WG21 PROPOSAL

Does `std::execution` Need More Time?

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

Table of Contents

Revision History

- R0 - Initial revision.

Abstract

`std::execution` brings valuable ideas to C++ - structured concurrency, composable task graphs, and a clean separation of work description from execution policy. However, a fundamental allocator sequencing gap prevents coroutine-based I/O from using stateful allocators: `operator new` runs before the receiver exists, so the allocator arrives too late. Because the gap arises from the interaction between coroutine evaluation order and the sender/receiver protocol, closing it may require changes to the API. Unless a backward-compatible solution addressing this gap can be demonstrated, we recommend deferring `std::execution` to C++29 - and perhaps separately publishing it as an evolvable white paper - so that the allocator story can be resolved before the ABI is frozen.

1. Introduction

P2300R10 ("std::execution") represents years of careful work and genuine progress in structured asynchronous programming for C++:

- **Structured concurrency** with well-defined lifetime guarantees for asynchronous operations
- **Composable sender/receiver pipelines** that separate work description from execution policy
- **Schedulers and execution contexts** that give callers control over where work runs
- **`run_loop`** and foundational primitives that enable deterministic testing of async code
- **A formal model** for reasoning about asynchronous completion signatures

These are real achievements, and the community of users and implementers who have invested in sender/receiver architectures - NVIDIA CCCL, stdexec, libunifex, Folly - has demonstrated substantial value in CPU-bound parallelism and GPU dispatch.

The concern this paper raises is narrower than a general objection: `std::execution` works well for its primary use cases. The gap arises when a sender pipeline must select a custom allocation strategy. The allocator must be available ***before*** the coroutine frame is constructed, but the sender/receiver model provides the allocator only after the frame has already been created.

Coroutine-based I/O causes this gap to become acute. Asynchronous execution makes optimising away the coroutine frame allocation unlikely or impossible, and the impact of multiple extra allocations per I/O operation can be disastrous. The committee has explicitly placed this use case in scope (see Section 6.4); the standard must accommodate it.

The question is whether the gap can be closed without either (1) a breaking API change or (2) leaving the current interface as vestigial cruft. Section 4 examines each proposed escape route; as of this writing, none appears to close the gap.

The allocator sequencing gap is architectural, not a missing feature that can be added later: `operator new` runs before `connect()`, and no library-level change known to the authors can alter that sequencing without destabilising the API. Discovering this gap before standardisation is fortunate, not a failure. The feature's size and complexity make thorough review genuinely difficult, and it is not surprising that a gap visible only in the I/O use case was not identified earlier. There is still time to address it.

---

2. The I/O Use Case

`std::execution` aspires to be a general-purpose asynchronous framework, including for networked I/O. Here is what that use case requires.

A typical networking application launches coroutines that perform I/O:

```
// User's coroutine - handles one client connection
task<void> handle_client(tcp_socket socket)
{
    char buf[1024];
    for (;;)
    {
        auto [ec, n] = co_await socket.async_read(buf);
        if (n > 0)
            process(buf, n);
        if (ec)
            break;
    }
}

// Application launches the coroutine
int main()
{
    io_context ctx;
    tcp_socket sock = accept_connection(ctx);
    auto alloc = ctx.get_frame_allocator();

    run_async(ctx.get_executor(), alloc)(
        handle_client(std::move(sock))   // Frame allocated HERE
    );

    ctx.run();
}
```

The critical sequencing:

```
run_async(executor, alloc)(handle_client(sock))
                                 |
                                 +-> promise_type::operator new runs HERE
                                     Allocator must already be known
```

When `handle_client(sock)` is evaluated, the compiler calls `promise_type::operator new` to allocate the coroutine frame. The allocator must be available at this moment - before the coroutine body executes, before any `co_await`, before any connection to a receiver.

---

3. The Problem

The allocator sequencing gap has three facets: the cost of uncontrolled allocation, the absence of any interception mechanism besides `operator new`, and the ordering constraint that places allocation before the receiver exists.

3.1 The Cost of Uncontrolled Allocation

High-performance I/O servers cannot tolerate uncontrolled heap allocation. Realistic server code calls nested coroutines for parsing, processing, and serialisation - each `task<T>` call allocates a frame:

```
task<void> handle_connection(tcp_socket socket)
{
    char buf[1024];
    while (socket.is_open()) {
        auto n = co_await socket.async_read(buf);  // I/O awaitable: no frame
        co_await process_request(socket, buf, n);    // task<void>: frame allocated
    }
}
// handle_connection itself: 1 frame per connection
// process_request (and coroutines it calls): frame(s) per request
// 10,000 connections x 1,000 requests/sec = 10 million+ frame allocations/sec
// global heap contention becomes the bottleneck
```

The solution is **stateful allocators**:

- recycling allocators that cache recently-freed frames
- arena allocators tied to connection scope
- pool allocators optimised for common frame sizes
- `pmr::memory_resource` implementations that customise the upstream allocation strategy

Modern general-purpose allocators (jemalloc, mimalloc, tcmalloc) achieve impressive throughput via per-thread caching, but they solve a different problem. They reduce contention on the global heap; they do not address the allocation patterns that I/O servers produce. Coroutine frames are short-lived, uniformly sized, and frequently allocated on one thread but freed on another - a cross-thread pattern that defeats per-thread caches.

More fundamentally, a general-purpose allocator has no notion of application-level context. Different coroutine chains might require different policies. A multi-tenant server may enforce per-tenant memory limits, with each tenant's connections using an allocator bound to that tenant's quota. A connection-scoped arena can reclaim all frames in one operation when the connection closes. These policies require the stateful allocators listed above - and that state must be accessible when `operator new` executes.

The performance difference is measurable. P4003R0 ("IoAwaitables: A Coroutines-First Execution Model") benchmarks a 4-deep coroutine call chain (2 million iterations) with and without a recycling frame allocator:

Allocator	Time (ms)	Speedup
`std::allocator`	4251.46	-
Recycling allocator	2398.20	77.3%

The recycling allocator exploits the uniform, short-lived allocation pattern that coroutine frames produce - precisely the pattern that general-purpose allocators cannot optimise for. With proper allocator support, coroutine-based networking matches or exceeds callback-based frameworks in throughput (see Section 6.4). Stateful allocators are not a luxury; for coroutine-based I/O they are a prerequisite for competitive performance.

3.2 HALO Cannot Help

HALO (Heap Allocation eLision Optimisation) allows compilers to elide coroutine frame allocation when the frame's lifetime is provably bounded by its caller. For I/O coroutines launched onto an execution context, the frame outlives its caller - the caller returns immediately after launching the operation, and the frame persists until the OS signals completion. The compiler cannot prove bounded lifetime when the lifetime depends on an external event; hence, HALO cannot apply in this common case, and allocation is mandatory. See Appendix A.1 for a deeper analysis and code example.

It is also worth noting that HALO does nothing when the coroutine's implementation is not visible.

3.3 `operator new` Is the *Only* Interception Point

The C++ language provides exactly one mechanism to intercept coroutine frame allocation: `promise_type::operator new`. This is a property of coroutines themselves, not of sender/receiver:

```cpp
struct promise_type
{
    static void* operator new(std::size_t size)
    {
        // the ONLY place to intercept allocation
        // the allocator must be known RIGHT NOW

        std::pmr::memory_resource* mr = /* ??? */;
        return mr->allocate(size, alignof(std::max_align_t));
    }
};
```

The allocator - along with its state - must be discoverable at this exact moment. No mechanism that provides the allocator later can help. The frame is already allocated by the time any later mechanism executes.

The sender/receiver model inherits this constraint but amplifies it: the protocol's natural source of contextual information - the receiver's environment - is available only after `connect()`, which runs after `operator new`. The result is that the protocol's own allocator delivery mechanism cannot serve coroutine frame allocation.

3.4 What P3552 Solves - and What It Does Not

P3552R3 ("Add a Coroutine Task Type") introduces a two-tier model for execution resources in sender/receiver:

- Creation-time concerns - the allocator - are passed at the call site via `std::allocator_arg_t`.
- Connection-time concerns - the scheduler and stop token - flow through the receiver environment.

This design solves initial allocation cleanly: the caller passes the allocator explicitly, and the coroutine's `operator new` can use it. P3552 is a valuable contribution, and the initial-allocation problem is closed.

Automatic propagation remains unsolved. When a coroutine calls a child coroutine, the child's `operator new` fires during the function-call expression - before the parent's promise has any opportunity to intervene. The receiver's allocator, delivered through the S/R protocol at `connect()`, is structurally unreachable at the point of child frame allocation. The only workaround is manual forwarding: every coroutine in the chain must accept the allocator - whether through a defaulted parameter, an overload, or a variadic template - query the current allocator from its environment, and pass it to every child call. See Appendix A.2 for the complete propagation example.

Manual forwarding works, but three properties distinguish the problem from a minor inconvenience:

1. **Composability loss.** Generic sender algorithms launch child operations without knowledge of the caller's allocator. Consider a pipeline that uses `let_value` to invoke a coroutine:

```
auto pipeline =
    just(std::move(socket))
  | let_value([](tcp_socket& s) -> task<void> {
        // operator new runs HERE, inside the lambda.
        // let_value has no allocator to forward -
        // it does not know the caller uses one.
        co_await s.async_read(buf);
    });
```

The `let_value` algorithm is generic: it invokes the callable and connects the result. It has no mechanism to forward an `allocator_arg` into the callable's return expression, because the algorithm's implementation does not participate in the forwarding chain. The same limitation applies to `when_all`:

```
auto work =
    when_all(
        do_read(socket),    // each returns task<T>
        check_timeout(id)   // operator new runs in each
    );
// when_all launches both tasks.
// Neither receives the caller's allocator.
```

Manual forwarding cannot cross algorithm boundaries.

2. **Silent failure.** Omitting `allocator_arg` from one call does not produce a compile error - the child silently falls back to the default heap. In a high-throughput server, the resulting allocation spike may surface only under production load.

3. **Protocol asymmetry.** Schedulers and stop tokens propagate automatically through the receiver environment. Allocators are the only execution resource that the protocol forces users to propagate by hand. Standardising this asymmetry means accepting that allocators are permanently second-class in the framework's own resource model.

The standard library partially addresses a similar problem for containers through ***uses_allocator*** construction and `scoped_allocator_adaptor` - though custom aggregates must still propagate allocators manually. No equivalent exists for coroutine frames even in the container-like case, and the evaluation-order constraint - `operator new` runs before `co_await` processing - means that adding one would require changes to the protocol.

3.5 The Receiver Arrives Too Late

Context in the sender/receiver model in `std::execution` flows backward from receiver to sender. This creates an allocator sequencing gap for coroutine frame allocation:

```
task<int> my_coro();                         // Coroutine returning a sender

auto sndr = my_coro();                       // Step 1: operator new runs HERE
                                             //         Frame already allocated

auto op = connect(std::move(sndr), rcvr);   // Step 2: Receiver connected HERE
                                             //         get_allocator(get_env(rcvr))
                                             //         available NOW – too late

start(op);                                   // Step 3: Operation starts
```

P2300R4 ("std::execution," 2022-01-18) acknowledges this sequencing:

> *"In the sender/receiver model... contextual information is associated with the receiver and is queried by the sender and/or operation state **after** the sender and the receiver are `connect` -ed."*

P3826R3 (2026-01-05) confirms the consequence:

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

The allocator is part of this contextual information. It becomes available only after `connect()` - but the coroutine frame was allocated in Step 1, before `connect()` was called.

For coroutine-based I/O, the allocator arrives too late. This lack of customisability is not a flaw in the sender/receiver formalism itself; rather, it is an unanticipated gap that emerges only when that formalism meets the concrete requirements of coroutine-frame allocation.

---

4. Can It Be Fixed Later?

How we proceed will depend on our ability to address the issue in a way that does not require either (1) an API/ABI break or (2) leaving what we have as a vestigial remnant of an incomplete framework.

Each potential escape route of which the authors are currently aware - `await_transform`, the two-tier design, P3826R3's algorithm dispatch fixes - leaves the fundamental sequencing constraint intact.

4.1 The Gap Is Structural

`operator new` runs before `connect()`. No library-level change known to the authors can alter this sequencing - it is determined by the C++ language specification for coroutines' interacting with the sender/receiver protocol. There is no obvious post-hoc extension point between "coroutine frame is allocated" and "receiver is connected." Closing the gap appears to require changing the sequencing - which means ***changing*** the API.

4.2 `await_transform` Cannot Help

A natural question is whether `promise_type::await_transform` could inject the allocator into child coroutine calls, eliminating explicit `allocator_arg` passing. It cannot. In `co_await child_coro(args...)`, the function call `child_coro(args...)` is evaluated first - the child's `operator new` fires and the frame is allocated before `co_await` processing begins. By the time `await_transform` sees the returned `task<T>`, the child's frame has already been allocated without the parent's allocator. See Appendix B.2 for a deeper evaluation-order analysis.

4.3 The Receiver's Allocator Cannot Reach Coroutine Frame Allocation

P3552R3 establishes a two-tier model: the allocator is a creation-time concern passed at the call site, while the scheduler and stop token are connection-time concerns from the receiver. The promise's `get_env()` exposes the stored allocator to child operations via `get_allocator`. That mechanism gives callers a standard way to allocate the initial coroutine frame with a chosen allocator - a valuable contribution to the design.

Propagating that choice through a chain of nested coroutine calls, however, remains the harder and (as yet) ***unsolved*** problem.

In `co_await child_coro(args...)`, the child's `operator new` fires before `co_await` processing begins - before the parent's promise has any opportunity to intervene. The receiver's allocator, delivered through the S/R protocol at `connect()`, is structurally unreachable at the point of child frame allocation. A caller can work around this by passing the allocator manually via `std::allocator_arg`, but that bypasses the S/R protocol entirely: the allocator no longer comes from the receiver's environment, and every intermediate coroutine in the chain must accept, query, and forward it explicitly - a manual forwarding burden that grows with call depth.

With standard containers, the allocator type is part of the type signature (`vector<T, Alloc>`), and ***uses_allocator*** construction gives generic code a standard way to propagate the allocator through nested construction. With coroutines, the allocator is consumed inside `operator new` (which runs before the receiver exists) and the coroutine's return type (`task<T>`) does not carry the allocator type. There is no ***uses_allocator*** equivalent for coroutine frame allocation. Nor is it clear that one could be created: any indirection layer that wraps a coroutine invocation risks defeating Heap Allocation eLision Optimisation (HALO), because the compiler must see the coroutine call directly at the call site.

4.4 P3826R3's Solutions Target a Different Problem

P3826R3 ("Fix Sender Algorithm Customization," 2026-01-05) proposes important fixes for sender algorithm customisation - a real problem prompted by NB comment US 207 and resolved over three revisions of the proposal. However, its five proposed solutions all target algorithm dispatch - i.e., which implementation of `then`, `let_value`, or `bulk` should run. Four of the five do not change when the allocator becomes available. The fifth - remove all `std::execution` - resolves the gap by deferral. See Appendix B.1 for a deeper analysis of each solution.

4.5 ABI Lock-In Makes the Architectural Gap Permanent

Once standardised, the relationship between `operator new` and `connect()` becomes part of the ABI. The standard does not break ABI on standardised interfaces; that stability guarantee is central to C++'s value proposition.

A fix would likely need `connect()` to propagate allocator context before the coroutine frame is allocated - a structural change to the sender protocol. An API change to a standardised interface implies an ABI change, and an ABI change to a standardised interface is effectively precluded by committee practice. The committee would then face a choice: accept the gap permanently, or introduce a parallel framework that serves coroutine-based I/O alongside the original. Neither outcome is impossible to manage, but both carry long-term costs that are worth weighing before the ABI boundary is set.

---

5. The Committee's Own Record

The committee's own proceedings confirm that the allocator story is unresolved and the design is still actively evolving.

5.1 The Allocator Poll

P3796R1 (Dietmar Kuhl, "Coroutine Task Issues") was reviewed by LEWG in a September 2025 telecon. The allocator question was polled directly:

> *"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 to neutral - an outcome consistent with the structural difficulty described in Section 4.3: without a mechanism to propagate allocator context through nested coroutine calls, the committee had no direction to endorse. The author was directed to explore preferred solutions with code examples.

Two US national body comments confirm the same concern. US 255 (LWG4335) requests that `task` use the allocator from the receiver's environment rather than the one supplied at coroutine creation. The NB comment itself acknowledges the difficulty: "the used allocator type needs to be known when the coroutine is created. At that time the receiver isn't known, yet." The core issue is that coroutine `operator new` overload resolution requires the allocator ***type*** at compile time - a runtime pointer to a `pmr::memory_resource` is not sufficient. At Kona (2025-11-06) LEWG polled the direction:

> *"We approve the direction proposed in the NB comment: US 255-384 [task.promise] Use allocator from receiver's environment LWG4335 and ask for wording."*
>
> | *SF* | *F* | *N* | *A* | *SA* |
> |---|---|---|---|---|
> | *4* | *7* | *0* | *1* | *0* |
>
> *Attendance: 17. Outcome: consensus in favour.*

The sole A vote - from the task paper's author - noted that using the receiver's allocator "goes against the general behaviour of allocators." Wording remains outstanding. US 253 (LWG4333) requests arbitrary allocator support for the coroutine frame; wording is also still required.

D3980R0 (Dietmar Kuhl, "Task's Allocator Use," 2026-01-25) addresses NB comments US 253, US 254, US 255, and US 261. Notably, D3980R0 changes the allocator propagation model relative to P3552R3, which was adopted at Sofia in June 2025. The fact that the task author's own design for allocator handling has changed between the adopted text and the NB comment resolution underscores that the allocator story for coroutine task types remains under active development.

LWG 4356 (`connect()` should use `get_allocator(get_env(rcvr))`) confirms the gap has been filed as a specification defect - not merely an external complaint, but a committee-level acknowledgement of the sequencing problem.

5.2 Post-Approval Changes

The volume of papers written to fix, rework, or complete `std::execution` after its approval for C++26 is substantial, and reflects the feature's scope and ambition.

Period	Removals	Reworks	Wording	Missing Functionality	LWG	Total
Pre-Wroclaw (Mar-Oct 2024)	1	5	-	1	-	7

Period	Removals	Reworks	Wording	Missing Functionality	LWG	Total
Pre-Hagenberg (Nov 2024-Feb 25)	1	-	2	2	3	8
Pre-Sofia (Mar-Jun 2025)	-	2	-	7	1	10
Pre-Kona (Jul-Nov 2025)	-	3	3	1	7	14
Pre-London (Dec 2025-Feb 2026)	-	2	1	-	-	3
Total	2	12	6	11	11	42

See Appendix D for the complete listing with dates, authors, and current status.

Key observations:

- **The churn is accelerating, not slowing.** The pre-Kona period produced 14 items - the highest volume - including Priority 1 safety defects. Two more papers appeared in the January 2026 mailing, with the London meeting still months away.
- **The severity has not decreased.** The pre-Kona period includes Priority 1 LWG defects: a dangling-reference vulnerability in `transform_sender` (LWG 4368) and an unconstrained alias in `connect_result_t` (LWG 4206). Two outstanding NB comments on allocator support remain without wording.
- **Design reworks span the entire timeline since `std::execution` was voted into the C++26 working draft** (Tokyo, March 2024). From P2855R1 (replacing `tag_invoke`, March 2024) through P3927R0 (fixing `task_scheduler` bulk execution, January 2026) - 11 rework papers over 22 months.

In total: 31 papers, 11 LWG defects, and 2 NB comments - 44 items modifying a single feature after its approval.

5.3 The C++23 Precedent

This is not the first time maturity concerns have led to deferral. P2300 failed to achieve consensus for C++23:

> *LEWG 2022-01-12: "Produce a P2300R4, modified as described below, and then send the revised paper to LWG for C++23 with priority 1"*
>
> *SF:15 / F:9 / N:1 / A:3 / SA:6 (Attendance: 39)*

Recorded SA reasons included "maturity/readiness" and "Feels like ranges all over again. Needs maturity." Some of the same categories of open work - algorithm reworks, allocator sequencing unresolved, LWG issues opened during review - are visible in the current cycle. The difference is that the concerns which were speculative in 2022 are now documented in specific LWG issues, committee polls, and unresolved NB comments.

6. A Path Forward

This paper asks the committee to consider whether the allocator sequencing gap should be resolved before the `std::execution` API is frozen in the IS.

6.1 Avoiding API Lock-in

The P2300 authors are best placed to propose a resolution to the allocator sequencing gap. We welcome any solution that closes it - whether by changes to the `task` type, to the coroutine promise protocol, or to the interaction between `operator new` and `connect`. Our concern is that freezing the API in the IS before the gap is addressed would lock C++ into an interface that cannot serve coroutine-based I/O without workarounds.

If no solution is forthcoming in the C++26 time frame, deferral to a later revision of the standard would avoid ABI lock-in while preserving the committee's freedom to fix what remains.

6.2 What Deferral Means

Deferring `std::execution` from C++26 does not discard the work done by the P2300 authors and implementers. Production-quality implementations already exist and are in active use. Deferral means the committee does not freeze the ABI until the remaining architectural gaps - allocator sequencing, allocator propagation, and the interaction with coroutine task types - are resolved.

Deferral is not failure. The committee deferred trivial relocation to C++29 because open design questions remained; the expectation is that deferral will yield a stronger proposal, built on the foundations already laid. The same precedent applies here: the P2300 authors' investment is preserved in full, and production-quality implementations (stdexec, NVIDIA CCCL, libunifex, Folly) remain available to users regardless of the standardisation timeline.

6.3 Trade-offs of the Current Timeline

The urgency question deserves an honest answer: what are the costs of deferral, and what are the costs of standardising before the allocator story is complete?

Organisations building on `std::execution` today are not blocked by the absence of an IS specification - they build against concrete library implementations. Unlike language features, non-standard library implementations are straightforward to adopt; organisations can use Boost, stdexec, or other implementations without compiler extensions. If the committee were to defer this feature until a future iteration of the language, no current users would be affected.

The networking use case - which arguably represents the largest constituency for asynchronous C++ - is the community most affected by shipping `std::execution` that has an incomplete allocator story. SG4 mandated that networking be built exclusively on sender/receiver (see Section 6.4). If the current sender/receiver framework ships without a workable allocator propagation model for coroutine-based I/O, the networking community faces a choice between inefficient workarounds and non-standard alternatives.

This framing is not an accusation that the committee is neglecting networking; rather, it is an observation that the current timeline favours constituencies that do not need allocator propagation (GPU dispatch, CPU-bound parallelism) over the constituency that does (I/O-bound networking). Deferral preserves optionality for everyone; standardising before the allocator story is complete would impose a concrete cost on the use case with the broadest demonstrated demand.

6.4 I/O Is in Scope

A likely response is that `std::execution` is for structured concurrency and parallelism, and networking is a separate paper track. The committee's own record contradicts this assertion.

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:5 / F:5 / N:1 / A:0 / SA:1 - Consensus.*

Networking must be built on `std::execution`'s sender/receiver model. The task type (P3552R3) is the primary I/O usability layer - it exists so that users can write `co_await socket.async_read(buf)`. If the coroutine integration cannot access the allocator, the I/O use case is effectively unserved. See Appendix C.1-C.3 for the complete committee record on this question.

Coroutine-based networking is not merely a theoretical possibility - it is already competitive with established callback-based frameworks. P4003R0 ("IoAwaitables: A Coroutines-First Execution Model") reports throughput benchmarks comparing a coroutine-based networking library (with frame recycling allocators) against Asio's callback model:

Threads	Coroutine (Kops/s)	Asio (Kops/s)	Ratio
1	229.68	227.80	1.01x
2	348.89	336.79	1.04x
4	493.08	479.70	1.03x
8	588.18	497.07	1.18x
16	625.25	460.21	1.36x

At low thread counts the two approaches match; at higher concurrency the coroutine-based approach pulls ahead - 18% higher throughput at 8 threads and 36% at 16. Asio's callback model degrades from 8 to 16 threads (497 to 460 Kops/s) while the coroutine-based framework continues to scale (588 to 625 Kops/s). The coroutine approach exhibits better contention behaviour under load.

The same paper reports a 77% throughput improvement when using a recycling frame allocator instead of `std::allocator` on a 4-deep coroutine call chain (2 million iterations). Without allocator support, those gains disappear.

These results demonstrate that coroutine-based networking with proper allocator support is production-viable today. The allocator sequencing gap is what stands between `std::execution` and this level of performance.

---

7. Conclusion

`std::execution` represents important progress in structured asynchronous programming for C++. Its contributions - structured concurrency, composable pipelines, and a formal model for async completion - are valuable and worth building upon.

For coroutine-based asynchronous I/O, however, a fundamental allocator sequencing gap remains:

1. **Allocation is mandatory** - HALO cannot optimise away frames that outlive their callers
2. **Stateful allocators are required** - 10 million+ allocations/sec demands recycling, pooling, or arena strategies
3. **`operator new` is the only interception point** - the allocator must be known when the frame is allocated
4. **`std::execution` provides the allocator too late** - the receiver's environment is available only after `connect()`
5. **The gap is difficult to fix later** - `operator new` runs before `connect()`, and no library change known to the authors can alter that sequencing without destabilising the API

P2300R4 established the sender/receiver context model in January 2022. Four years later, the design continues to evolve: P3826R3 proposes significant architectural changes to `transform_sender`, removes early customization entirely, and restructures the relationship between `continues_on` and `schedule_from`. This level of active change reflects a design that is still evolving and suggests there may be value in allowing that evolution to complete before freezing the ABI.

If `std::execution` ships with C++26 but cannot serve coroutine-based I/O without workarounds, the committee risks standardising a framework that works for CPU-bound parallelism and GPU dispatch but leaves the largest async constituency - networking - reliant on non-standard alternatives.

The goal is not to block `std::execution`. It is to ensure that when it ships, it will be capable of serving the full spectrum of its intended use cases - including coroutine-based I/O - so that the C++ community gets one async framework, not two.

We ask the committee to address the allocator sequencing gap before freezing the `std::execution` API in the IS. We welcome a solution from the P2300 authors and are eager to collaborate on one. If the gap cannot be closed in the C++26 time frame, we encourage the committee to continue to evolve this important feature towards its inclusion in C++29.

Appendix A - Code Examples

A.1 Why HALO Cannot Help

HALO (Heap Allocation eLision Optimisation) 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:

```
task<size_t> read_data(socket& s, buffer& buf)
{
    co_return co_await s.async_read(buf);  // Suspends for network I/O
}

void start_read(executor ex)
{
    start_detached(ex, read_data(sock, buf)); // Frame must outlive this function
}                                             // Caller returns immediately
```

`start_detached` transfers the frame's lifetime to the execution context. The caller returns, and the frame persists until the OS signals I/O completion - which may take microseconds or seconds. The compiler cannot prove bounded lifetime, so HALO cannot apply and allocation is mandatory.

A note of precision: HALO cannot apply when the coroutine frame outlives its caller, which is the common case for I/O coroutines launched onto an execution context. In the nested case - where coroutine A `co_await`s coroutine B - A's frame remains alive while B executes, and B's frame lifetime is bounded by A's. In principle, a sufficiently advanced compiler could elide B's allocation into A's frame. No current compiler does this for `task<T>` coroutines, and the optimisation becomes infeasible once any frame in the chain is transferred to an execution context. HALO cannot help when frames outlive their callers - which is the case that matters for I/O.

A.2 The `allocator_arg` Propagation Chain

P3552R3 provides a mechanism for passing an allocator to a single coroutine via `std::allocator_arg_t`. Here is what propagation through a call chain looks like in practice:

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

// Defaulted parameter approach:
task<void> level_three(
    std::allocator_arg_t = {},
    auto alloc = std::allocator<>{})
{
    co_return;
}

// Defaulted parameter approach:
task<void> level_two(
    int x,
    std::allocator_arg_t = {},
    auto alloc = std::allocator<>{})
{
    co_await level_three(std::allocator_arg, alloc);
}

// Variadic template approach (also works):
template<class... Args>
task<int> level_one(int v, Args&&... args)
{
    auto alloc =
        co_await ex::read_env(ex::get_allocator);
    co_await level_two(42, std::allocator_arg, alloc);
    co_return v;
}
```

Every coroutine in the chain must accept the allocator argument - via defaulted parameters, overloads, or variadic templates - and forward it to each child. Function signatures no longer reflect algorithmic intent. The allocator becomes viral: forgetting to forward it at any point silently breaks the chain without a compile error.

A.3 The `connect` / `start` / `operator new` Sequence

The complete evaluation sequence showing the gap between frame allocation and receiver connection:

```
auto sndr = my_coro();                     // Step 1: operator new runs HERE
                                           //         Frame already allocated

auto op = connect(std::move(sndr), rcvr); // Step 2: Receiver connected HERE
                                           //         get_allocator(get_env(rcvr))
                                           //         available NOW - too late

start(op);                                 // Step 3: Operation starts
```

P2300R4 (2022-01-18) acknowledges this sequencing:

> *"In the sender/receiver model... contextual information is associated with the receiver and is queried by the sender and/or operation state **after** the sender and the receiver are `connect` -ed."*

P3826R3 (2026-01-05) confirms the consequence for customisation:

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

Note: the P3826R3 quote refers specifically to algorithm dispatch customisation, not to allocator sequencing. But the underlying cause is the same: the receiver - and its environment including the allocator - is not available until `connect()` .

A.4 Sender/Receiver vs Coroutine - Side by Side

The same `handle_connection` logic in both models illustrates where the gap opens.

Pure sender/receiver (no coroutines):

```
auto handle_connection(tcp_socket socket)
{
    return just(std::move(socket))
      | let_value([](tcp_socket& s) {
            return async_read(s)
              | let_value([&](auto data) {
                    return process(s, data);
                });
        });
}

// Usage:
auto op = connect(handle_connection(sock), rcvr);
//                                          ^^^^
// The receiver's allocator is available here.
// connect() can allocate the entire operation state
// tree using get_allocator(get_env(rcvr)).
start(op);
```

The standard sender algorithms (`just` , `let_value` , etc.) build a lazy description; no operation state is allocated until `connect(sender, receiver)` , at which point the receiver's environment - including the allocator - is available for the entire operation state tree. A custom sender theoretically ***could*** allocate earlier, but the protocol's design defers allocation to `connect()` precisely so that the receiver's allocator can be used. There is no allocator sequencing gap.

Coroutine model:

```
task<void> handle_connection(tcp_socket socket)
{
    // operator new has ALREADY run by this point.
    // The frame is allocated. The receiver does not exist yet.
    char buf[1024];
    auto n = co_await socket.async_read(buf);
    co_await process_request(socket, buf, n);
}
```

The coroutine frame is allocated during the function call - before any receiver exists. This is the precise moment the gap opens. Any sender that allocates storage at creation time faces the same allocator sequencing gap, but coroutines make it pervasive: `task<T>` always allocates a frame in `operator new` before any receiver exists. Since `task<T>` is the primary usability layer for writing sender/receiver code, the gap affects the most common authoring path.

A.5 The Full Ceremony for Allocator-Aware Coroutines

The current sender/receiver model requires five layers of machinery to propagate a custom allocator through a coroutine call chain. The following example uses the P3552 mechanism with `write_env` to inject the allocator at the launch site:

```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
my_task level_two(
    int x,
    std::allocator_arg_t = {},
    auto alloc = std::allocator<>{})
{
    co_return;
}

my_task level_one(
    int v,
    std::allocator_arg_t = {},
    auto alloc = std::allocator<>{})
{
    // 4. Query the allocator from the environment
    auto a =
        co_await ex::read_env(ex::get_allocator);
    // 5. Forward it to every child call
    co_await level_two(42, std::allocator_arg, a);
    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());
}
```

The five requirements are:

1. A custom environment struct with the allocator type and a `get_allocator` query
2. A task type alias parameterised on the allocator type
3. `std::allocator_arg_t` + allocator parameter on every coroutine in the chain
4. `read_env(get_allocator)` to query the current allocator at each call site
5. `write_env` at the launch site to inject the allocator into the sender's environment

Forgetting any one of steps 3-5 silently falls back to the default allocator. The compiler provides no diagnostic.

By contrast, P4003R0 demonstrates a model where the allocator propagates automatically through the coroutine call chain. The allocator is set once at the launch site; every child coroutine's `operator new` reads it from thread-local storage without any parameter forwarding:

```
// P4003 model: allocator propagates automatically
task<int> level_two(int x)
{
    co_return x;   // inherits caller's allocator
}

task<int> level_one(int v)
{
    auto r = co_await level_two(42);  // no forwarding
    co_return v + r;
}

void launch(auto& ex)
{
    recycling_allocator<> alloc;
    // allocator supplied once; propagates to all children
    run_async(ex, stop_token{}, alloc,
        [](int) {},
        [](std::exception_ptr) {})(level_one(0));
}
```

The function signatures reflect algorithmic intent. The allocator is supplied once at the launch site and propagates to every child in the chain without user intervention. Forgetting nothing is possible because there is nothing to forget.

Appendix B - Counterarguments Examined

B.1 P3826R3 and Algorithm Dispatch

P3826R3 proposes important fixes for sender algorithm customisation, prompted by NB comment US 207. The resolution went through three revisions before acceptance in January 2026 - evidence of ongoing design evolution, but not itself an indictment.

P3826 targets algorithm dispatch and was never intended to address allocator sequencing. Because its Solution 4.5 restructures `transform_sender` to accept the receiver's environment, however, a natural question is whether that change also closes the allocator sequencing gap. It does not - but examining why is instructive.

P3826 offers five solutions. All target algorithm dispatch - i.e., which implementation of `then` , `let_value` , or `bulk` should run:

Solution 4.1: Remove all `std::execution`. This would resolve the allocator sequencing gap by deferral.

Solution 4.2: Remove customisable sender algorithms. This removes `then` , `let_value` , `bulk` , etc. It does not change when the allocator becomes available.

Solution 4.3: Remove sender algorithm customisation. This removes the ability to customise algorithms. It does not change when the allocator becomes available.

Solution 4.4: Ship as-is, fix via DR. This defers the fix. It does not change when the allocator becomes available.

Solution 4.5: Fix algorithm customisation now. P3826's recommended fix restructures `transform_sender` to take the receiver's environment, changing information flow at `connect()` time:

```
get_completion_domain<set_value_t>(get_env(sndr), get_env(rcvr))
```

This tells senders where they will start, enabling correct algorithm dispatch. It does not change when the allocator becomes available - the receiver's environment is still only queryable after `connect()` .

There was an assertion made during the Kona discussion that removing sender algorithm customisation could preclude shipping the task type entirely, since the task type's utility depends on the ability to customise algorithms. This would imply that the allocator sequencing issue, the algorithm customisation issue, and the task type are architecturally intertwined.

B.2 Why `await_transform` Cannot Close the Gap

A reader familiar with C++ coroutine machinery may ask whether `promise_type::await_transform` could inject the allocator into child coroutine calls, eliminating explicit `allocator_arg` passing.

The issue is evaluation order. In the expression:

```
co_await child_coro(args...)
```

The function call `child_coro(args...)` is evaluated ***first***. The child's `operator new` fires, the coroutine frame is allocated, the promise is constructed, and `get_return_object()` runs - all during the function call expression. Only then does `co_await` processing begin, at which point `await_transform` sees the returned `task<T>`.

By the time `await_transform` executes, the child's frame has already been allocated without the parent's allocator. There is no customisation point in the C++ coroutine specification that allows a caller to inject context into a callee's allocation. `operator new`, `get_return_object`, and `initial_suspend` all execute during the function call expression, before `co_await` processing begins.

`await_transform` ***can*** inject context into a child's ***execution*** phase - this is how P3552's `task` propagates the scheduler and stop token via the receiver environment at `connect()` / `start()` time. But allocation and execution are different phases: allocation happens at call time, execution happens at start time. The sender/receiver model's backward-flow context is designed for the execution phase. It has no mechanism to influence the allocation phase, because allocation precedes the existence of the receiver.

B.3 The Two-Tier Design and Allocator Propagation

P3552R3 establishes a two-tier model:

- **Creation-time concerns:** The allocator is passed at the call site via `std::allocator_arg_t`. The promise stores it and exposes it through `get_env()`.
- **Connection-time concerns:** The scheduler and stop token come from the receiver's environment, available after `connect()`.

This is a thoughtful design that correctly separates two categories of concern. It solves allocating the ***initial*** coroutine frame with a custom allocator - a real contribution.

The unsolved problem is ***propagation***. When `level_one` calls `level_two` which calls `level_three`, each child coroutine needs the allocator at its call site - before it has any connection to a receiver. The parent must query its own environment, extract the allocator, and pass it explicitly.

A likely counterargument is that this is analogous to how allocators work for standard containers: you pass the allocator at construction time, and the container propagates it internally. But coroutines are fundamentally different:

- With containers, the allocator type is part of the type signature (`vector<T, Alloc>`). With coroutines, the return type is `task<T>` - it does not carry the allocator type.
- With containers, `uses_allocator` construction gives generic code a standard way to propagate the allocator. With coroutines, there is no equivalent mechanism.

- With containers, the allocator is passed once at construction and the container handles propagation internally. With coroutines, every call site in the chain must manually query and forward.

Until a propagation mechanism comparable to `uses_allocator` exists for coroutine frame allocation, the two-tier design solves only the first call in the chain. Production I/O code involves deep call chains, and the manual forwarding burden makes it impractical to maintain allocator discipline throughout.

One possible direction would be to make the promise type a factory for child coroutine calls, allowing it to inject the allocator automatically. This would require either a new coroutine customisation point that fires before the child's `operator new`, or a change to the evaluation order specified in [expr.await] so that frame allocation can be deferred until the parent's promise has had a chance to supply the allocator. Either path is a language-level change to the coroutine specification, and would need time to design and validate.

Appendix C - Committee Record

C.1 The Single Async Model Debate

On 2021-09-28 (attendance: 49), the committee debated whether `std::execution` should be the single async model for C++:

> *"We must have a single async model for the C++ Standard Library"*
>
> | *SF* | *F* | *N* | *A* | *SA* |
> |---|---|---|---|---|
> | *5* | *9* | *10* | *11* | *4* |
>
> *No consensus.*

The committee could not agree on a single model, but the question itself reveals the ambition: `std::execution` was being positioned to serve all async use cases including I/O.

C.2 The C++23 Deferral

LEWG 2022-01-12:

> *"Produce a P2300R4 (std::execution), modified as described below, and then send the revised paper to LWG for C++23 with priority 1"*
>
> | *SF* | *F* | *N* | *A* | *SA* |
> |---|---|---|---|---|
> | *15* | *9* | *1* | *3* | *6* |
>
> *Attendance: 39. Weak consensus.*

Recorded SA reasons included: "maturity/readiness/tag_invoke and the numerous change in the last few meetings," "Feels like ranges all over again. Needs maturity," and "Too few reviews from embedded people."

The C++23 deferral happened because of maturity concerns. Some of those same categories of open work remain visible in the current cycle.

C.3 SG4 Networking Mandate

SG4 polled at Kona (November 2023) on P2762R2:

> *"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.*

SG4's feature requirements for networking (Kona 2023):

Feature	Yes	No
`async_read`	8	2
`async_write`	7	2
`timeout`	8	2
`connect(context, name)`	8	1

These are the I/O primitives that will be built on `std::execution`.

C.4 P3552 Design Approval and Forwarding

The P3552R1 design approval poll (LEWG, 2025-04-22):

> | *SF* | *F* | *N* | *A* | *SA* |
> |---|---|---|---|---|
> | *5* | *6* | *6* | *1* | *0* |
>
> *Attendance: 23.*

The neutral votes matched the favourable votes in number.

The forwarding poll (LEWG, 2025-05-06):

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

C++29 was unanimous; C++26 was conditional and weak.

C.5 Hagenberg Design Reworks

At Hagenberg (February 2025), SG1 forwarded P3552R0 with required architectural changes: replacing `continues_on` with a new `affine_on` mechanism, adding a cancellation channel, supporting errors without exceptions, and implementing scheduler affinity (SF:7 / F:0 / N:0 / A:1 / SA:0). The sole A voter specifically wanted "a more general concept for scheduler affinity to come to SG1 before we ship something in the standard."

LEWG then polled on `affine_on` replacing `continue_on` (SF:9 / F:6 / N:2 / A:1 / SA:0) and on error-without-exceptions support (SF:6 / F:6 / N:4 / A:1 / SA:1). These are fundamental design changes - not editorial fixes - made in the same year as the C++26 deadline.

C.6 P3796R1 and the Allocator Polls

See Section 5.1 for the LEWG allocator poll. The broader context:

P3796R1 (Dietmar Kuhl, "Coroutine Task Issues") was reviewed in LEWG telecons during August-September 2025. Some sections achieved consensus; others remain pending. Multiple LWG issues were opened (4329-4332, 4344).

US 255 (cplusplus/nbballot#959, LWG4335): "Use allocator from receiver's environment." The NB comment states: "Normally, the get_allocator query forwards the allocator from the receiver's environment. For task the get_allocator query used for co_awaited senders uses the allocator passed when creating the coroutine or the default if there was none. It should use the receiver's environment, at least, if the receiver's environment supports a get_allocator query. Supporting the receiver's allocator isn't always possible: the used allocator type needs to be known when the coroutine is created. At that time the receiver isn't known, yet."

The NB comment identifies the same allocator sequencing constraint that this paper examines.

US 253 (cplusplus/nbballot#961, LWG4333): "Allow use of arbitrary allocators for coroutine frame." Status: needs wording.

C.7 Kona Algorithm Customisation Straw Poll

At Kona (November 2025), LEWG reviewed P3826's proposed fix for algorithm customisation:

Option	Description	F	A
1	Remove all of the C++26 `std::execution`	3	30
2	Remove all of the customisable sender algorithms for C++26	12	14
3	Remove sender algorithm customisation (early & late)	17	5
4	Fix customisations now (only late CPs)	21	7
5	Ship as-is and fix algorithm customisation in a DR	6	21

Attendance: 44.

Option 4 (fix now) won the most support. The resolution went through three revisions before eventual acceptance in January 2026.

C.8 Outstanding LWG Defects

Two Priority 1 defects remain open as of the Kona meeting:

- LWG 4206: `connect_result_t` unconstrained, causing hard errors instead of SFINAE-friendly failures.
- LWG 4368: Dangling-reference vulnerability in `transform_sender` (stack-use-after-scope) - returns xvalue to a dead temporary, potential undefined behaviour.

Additional open defects:

- LWG 4190: `completion-signatures-for` specification is recursive - a circular dependency that cannot be satisfied.
- LWG 4215: `run_loop::finish` should be `noexcept` - throwing causes `sync_wait` to hang forever.

- **LWG 4355:** `connect-awaitable()` should mandate receiver completion-signals.
- **LWG 4356:** `connect()` should use `get_allocator(get_env(rcvr))` - directly relevant to the allocator sequencing issue.

Priority 1 issues in an approved feature indicate that the specification is still stabilising. Both affect core mechanisms (`connect` and `transform_sender`) that most sender/receiver programs exercise.

Appendix D - Post-Approval Modification Catalogue

The following tables list all WG21 papers identified as fixing, reworking, removing, or completing missing functionality in `std::execution` (P2300) after its approval for C++26. Papers that extend the framework into new domains (e.g., networking) are excluded.

Removals

Paper	Title	Author(s)	Date	Status	Change
P3187R1	Remove `ensure_started` and `start_detached` from P2300	Lewis Baker, Eric Niebler	2024-10-15	Adopted	Removes two algorithms that dynamically allocate with no allocator customization and break structured concurrency.
P3682R0	Remove `std::execution::split`	Eric Niebler	2025-02-04	Adopted (Sofia)	Removes `split` due to incorrect description of its purpose and problematic semantics.

Major Design Reworks

Paper	Title	Author(s)	Date	Status	Change
P2855R1	Member customization points for Senders and Receivers	Ville Voutilainen	2024-03-18	Adopted	Replaces `tag_invoke` -based ADL customization with member functions - a breaking change.
P2999R3	Sender Algorithm Customization	Eric Niebler	2024-04-16	Adopted	Removes ADL-based customization of sender algorithms in favour of member-function customization on a domain object.

Paper	Title	Author(s)	Date	Status	Change
P3303R1	Fixing Lazy Sender Algorithm Customization	Eric Niebler	2024-10-15	Adopted	Fixes gross oversight in P2999 where wording changes that implement the approved design were missing.
P3175R3	Reconsidering the `std::execution::on` algorithm	Eric Niebler	2024-10-15	Adopted	Renames `on` to `starts_on` and `transfer` to `continues_on` because usage revealed a gap between users' expectations and actual behaviour. Also fixes a bug in `get_scheduler` .
P3557R3	High-Quality Sender Diagnostics with Constexpr Exceptions	Eric Niebler	2025-06-10	Adopted (Sofia)	Reworks `get_completion_signatures` from member function to static constexpr function template. Adds `dependent_sender` concept.
P3570R2	Optional variants in sender/receiver	Lewis Baker	2025-06-14	Adopted - forwarded to LWG for C++26.	Adds `get_await_completion_adapter` for coroutine users.
P3718R0	Fixing Lazy Sender Algorithm Customization, Again	Eric Niebler	2025-07-24	In Progress - open, bumped from 2025-telecon to 2026-telecon milestone. Third paper attempting to fix lazy customization (after P2999, P3303). Linked to NB comment.	Further fixes to the lazy customization mechanism after P3303.

Paper	Title	Author(s)	Date	Status	Change
P3826R3	Fix Sender Algorithm Customization	Eric Niebler	2025-11-14	In Progress - open, 2026-telecon milestone. Title evolved from "Defer... to C++29" (R0) to "Fix or Remove..." (R1) to "Fix..." (R3). Linked to 5 NB comments. Under active LEWG review.	Proposes deferring algorithm customization features that cannot be fixed in time for C++26.
P3927R0	`task_scheduler` Support for Parallel Bulk Execution	Eric Niebler	2026-01-17	In Progress - open, 2026-telecon milestone. January 2026 mailing. Not yet reviewed in telecon. Implemented in NVIDIA CCCL.	Fixes `task_scheduler` not parallelizing bulk work when wrapping a `parallel_scheduler` .

Wording Fixes and Corrections

Paper	Title	Author(s)	Date	Status	Change
P3396R1	`std::execution` wording fixes	Eric Niebler	2024-11-22	Adopted	Omnibus paper addressing multiple wording issues: `run_loop` preconditions, environment ownership, scheduler concept inconsistencies.

Paper	Title	Author(s)	Date	Status	Change
P3388R3	When Do You Know `connect` Doesn't Throw?	Ville Voutilainen	2025-02-14	Adopted	Fixes incorrect `noexcept` clause of the constructor of `basic-state` .
P3914R0	Assorted NB comment resolutions for Kona 2025	Various	2025-11-07	In Progress - omnibus NB comment resolution paper. Sections 2.2-2.5 address `std::execution` . Individual resolutions adopted piecemeal.	Addresses national body comments on the C++26 CD related to `std::execution` .
P3887R1	Make `when_all` a Ronseal Algorithm	Robert Leahy	2025-11-07	Adopted - forwarded at Kona (SF:10/F:5/N:0/A:0/SA:0), wording merged into draft (Dec 2025).	Fixes `when_all` stop-request handling - removes unnecessary stop-detection complexity that made the algorithm's behaviour surprising.
P3940R0	Rename concept tags for C++26: `sender_t` to `sender_tag`	Arthur O'Dwyer, Yi'an Ye	2025-12-15	In Progress - open, 2026-telecon milestone. Post-Kona mailing. Not yet reviewed in telecon.	Renames concept tag types (`sender_t` to `sender_tag` , etc.) for naming consistency - another post-approval naming correction.

Missing Functionality

Paper	Title	Author(s)	Date	Status	Change
P3425R1	Reducing operation-state sizes for subobject child operations	Eric Niebler	2024-11-19	Design approved - LEWG Wroclaw (Nov 2024) strong consensus. C++26-targeted.	Optimisation saving 8 bytes per nesting level - performance fix for deeply nested sender expressions.
P3284R4	`write_env` and `unstoppable` Sender Adaptors	Eric Niebler	2025-02-14	Adopted (Sofia)	Adds missing sender adaptors for modifying execution environments.
P3685R0	Rename `async_scope_token`	Ian Petersen, Jessica Wong	2025-04-09	Adopted	Renames `async_scope_token` to `scope_token` for clarity.
P3706R0	Rename `join` and `nest` in async scope proposal	Ian Petersen, Jessica Wong	2025-04-09	Adopted	Renames `nest` to `associate` because original names were misleading.
P3325R5	A Utility for Creating Execution Environments	Eric Niebler	2025-05-22	Adopted	Adds `prop` and `env` class templates for creating and manipulating environments - fundamental infrastructure that was absent.
P2079R10	Parallel scheduler	Lee Howes	2025-06-02	Adopted (Sofia)	Provides `system_context` and `system_scheduler` - a basic execution context needed to actually run code.
P3149R11	`async_scope`	Ian Petersen, Jessica Wong, Kirk Shoop, et al.	2025-06-02	Adopted (Sofia)	Provides the async scope abstraction needed for safe non-sequential concurrency - replacing the removed `ensure_started` / `start_detached`.

Paper	Title	Author(s)	Date	Status	Change
P3164R4	Early Diagnostics for Sender Expressions	Eric Niebler	2025-06-02	Adopted	Moves diagnosis of invalid sender expressions to construction time rather than connection time.
P3552R3	Add a Coroutine Task Type	Dietmar Kuhl, Maikel Nadolski	2025-06-20	Adopted (Sofia)	Adds `std::execution::task` - the coroutine type that users need to use the framework. Adopted at Sofia with 29 abstentions and 11 against (77-11-29).
P3815R1	Add `scope_association` concept to P3149	Jessica Wong, Ian Petersen	2025-09-12	Adopted - closed Dec 2025. NB comment resolution.	Adds missing `scope_association` concept needed by the async scope facility.

Post-Adoption Issues

Paper	Title	Author(s)	Date	Status	Change
P3433R1	Allocator Support for Operation States	Dietmar Kuhl	2024-10-17	Plenary-approved - closed. LEWG Wroclaw (Nov 2024) approved design with strong consensus (SF:4/F:6/N:0/A:0/SA:0). Wording merged into draft.	Identifies allocator gaps in `ensure_started`, `split`, and `start_detached` operation states - confirming allocator support was missing from the original design.
P3481R5	`std::execution::bulk()` issues	Lucian Radu Teodorescu, Lewis Baker, Ruslan Arutyunyan	2024-10-17	Plenary-approved - closed. Five revisions. SG1 Wroclaw (Nov 2024) achieved unanimous consent on splitting bulk into `bulk`, `bulk_chunked`, and `bulk_unchunked`. Wording merged into draft.	Addresses outstanding issues with the bulk algorithm; required five revisions and addition of two new API variants (`bulk_chunked`, `bulk_unchunked`).

Paper	Title	Author(s)	Date	Status	Change
P3796R1	Coroutine Task Issues	Dietmar Kuhl	2025-07-24	In Progress - open, 2026-telecon milestone. Under active LEWG telecon review (Aug-Sep 2025); some sections achieved consensus, others still pending. Multiple LWG issues opened (4329-4332, 4344). Linked to NB comments.	Collects issues discovered after the task type was forwarded, including `unhandled_stopped` missing `noexcept` , wording issues, and performance concerns.
P3801R0	Concerns about the design of `std::execution::task`	Jonathan Wakely	2025-07-24	In Progress - open, 2026-telecon milestone. LEWG telecon review (2025-08-26) reached “no consensus” on the core stack overflow issue; “consensus against” treating dangling reference concern as C++26 blocker. Linked to NB comment.	Documents significant concerns including stack overflow risk due to lack of symmetric transfer support.
D3980R0	Task’s Allocator Use	Dietmar Kuhl	2026-01-25	In Progress - pre-London mailing. Addresses NB comments US 253, US 254, US 255, US 261.	Changes allocator propagation model for coroutine task types relative to P3552R3.

LWG Issues

Issue	Title	Date	Status	Change
LWG 4190	`completion-signatures-for` specification is recursive	2025-01-02	Open - circular dependency in spec.	Specification defect: recursive definition makes the requirement impossible to implement.
LWG 4206	`connect_result_t` should be constrained with `sender_to`	2025-02-04	Open - Priority 1.	Unconstrained alias causes hard errors instead of SFINAE-friendly failures.

Issue	Title	Date	Status	Change
LWG 4215	`run_loop::finish` should be `noexcept`	2025-02-13	Open - correctness bug.	Throwing `finish()` causes `sync_wait` to hang forever.
LWG 4260	Query objects must be default constructible	2025-05-07	Resolved (Kona 2025)	CPO constructors were not mandated `noexcept` .
LWG 4355	`connect-awaitable()` should mandate receiver completion-signals	2025-08-27	Open	Redundant `requires` -clause should defer to parent `Mandates` .
LWG 4356	`connect()` should use `get_allocator(get_env(rcvr))`	2025-08-27	Open - directly relevant to allocator sequencing issue.	`connect-awaitable` should respect the receiver's allocator.
LWG 4358	`[exec.as.awaitable]` uses Preconditions when should be constraint	2025-08-27	Resolved (Kona 2025)	Incorrectly-written preconditions should be `Mandates` .
LWG 4360	`awaitable-sender` concept should qualify `awaitable-receiver`	2025-08-27	Resolved (Kona 2025)	Ambiguous type reference in the `awaitable-sender` concept.
LWG 4368	Potential dangling reference from `transform_sender`	2025-08-31	Open - Priority 1. Stack-use-after-scope vulnerability.	Returns xvalue to a dead temporary - potential undefined behaviour.
LWG 4369	`check-types` for `upon_error` and `upon_stopped` is wrong	2025-08-31	Resolved (Kona 2025)	Uses `set_value_t` where `set_error_t` / `set_stopped_t` should be used.
LWG 4336	`bulk` vs. `task_scheduler`	2025-10-23	Open - NB comment. C++26-targeted.	`task_scheduler` does not parallelise bulk work - dispatching `bulk` via a `task_scheduler` wrapping `parallel_scheduler` serialises execution. P3927R0 proposes the fix.

Allocator-Related NB Comments (Kona 2025)

NB Comment	Title	Status
US 255 (LWG4335)	Use allocator from receiver's environment	Needs wording
US 253 (LWG4333)	Allow use of arbitrary allocators for coroutine frame	Needs wording

Total: 31 papers, 11 LWG issues, and 2 NB comments - 44 items modifying a single feature after its approval.

References

- P2079R10 Lee Howes. "Parallel scheduler." 2025-06-02.
- P2300R4 Michal Dominiak, et al. "std::execution." 2022-01-18.
- P2300R10 Michal Dominiak, et al. "std::execution." 2024-07-16.
- P2762R2 Dietmar Kuhl. "Sender/Receiver Interface For Networking." 2023-10-15.
- P2855R1 Ville Voutilainen. "Member customization points for Senders and Receivers." 2024-03-18.
- P2999R3 Eric Niebler. "Sender Algorithm Customization." 2024-04-16.
- P3149R11 Ian Petersen, Jessica Wong, Kirk Shoop, et al. "async_scope." 2025-06-02.
- P3175R3 Eric Niebler. "Reconsidering the std::execution::on algorithm." 2024-10-15.
- P3187R1 Lewis Baker, Eric Niebler. "Remove ensure_started and start_detached from P2300." 2024-10-15.
- P3433R1 Dietmar Kuhl. "Allocator Support for Operation States." 2025-06-18.
- P3552R3 Dietmar Kuhl, Maikel Nadolski. "Add a Coroutine Task Type." 2025-06-20.
- P3557R3 Eric Niebler. "High-Quality Sender Diagnostics with Constexpr Exceptions." 2025-06-10.
- P3796R1 Dietmar Kuhl. "Coroutine Task Issues." 2025-07-24.
- P3801R0 Jonathan Wakely. "Concerns about the design of std::execution::task." 2025-07-24.
- P3826R3 Eric Niebler. "Fix Sender Algorithm Customization." 2026-01-05.
- P3927R0 Eric Niebler. "task_scheduler Support for Parallel Bulk Execution." 2026-01-17.
- D3980R0 Dietmar Kuhl. "Task's Allocator Use." 2026-01-25.
- P4003R0 Vinnie Falco. "IoAwaitables: A Coroutines-First Execution Model." 2026-01-21.