Open Issues in `std::execution::task`

Document Number:	P4007R3
Date:	2026-05-01
Intent:	Inform
Audience:	LEWG

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

Table of Contents

Abstract

`std::execution::task` (P3552R3 [1]) had open issues identified by national ballot comments, LWG issues, and published papers. Croydon resolved several. This paper classifies each issue by whether it can be resolved after C++26 ships or whether shipping forecloses the fix, and notes which classified issues were addressed at Croydon.

Revision History

R3: May 2026 (pre-Brno mailing)

- Formatting corrections.

R2: April 2026 (post-Croydon mailing)

- Updated classification to reflect Croydon motions 28-29, 33, 35-38. Acknowledged issues resolved by P3941R4, P3927R2, P3980R1, and P4151R1. Rewrote allocator descriptions to credit P3980R1 while preserving residual frame-allocator concerns.
- Reference corrections.

R1: March 2026 (prior to Croydon meeting)

- Complete rewrite as an informational classification of open issues.

R0: February 2026 (pre-Croydon mailing)

- Original analysis of structural gaps. See P4007R3 [2].

---

1. Disclosure

The author provides information and serves at the pleasure of the committee.

Coroutine-native I/O and `std::execution` are complementary. Each serves the domain where its design choices pay off.

The authors developed P4007R3 [2] ("Senders and Coroutines") and P2583R4 [3] ("Symmetric Transfer and Sender Composition"). The classification below holds regardless of whether any alternative design exists.

This paper asks for nothing.

---

2. Fixed After Ship

`task`'s `promise_type` is a class template instantiated in user code. Its `operator new`, allocator selection, stop token storage, environment forwarding, and destruction ordering can change between standard revisions without binary incompatibility. These issues are fixable post-ship: Unusual Allocator Customisation [4], Flexible Allocator Position [4], Shadowing The Environment Allocator [4], Stop Source Always Created [4], Stop Token Default Constructible [4], Task Not Actually Lazily Started [4], Frame Destroyed Too Late [4], No Default Arguments [4], `unhandled_stopped` Not `noexcept` [4], Environment Design Inefficient [4], Non-Sender Awaitables Unsupported [4], Future Language Feature Could Avoid `co_yield` [4], No TLS Capture/Restore Hook [4], `return_value`/`return_void` Have No Specification [4], `co_return { args... }` Unsupported [4], `change_coroutine_scheduler` Requires Assignable Scheduler [4], Sender-Unaware Coroutines Cannot `co_await` a `task` [4], Missing Rvalue Qualification [4], Parameter Lifetime Is Surprising [5], No Protection Against Dangling References [5], `co_yield with_error` Is Clunky [5], `co_await schedule(sch)` Is an Expensive No-Op [5], Coroutine Cancellation Is Ad-Hoc [5].

Of these, Croydon resolved six: Unusual Allocator Customisation, Flexible Allocator Position, and Shadowing The Environment Allocator were addressed by P3980R1 [6]; `unhandled_stopped` Not `noexcept`, `change_coroutine_scheduler` Requires Assignable Scheduler (the mechanism was removed entirely), and Missing Rvalue Qualification were addressed by P3941R4 [13]. The

remaining issues are still open and fixable post-ship.

`affine` semantics (formerly `affine_on`, renamed by P4151R1 [14]), rescheduling behavior, and algorithm customization are specification-level concerns. Tightening requirements or adding default implementations does not change any published interface. These issues are fixable post-ship: `affine_on` Default Implementation Lacks Specification [4], `affine_on` Semantics Not Clear [4], `affine_on` Shape May Not Be Correct [4], `affine_on` Shouldn't Forward Stop Requests [4], `affine_on` Customisation For Other Senders [4], Starting a `task` Reschedules Unconditionally [4], Resuming After a `task` Reschedules Unnecessarily [4], `bulk` vs. `task_scheduler` [4], No Completion Scheduler [4], `with_awaitable_senders` Unused [4].

Of these, Croydon resolved eight. P3941R4 [13] rewrote scheduler affinity from scratch - `affine_on` was made unary, `change_coroutine_scheduler` removed, and `get_start_scheduler` introduced - resolving the five `affine_on`/`affine` items and both rescheduling items. P4151R1 [14] renamed `affine_on` to `affine`. P3927R2 [15] resolved `bulk` vs. `task_scheduler` by giving `task_scheduler` parallel bulk support. Two items remain open: No Completion Scheduler and `with_awaitable_senders` Unused.

---

3. Not Fixable Post-Ship

The issues in this section are items where shipping forecloses the fix.

Issue	References	Fixed
Allocator Timing	P3980R1 [6], P3796R1 [4], LWG 4356 [7], US 254-385 [8]	partial
Allocator Propagation	P3980R1 [6], P3796R1 [4]	partial
Error Return	P3950R0 [9], P3801R0 [5], P1713R0 [10]	no
Symmetric Transfer	P2583R4 [3], US 246-373 [11], LWG 4348 [12], P3801R0 [5], P3796R1 [4]	no

- **Allocator Timing.** P3980R1 [6] separates frame allocation from environment allocation: the environment allocator is now sourced from `get_allocator(get_env(rcvr))` at `connect` time, resolving environment-based injection. The frame allocator, however, remains call-site-specified via `allocator_arg` in the coroutine parameter list. This is a structural consequence of coroutine allocation timing - the frame is allocated by `operator new` before `connect`/`start` runs, so the receiver's environment is unavailable. Shipping standardizes the two-tier split. A design where frame allocation participates in environment-based propagation is foreclosed without a language change to coroutine allocation.
- **Allocator Propagation.** P3980R1 [6] provides environment allocator propagation: each `task` obtains its environment allocator from `get_allocator(get_env(rcvr))`, so a parent's environment allocator flows to children through the receiver chain. Frame allocator propagation remains absent. Each coroutine call site must independently specify `allocator_arg` to use a non-default frame allocator. In a deep coroutine call tree where frame allocation matters - arena-per-request, pool allocators, device memory - every function signature must carry `allocator_arg` explicitly. Shipping this design forecloses automatic frame allocator propagation through the coroutine call tree.

- **Error Return.** `task` requires `co_yield with_error(e)` to propagate an error to the caller. `co_return` cannot carry an error value because `return_value` and `return_void` are mutually exclusive in the current coroutine specification. Shipping this interface locks in the `co_yield` mechanism and forecloses `co_return`-based error propagation, which would require a language change.
- **Symmetric Transfer.** The completion functions (`set_value`, `set_error`, `set_stopped`) and `start()` return `void`, providing no channel to propagate a `coroutine_handle<>`. When a sender completes synchronously, the receiver calls `handle.resume()` on the caller's stack, adding a frame per completion with no upper bound. Shipping this protocol forecloses the `coroutine_handle<>`-returning completion protocol that would enable symmetric transfer.

Jonathan Müller wrote in P3801R0 [5]: *"Having iterative code that is actually recursive is a potential security vulnerability."* The same paper observes: *"This is surprising behavior that can lead to unnecessary memory consumption and potentially hard to figure out bugs. It fundamentally breaks the promises of RAII where destruction is strictly tied to the end of a scope."*

In April 2026, @mika-fischer reported [16] a production crash in stdexec where *"Destroying the spawn state destroys the task operation, which destroys the currently executing task coroutine frame, including the sender awaiter whose `await_suspend()` has not returned yet."*

Acknowledgements

The authors thank Andrzej Krzemieński for feedback that sharpened the scope of this revision. Thanks are also due to Dietmar Kühl, Michael Hava, Mark Hoemmen, Ian Petersen, and Ville Voutilainen for technical discussion that informed the analysis.

References

Papers, issues, and ballot comments referenced in this document.

WG21 Papers

[1] P3552R3 - "Add a Coroutine Task Type" (Dietmar Kühl, Maikel Nadolski, 2025).

[2] P4007R3 - "Open Issues in `std::execution::task`" (Vinnie Falco, Mungo Gill, 2026).

[3] P2583R4 - "Symmetric Transfer and Sender Composition" (Mungo Gill, Vinnie Falco, 2026).

[4] P3796R1 - "Coroutine Task Issues" (Dietmar Kühl, 2025).

[5] P3801R0 - "Concerns about the design of std::execution::task" (Jonathan Müller, 2025).

[6] P3980R1 - "Task's Allocator Use" (Dietmar Kühl, 2026).

[7] LWG 4356 - "connect() should use get_allocator(get_env(rcvr))".

[8] US 254-385 - C++26 NB ballot comment.

[9] P3950R0 - "return_value & return_void Are Not Mutually Exclusive" (Robert Leahy, 2025).

[10] P1713R0 - "Allowing both co_return; and co_return value; in the same coroutine" (Lewis Baker, 2019).

[11] US 246-373 - C++26 NB ballot comment.

[12] LWG 4348 - "task doesn't support symmetric transfer".

Croydon Papers (R2)

[13] P3941R4 - "Scheduler Affinity" (Dietmar Kühl, 2026).

[14] P4151R1 - "Rename affine_on" (Robert Leahy, 2026).

[15] P3927R2 - "task_scheduler Bulk Execution" (Eric Niebler, 2026).

[16] NVIDIA/stdexec issue #2047: Another crash with stdexec::task - @mika-fischer, April 2026.