&&... args) noexcept {
    store(std::forward<Args>(args)...);
    if (--count_ == 0)
        return execution::set_value(final_, get_values());
    return {};
}
```

Receivers that are not the last to complete return a null handle. The caller receives null, propagates it through `start()`, and `when_all`'s own `start()` proceeds to the next sub-sender. Only the final completion produces a non-null handle.

The `set_stopped` path requires care. When a sub-sender completes with an error or cancellation, `when_all` calls `request_stop()` on the remaining sub-senders. Stop callbacks registered by those sub-senders may fire synchronously inside `request_stop()`. P2300's stop callbacks are `void`-returning. If a sibling sub-sender completes inside a stop callback and its counter decrement is the last one, the returned handle has no propagation path through the `void` -returning callback.

The ordering constraint resolves this: the counter decrement must occur after `request_stop()` returns.

```
coroutine_handle<> set_stopped() noexcept {
    stopped_.store(true, std::memory_order_relaxed);
    stop_source_.request_stop();
    if (--count_ == 0)
        return execution::set_stopped(final_);
    return {};
}
```

Sibling completions that fire synchronously inside `request_stop()` decrement the counter but cannot be the last decrement, because the caller's own decrement has not yet occurred. The last decrement always occurs in a normal completion context where the returned handle can propagate.

10.4 `let_value` : Dynamic Chains

A `let_value` receiver calls its factory function, connects the resulting sender, and starts the new operation:

```
// exception handling elided
coroutine_handle<> set_value(Args&&... args) noexcept {
    auto sndr = invoke(f_, std::forward<Args>(args)...);
    auto& op = emplace_op(connect(std::move(sndr), next_));
    return start(op);
}
```

Exception handling is elided. If `invoke(f_, args...)` or `connect` throws, the implementation catches the exception and routes it through `set_error(next_, current_exception())` . As with `then` , both completion channels return `coroutine_handle<>` and the handle propagates identically regardless of which channel produced it.

The inner sender's `start()` may complete synchronously and return a handle. The `let_value` receiver propagates it. The pattern is the same: return what is received from downstream.

10.5 Synchronous and Asynchronous Paths

Two paths through the protocol:

Synchronous. The sender completes inside `start()`. The handle propagates from the receiver through `start()` to `await_suspend`, which performs symmetric transfer. The stack does not grow.

Asynchronous. The sender does not complete inside `start()`. `start()` returns a null handle. `await_suspend` returns `noop_coroutine()`. The coroutine suspends. When the asynchronous operation later completes, the completion context calls `set_value` on the receiver, receives the continuation handle, and calls `.resume()`. The coroutine resumes on whatever thread the completion occurred.

The synchronous path uses symmetric transfer. The asynchronous path uses `.resume()`. Both are correct. The stack growth documented in Sections 5 through 8 occurs only on the synchronous path, and handle propagation eliminates it.

Every coroutine bridge point where `await_suspend` receives a handle from `start()` or a completion function must convert a null handle to `noop_coroutine()`. The `sender-awaitable` bridge in Section 10.1 demonstrates the pattern. The `connect-awaitable` / `suspend-complete` bridge requires the same conversion. This is a general invariant of the proposed protocol: null means "no transfer needed," and `noop_coroutine()` is the mechanism by which the compiler suspends the coroutine without transferring to another.

10.6 Zero Allocation Preserved

Sender algorithm receivers remain structs. No coroutine frames are allocated at intermediate points. The return value is a `coroutine_handle<>` - a pointer-sized value passed on the stack. The compile-time type-level composition is unchanged. The zero-allocation property is preserved.

10.7 Conditional Return Types

The proposed protocol changes every completion function to return `coroutine_handle<>` unconditionally. An alternative approach is possible: make the return type a compile-time property of the receiver. Receivers that participate in coroutine composition would return `coroutine_handle<>`; receivers that do not (terminal receivers, GPU-targeted receivers) would return `void`. Sender algorithms, which are already templates parameterized on the receiver type, could branch on a trait to select the return type at compile time.

This approach would eliminate the null-handle check at async completion sites for non-coroutine paths. The cost of the unconditional approach on those paths is one branch per completion - a `test` / `jz` pair on x86-64 - which is small but not zero.

The conditional approach doubles the implementation surface: every sender algorithm must implement two code paths for every completion function, one returning `coroutine_handle<>` and one returning `void`. It also introduces a trait that must propagate correctly through every wrapper receiver in the chain. A wrapper that fails to propagate the trait silently disables symmetric transfer for the entire pipeline, with no compile-time diagnostic. We note these costs without implementation experience in P2300R10[5]. The P2300 architects are in a better position to determine whether this approach is viable within the existing algorithm implementations.

---

11. Scope of Changes to `std::execution`

The fix requires changing the return type of every completion function and every `start()` in the sender model. The following enumeration covers P2300R10[5] and P3552R3[2] (“Add a Coroutine Task Type”).

11.1 Concepts

The `receiver` and `operation_state` concepts constrain the return types of completion functions and `start()`. Both currently require `void`:

Concept	Expression	Current	Proposed
`receiver`	`set_value(rcvr, args...)`	`void`	`coroutine_handle<>`
`receiver`	`set_error(rcvr, err)`	`void`	`coroutine_handle<>`
`receiver`	`set_stopped(rcvr)`	`void`	`coroutine_handle<>`
`operation_state`	`start(op)`	`void`	`coroutine_handle<>`

Every type that models `receiver` or `operation_state` - in the standard library and in user code - must change.

11.2 Sender Factory Algorithms

Sender factories complete inside `start()`. The return type of `start()` changes from `void` to `coroutine_handle<>`. The factory propagates whatever the receiver’s completion function returns.

Example - `just`:

```
// current
void start() noexcept {
    execution::set_value(rcvr_, values_...);
}

// proposed
coroutine_handle<> start() noexcept {
    return execution::set_value(rcvr_, values_...);
}
```

Affected factories:

- `just`
- `just_error`
- `just_stopped`
- `read_env`

11.3 Sender Adaptor Algorithms

Sender adaptors create internal receivers that forward completions to the next receiver. Every internal receiver's `set_value`, `set_error`, and `set_stopped` must change return type. Every operation state's `start()` must change return type.

Affected adaptors:

- `then`
- `upon_error`
- `upon_stopped`
- `let_value`
- `let_error`
- `let_stopped`
- `bulk`
- `split`
- `ensure_started`
- `into_variant`
- `stopped_as_optional`
- `stopped_as_error`

11.4 Scheduler Algorithms

Scheduler algorithms create internal receivers that manage transitions between execution contexts. Every internal receiver's completion functions and every operation state's `start()` must change return type.

Affected algorithms:

- `starts_on`
- `continues_on`
- `on`
- `schedule_from`
- `affine_on` (P3552R3[2])

11.5 Multi-Sender Algorithms

Multi-sender algorithms create per-sub-sender receivers with counter-based fan-in logic. Every per-sub-sender receiver's completion functions and every operation state's `start()` must change return type.

Affected algorithms:

- `when_all`
- `when_all_with_variant`

11.6 Consumer Algorithms

Consumer algorithms provide terminal receivers. These receivers store results and signal completion (e.g. through a condition variable or `run_loop`). Their completion functions return a null handle - there is no coroutine to transfer to at the terminal point.

Affected consumers:

- `sync_wait`
- `sync_wait_with_variant`

11.7 Coroutine Integration

The coroutine bridges in P2300R10[5] and P3552R3[2] are the primary beneficiaries of the fix. The following must change:

- `sender-awaitable`: `await_suspend` changes from `void` return to `coroutine_handle<>` return (Section 10.1).

- `connect-awaitable` / `suspend-complete` : `await_suspend` changes from `void` return to `coroutine_handle<>` return.
- `task` promise type: internal awaiters and completion machinery must propagate handles.
- `awaitable-receiver` : `set_value` , `set_error` , and `set_stopped` return the continuation handle instead of calling `.resume()` (Section 10.1).

11.8 New Correctness Requirement

The current protocol has one completion path: the receiver calls `.resume()` or signals a synchronization primitive. The proposed protocol has two:

1. **Synchronous completion.** The receiver returns a handle through `set_value` . The caller propagates it through `start()` . `await_suspend` performs symmetric transfer.
2. **Asynchronous completion.** An I/O completion handler, thread pool callback, or scheduler invokes `set_value` on the receiver outside of `start()` . The caller receives the handle and must call `.resume()` .

Every asynchronous completion context must check the returned handle and call `.resume()` if it is non-null. Failure to do so silently leaves the coroutine suspended. The program does not crash. It hangs. The requirement does not exist in the current protocol.

The synchronous path is self-correcting: handles propagate through return values and the compiler's symmetric transfer machinery consumes them. The asynchronous path requires explicit action from the caller of `set_value` . Standard library algorithms (sender factories, adaptors, consumers) are written once and reviewed. Custom senders that wrap OS I/O operations, timers, or other async primitives are the risk population. An author who tests with synchronous completions will see correct behavior; the bug manifests only when the sender completes asynchronously.

The return types of `set_value` , `set_error` , `set_stopped` , and `start()` should be marked `[[nodiscard]]` . A discarded `coroutine_handle<>` return is always a bug under this protocol. The `[[nodiscard]]` attribute catches the most common form of misuse - calling a completion function as a statement without using the return value - at compile time.

11.9 Third-Party Impact

The concept changes in Section 11.1 affect every type that models `receiver` or `operation_state` , including types outside the standard library. The following must be updated:

- stdexec[15] (NVIDIA's reference implementation)
- Every stdexec-based project
- Every user-written sender algorithm

- Every user-written receiver
- Every user-written operation state
- Every custom scheduler's `schedule` sender

Code that models the current `receiver` or `operation_state` concepts with `void` -returning functions will fail to compile against the updated concepts.

---

12. ABI Considerations

Changing the return type of a function changes its calling convention and, for template specializations, its mangled symbol name. Code compiled against `void` -returning `set_value` , `set_error` , `set_stopped` , and `start()` cannot link with code compiled against `coroutine_handle<>` -returning versions. Template instantiations of sender algorithms in different translation units must agree on the return types of receiver completion functions.

`std::execution` is in the C++26 working draft. Once a standard ships function signatures, subsequent changes to those return types are ABI-breaking. The sender algorithms are templates, but `std::execution::task` and `sync_wait` produce concrete instantiations in the standard library whose signatures become part of the implementation's binary interface.

---

13. Conclusion

P0913R1[1] was adopted into C++20 to solve a specific problem: stack overflow in coroutine chains. The solution is `coroutine_handle<>` -returning `await_suspend` . Every major coroutine library adopted it.

Under the current protocol, the sender model's architectural choice - composing operations through non-coroutine sender algorithms with void-returning completions - prevents this mechanism from operating. P3552R3[2]'s `std::execution::task` inherits this limitation. The proposed task-to-task fix does not reach the general case. No launch mechanism avoids the sender composition layer. The trampoline scheduler mitigation reintroduces the runtime cost that symmetric transfer was designed to eliminate.

A protocol-level fix exists. Section 10 describes a mechanism where completion functions and `start()` return `coroutine_handle<>` instead of `void` . Struct receivers propagate handles from downstream without becoming coroutines. Zero-allocation composition is preserved. The fix requires changing the return type of four conceptlevel expressions, the internal receivers and operation states of twenty-five sender algorithms, the coroutine integration bridges, and every third-party type that models `receiver` or `operation_state`. Once `std::execution` ships with void-returning completions, the change becomes ABI-breaking.

Three directions could be explored if the fix is not adopted: domain customization that short-circuits the receiver abstraction for known coroutine types, a language feature for guaranteed tail calls, or coroutine task types that use the awaitable protocol directly for coroutine-to-coroutine composition and reserve the sender model for the boundaries where it is needed.

Users and WG21 should be aware of this cost when evaluating coroutine integration with `std::execution`.

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 like to thank Gor Nishanov for the design of symmetric transfer, Lewis Baker for demonstrating it in cppcoro, and Dietmar Kühl and Jonathan Müller for documenting the limitation in P3796R1 and P3801R0.

References

WG21 Papers

1. P0913R1 - “Add symmetric coroutine control transfer” (Gor Nishanov, 2018). https://wg21.link/p0913r1
2. P3552R3 - “Add a Coroutine Task Type” (Dietmar Kühl, Maikel Nadolski, 2025). https://wg21.link/p3552r3
3. P4003R0 - “Coroutines for I/O” (Vinnie Falco, Steve Gerbino, Mungo Gill, 2026). https://wg21.link/p4003r0
4. P4007R0 - “Senders and Coroutines” (Vinnie Falco, Mungo Gill, 2026). https://wg21.link/p4007r0
5. P2300R10 - “std::execution” (Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach, 2024). https://wg21.link/p2300r10
6. P3796R1 - “Coroutine Task Issues” (Dietmar Kühl, 2025). https://wg21.link/p3796r1
7. P3801R0 - “Concerns about the design of std::execution::task” (Jonathan Müller, 2025). https://wg21.link/p3801r0

8. P3149R11 - “async_scope” (Ian Petersen, Jessica Wong, Kirk Shoop, et al., 2025). https://wg21.link/p3149r11

Libraries

9. cppcoro - A library of C++ coroutine abstractions (Lewis Baker). https://github.com/lewissbaker/cppcoro
10. folly::coro - Facebook’s coroutine library. https://github.com/facebook/folly/tree/main/folly/coro
11. Boost.Cobalt - Coroutine task types for Boost (Klemens Morgenstern). https://github.com/boostorg/cobalt
12. libcoro - C++20 coroutine library (Josh Baldwin). https://github.com/jbaldwin/libcoro
13. Boost.Capy - Coroutines for I/O (Vinnie Falco). https://github.com/cppalliance/capy
14. asyncpp - Async coroutine library (Péter Kardos). https://github.com/petiaccja/asyncpp
15. stdexec - NVIDIA’s reference implementation of std::execution. https://github.com/NVIDIA/stdexec

Other

16. C++ Working Draft - (Richard Smith, ed.). https://eel.is/c++draft/