On the Diversity of Coroutine Task Types

Document Number:	P4089R0
Date:	2026-04-17
Intent:	Inform
Audience:	LEWG

Reply-to:	Vinnie Falco vinnie.falco@gmail.com C++ Alliance Proposal Team

Table of Contents

Abstract

The Environment parameter in `std::execution::task` makes cross-library coroutine interoperability structurally impossible without knowing every query by name.

`std::execution::task<T, Environment>` (P3552R3[1], "Add a Coroutine Task Type") was approved as the lingua franca for coroutine-based asynchronous code. The `Environment` parameter is an open query-response protocol whose interoperability surface is defined by a single concept: `queryable`, which is `destructible`. This paper asks a simple question: when two libraries define different environments, how does one task `co_await` the other? The answer, traced step by step through the specification, is that no general conversion exists. The query set is open by design, and the only adaptation mechanism - `write_env` - requires the caller to know every missing query by name. The risk to the ecosystem is structural, documented by the specification itself, by NVIDIA's reference implementation, by the only production precedent (Boost.Asio), and by `task`'s own author.

Revision History

R0: April 2026 (post-Croydon mailing)

- Initial version.

---

1. Disclosure

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

This paper is part of the Network Endeavor (P4100R0), a project to bring coroutine-native I/O to C++.

The author developed and maintains Capy[22] and Corosio[23] and believes coroutine-native I/O is a practical foundation for networking in C++.

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

The author has published Boost libraries and has a stake in the project's success.

The cross-library bridges (Section 8) were authored by Klemens Morgenstern. The frame allocator gap was identified by Peter Dimov. Neither is a co-author.

This paper asks for nothing.

---

2. P3552R3

The findings documented in this paper and in P3801R0[2], "Concerns about the design of `std::execution::task`," are structural consequences of making `task` do more than one thing. Dietmar Kühl responded to every concern with professionalism and technical precision, and wrote P3796R1[3], "Coroutine Task Issues," to collect the issues himself.

The Environment parameter is a structural requirement imposed by P2300.

2.1 Scope

P3552R3[1] was plenary-approved for C++26 at Sofia. Six Croydon motions modified `task` after plenary approval - restructuring the allocator model (P3980R1[4]), rewriting scheduler affinity with a new `get_start_scheduler` query and renaming `affine_on` to `affine` (P3941R4[24]), overhauling sender algorithm customization, and adding parallel bulk execution to `task_scheduler`. None of these changes address the `Environment` template parameter, the open query protocol, or the structural interoperability risk documented here.

Three topics are outside scope. Allocator timing - how and when the frame allocator reaches `operator new` - was addressed at Croydon in P3980R1[4]. Allocator propagation - how the allocator flows to child operations through the environment - is an engineering problem with known solutions. Categorization of compound I/O results into the three channels (`set_value`, `set_error`, `set_stopped`) is the subject of P4091R0[5] and P4090R0[6], not this paper.

---

3. The Claim

A standard task type provides a lingua franca. It eliminates pairwise bridges. It gives the ecosystem a common type that every library can accept and return. P3552R3[1] Section 3 states: "The `task` coroutine provided by the standard library may not always fit user's needs." SG1 discussion notes on P1056R1[25], "Add lazy coroutine (coroutine task) type," record (reproduced in P3552R0[26]): "There can be more than one `task` type for different needs."

The claim is that `std::execution::task` serves as that lingua franca. This paper stress-tests that claim.

---

4. `destructible`

P2300R10 defines `queryable` as the base concept for all objects that respond to property queries - schedulers, senders, receivers, and environments all refine it. P2300R10[7], "std::execution," defines `queryable` in [exec.queryable.concept][27] as follows:

```
template<class T>
  concept queryable = destructible<T>;
```

We will return to this.

5. Two Libraries, Two Environments

The `Environment` parameter is an extension point. Two libraries will define two environments. The progression below begins with empty environments and escalates through standard type mismatches, custom queries, and NVIDIA's deployed GPU environment. Each step is individually reasonable. The specification provides no general conversion mechanism between environments.

5.1 From trivial to standard divergence

Library A	Library B
`struct env_a {};`	`struct env_b {};`
`task<int, env_a> compute();`	`task<int, env_b> fetch();`

Both environments are empty. Both produce identical default behavior. The types are incompatible - a function accepting `task<int, env_a>` cannot accept `task<int, env_b>`. A converting constructor (`env_a(auto const&)`) compiles but discards everything; the moment either environment carries state, the constructor must know what to extract.

A second axis of divergence is observable with standard nested types. If Library A binds a concrete scheduler for performance - exactly what Asio users do when they replace `any_io_executor` with `io_context::executor_type` - the types diverge further:

```
struct env_a {
    using scheduler_type = my_thread_pool;
};
```

Library B uses the default type-erased `task_scheduler`. The `scheduler_type` mismatch means the inner task cannot extract the scheduler it needs. Add a second axis - `allocator_type` - and the construction protocol cannot reconcile the choices. Still just standard nested types. No custom queries yet. Already incompatible.

5.2 Custom queries and `write_env`

Library A needs to propagate tenant context in a multi-tenant service:

```
struct get_tenant_id_t
{
    static constexpr auto
    query(forwarding_query_t) noexcept
        -> bool { return true; }
};
inline constexpr get_tenant_id_t
    get_tenant_id{};

struct env_a {
    using scheduler_type = my_thread_pool;
    tenant_id tid;
    auto query(get_tenant_id_t) const
        -> tenant_id { return tid; }
};
```

One custom query. Completely reasonable. Library B knows nothing about `get_tenant_id`. P2300R10[7] provides `write_env` in [exec.write.env] - the caller names each missing query and provides its value by hand:

```
auto adapted = write_env(
    compute(),
    prop{get_tenant_id, my_tid});
```

It compiles. It works. But the moment Library B also defines a custom query - `get_connection_pool` - the caller must inject `get_tenant_id` into one side *and* `get_connection_pool` into the other. The caller must know every custom query from every library, by name, and inject them all manually. There is no discovery mechanism - no way to ask an environment "what queries do you need?" For two libraries with one custom query each, a determined caller can make it work. Tedious, but possible.

5.3 What do real domains look like?

NVIDIA's reference implementation defines custom forwarding queries in nvexec/stream/common.cuh[28]:

```
struct get_stream_provider_t
    : __query<get_stream_provider_t>
{
    static constexpr auto
    query(forwarding_query_t) noexcept
        -> bool { return true; }
};
```

The `stream_provider` carries a CUDA stream, pinned and managed memory resources, a stream pool, a task hub, and a priority level:

```
struct stream_provider
{
    cudaError_t status_{cudaSuccess};
    std::optional<cudaStream_t> own_stream_{};
    context context_;
    std::mutex custodian_;
    std::vector<std::function<void()>>
        cemetery_;
};
```

Where `context` carries:

```
struct context
{
    std::pmr::memory_resource*
        pinned_resource_;
    std::pmr::memory_resource*
        managed_resource_;
    stream_pools_t* stream_pools_;
    queue::task_hub* hub_;
    stream_priority priority_;
};
```

This is injected into the environment via `make_stream_env`:

```
auto make_stream_env(
    BaseEnv&& base_env,
    stream_provider* sp) noexcept
{
    return __env::__join(
        prop{get_stream_provider, sp},
        static_cast<BaseEnv&&>(base_env));
}
```

This is not hypothetical. This is deployed code in the `std::execution` reference implementation. Now imagine a networking domain with its own custom queries:

GPU domain (nvexec)	Network domain
`get_stream_provider` -> `stream_provider*`	`get_ssl_context` -> `ssl::context*`
`get_stream` -> `cudaStream_t`	`get_connection_pool` -> `pool*`
pinned/managed memory resources, stream pool, ...	...

Neither environment can construct itself from the other. The caller would need to inject `get_stream_provider` - which requires a `stream_provider*` pointing to a live object managing CUDA resources - plus `get_ssl_context`, `get_connection_pool`, and every other custom query from every domain involved. The caller must construct the domain-specific objects correctly, manage their lifetimes, and know the semantic requirements of each query. `write_env` does not help with any of this. It is a syntactic mechanism for injecting key-value pairs. The semantic burden is entirely on the caller.

5.4 What about the standard queries?

P2300R10[7] defines seven forwarding queries: `get_scheduler`, `get_allocator`, `get_stop_token`, `get_domain`, `get_delegation_scheduler`, `get_forward_progress_guarantee`, and `get_completion_scheduler<Tag>`. P3552R3[1] adds an eighth: `get_await_completion_adaptor`. Croydon added a ninth: `get_start_scheduler` (P3941R4[24]), distinct from `get_scheduler`. The standard set continues to grow.

A conversion layer could forward these. But NVIDIA already defines custom queries (`get_stream_provider`, `get_stream`) that are not in this list. Any domain that needs custom queries - GPU, networking, database, audio - is outside the standard set. The standard queries are the *minimum*. The `Environment` parameter exists precisely so domains can add *more*. P3552R3[1] itself defines a custom `get_value_t` query in its own Section 4.7.

The design *encourages* custom queries. A conversion that handles only standard queries defeats the purpose of the open protocol.

5.5 `destructible`

No general conversion exists. The query set is open by design. `write_env` requires foreknowledge of every query. Forwarding standard queries misses the custom queries that justify the Environment's existence.

Good stewardship of the standard means shipping features narrow and widening with evidence. The committee has practiced this consistently - `std::string_view`, `std::span`, the Lakos Rule.

Every concept in the sender/receiver model eventually bottoms out at `queryable`:

Concept	Refines
`scheduler`	`queryable`, `copyable`, `equality_comparable`, ...
`sender`	`queryable`, `move_constructible`, ...
`receiver`	`queryable`, `move_constructible`, ...
`queryable`	

[exec]	[exec.queryable.concept]
6,607 lines	2 lines

`queryable` is `destructible`. It cannot be narrowed after C++26 ships.

N libraries with N different environments produce N incompatible task types with no general conversion path.

Jonathan Müller wrote in P3801R0[2]:

> *"As a standardization committee, we are drafting a standard that should outlive us. We are not working on some open-source library, we are designing the foundation for an entire ecosystem."*

5.6 If `Environment` has a default, who customizes it?

Nobody	Somebody
The parameter is unnecessary.	The types are incompatible (Section 5.1).

A default delays the arrival of column two. Section 6 documents how long the delay lasted for Asio. The parameter exists so that libraries provide non-default environments. The moment one does, column two applies.

6. The Asio Precedent

The only production two-parameter coroutine type is Asio's `awaitable<T, Executor>`. Asio provides a type-erased default - `any_io_executor` - that works for the majority of users. Most users stay on the default and succeed. But when users deviate - for performance, for concrete executor binding - the type incompatibility surfaces. Five independent reports illustrate the kind of failure the `Environment` parameter will produce at greater scale:

- A user cannot `co_await` a custom awaitable inside an Asio coroutine (StackOverflow #69280674[13], 11 upvotes, 3,014 views): "I am struggling to find out how to implement custom awaitable functions."
- A user describes Asio's executor behavior in coroutines as "rather confusing" and "very irritating" (StackOverflow #78593944[14], 990 views).
- A user hits compilation errors from a concrete executor type (StackOverflow #79115751[15]).
- A user cannot `co_await` across `io_context` boundaries (StackOverflow #73517163[16]).
- A user cannot create custom awaitable functions (GitHub asio #795[17]).

Asio's case is mild. The executor concept has a closed interface. `any_io_executor` provides an escape hatch at the cost of one virtual dispatch per operation. Users who need performance can opt into a concrete executor and accept the type incompatibility.

P3552R3[1]'s `Environment` has an open interface. The query set is unbounded by design - no fixed set of operations to erase against. The type-erasure mechanism that Asio provides for executors has no analogue for an open query protocol. The mild case already exhibits the predicted symptoms at smaller scale.

---

7. Design Intent

The C++20 coroutine mechanism separates the return type from the promise type by design. Nicol Bolas explained the rationale on StackOverflow[18]:

> *"That a function is a coroutine is intended to be an implementation detail of the function. Nothing in a function declaration requires it to be a coroutine. Including the return type."*

The separation was designed so that a library can change its internals from callbacks to coroutines without changing the return type. `task<T, Environment>` puts sender policy in the return type, violating this separation.

Bolas continued[18]:

> *"Separating the two is intended to allow existing types that support continuations to be re-implemented internally as coroutines."*

When the Environment changes, the return type changes, and every caller breaks. The separation was designed to prevent exactly this.

Gor Nishanov designed the coroutine mechanism with an explicit layering model. P1362R0[8] Section 4.4 and N4287[29] define four tiers:

Who	What
Everybody (millions)	Uses coroutines and awaitables defined by the standard library, Boost, and other high-quality libraries
Power user (10,000)	Defines new awaitables to customize await for their environment **using existing coroutine types**
Expert (1,000)	Defines new coroutine types
Cream of the crop (200)	Defines metafunctions, adapters, and composition utilities for coroutines

The middle tier is the one that matters. Power users customize the *environment* by defining new *awaitables* - not by changing the coroutine type. The coroutine type stays fixed; the awaitable carries the domain-specific protocol. The `Environment` parameter inverts this layering: it puts environment customization in the coroutine type itself, forcing a new type for each domain. What Nishanov assigned to the awaitable tier, `task<T, Environment>` moves into the expert tier and exposes in the return type.

Nishanov reinforced the principle in P0975R0[9]:

> *"Unlike most other languages that support coroutines, C++ coroutines are open and not tied to any particular runtime or generator type and allow libraries to imbue coroutines with meaning."*

And at CppCon 2017[20]:

> *"Different category of people have different desires for their asynchronous runtime so coroutines are open ended, they can hook up to anything you want."*

The coroutine mechanism was designed so that environment diversity lives in awaitables, not in the coroutine type. The `Environment` parameter moves that diversity into the return type, producing the fragmentation that Section 5 documents.

---

8. The Ecosystem

Nine coroutine libraries are surveyed below. Asio is the most widely deployed C++ async library; the standard proposal carries normative weight. The convergence argument is not about counting libraries - it is about independent design decisions. Five independent teams - plus the author's Capy and Morgenstern's Cobalt - all arrived at one parameter. The two that added a second parameter both needed an escape hatch: Asio provides `any_io_executor`; P3552R3[1] does not.

Library	Declaration	Params	Source
asyncpp	`template<class T> class task`	1	task.hpp
Boost.Cobalt	`template<class T> class task`	1	task.hpp
Capy	`template<class T = void> struct task`	1	task.hpp
cppcoro	`template<typename T> class task`	1	task.hpp
aiopp	`template<typename Result> class Task`	1	task.hpp
libcoro	`template<typename return_type> class task`	1	task.hpp
folly::coro	`template<typename T> class Task`	1	Task.h
Boost.Asio	`template<class T, class Executor = any_io_executor> class awaitable`	2	awaitable.hpp
P3552R3 (std)	`template<class T, class Environment> class task`	2	task.hpp

The interface converged. The machinery diverged:

Library	Invariant enforced through promise
folly::coro	Cancellation token, fiber scheduler, async stack traces
Boost.Cobalt	Intrusive list cancellation, Asio executor binding
Google `Co`	Immovable prvalue-only `co_await` - prevents dangling references
Capy	`io_env` propagation: executor, stop token, frame allocator
cppcoro	Symmetric transfer, lazy start
C++26 `task`	`affine` scheduler (via `get_start_scheduler`), sender environment, stop token via connect

Jonathan Müller describes Google's approach in P3801R0[2]:

> *"Google's coroutine library has a pure library solution: Their default coroutine type `Co` is immovable and `co_await` takes it by-value. That way you can only `co_await` prvalue coroutines, which makes it impossible to have dangling references."*

Google needed a safety property that no other library provides. One template parameter. The promise enforces the invariant. Callers see `Co<T>`. The one-parameter design let them build it without fragmenting the ecosystem.

Domain-specific task types interoperate through the C++20 awaitable protocol. The cross_await[30] repository (Klemens Morgenstern) contains four cross-library composition examples, 51-105 lines each. Sender bridges follow the same pattern (P4092R0[10], P4093R0[11]). The ecosystem independently arrived at the design that avoids the problem documented in Section 5.

---

9. Concepts Mitigate the Risk

The fix is a general pattern: define a concept that constrains awaitables, not task types. The promise remains the extension point. The return type stays clean.

`IoAwaitable` (P4003R0[12]) is one realization:

```
template<typename A>
concept IoAwaitable =
    requires(A a, std::coroutine_handle<> h,
             io_env const* env)
    {
        a.await_suspend(h, env);
    };
```

Any type satisfying the concept works. The task type remains `task<T>` - one parameter. The promise delivers the domain environment through `await_transform`.

The concept constrains the awaitable, not the task type. Any task type whose promise propagates `io_env` can `co_await` any `IoAwaitable` - N task types plus M awaitables equals N+M implementations.

The standard queries that P2300R10 propagates through environments have natural homes in this model. The scheduler and stop token reach the awaitable through `await_transform`, which passes the promise's state into the awaitable's `await_suspend` - exactly what `IoAwaitable`'s signature captures. The allocator reaches the coroutine frame through `promise_type::operator new`, which the compiler already calls with the coroutine's parameters. The detailed mapping is in P4003R0[12].

Bolas explained the role of `await_transform`[19]:

> *"To declare the existence of such a function is to declare that your promise cannot work unless it instruments the awaitable... It is also useful for being able to actively shut off specific awaitables, or to only allow the use of specific awaitables."*

A concept with a domain-specific `await_suspend` signature gets compile-time rejection of foreign awaitables for free. `IoAwaitable` is one such concept. It is not the only one. The pattern generalizes to any domain.

`IoAwaitable` is a concept, not a type. It constrains what a promise can `co_await`, not what a coroutine returns. A promise that does not define `await_transform` for `IoAwaitable` gets a compile-time error - not a silent type mismatch. This is the difference between concept-level and type-level fragmentation: `task<T, Environment>` changes the return type when the Environment changes, breaking every caller. A concept lets the promise declare which domains it supports.

Concept-level incompatibility still exists. A promise that does not provide `await_transform` for a given domain's awaitables cannot `co_await` them. The incompatibility surfaces at the `co_await` site as a compile-time error - local and diagnosable, not viral.

---

10. Frequently Raised Concerns

Q1: Nicol Bolas is not normative. Bolas is explaining the rationale, not writing the standard. The normative backing is Nishanov's P0975R0[9] ("open and not tied to any particular runtime") and P1362R0[8] (layered complexity model), both published WG21 papers. Even setting aside design-intent sources, the empirical evidence in Section 8 - seven independent libraries converging on one-parameter designs - shows that the ecosystem treats the principle as operative.

Q2: `IoAwaitable` is the author's own design. `IoAwaitable` is one realization of the principle. The cross_await bridges (Section 8, Klemens Morgenstern, independent author) and Google's `Co<T>` both use the same principle without `IoAwaitable`. The principle is: domain-specific invariants belong in the promise, not in the return type.

Q3: The Environment parameter serves a real need. Agreed. The Environment creates a structural risk to task type diversity. The question is whether the risk is worth accepting for domains outside the sender model.

Q4: Production libraries do not interoperate anyway. The cross_await bridges in Section 8 refute this. Four working examples, 51-105 lines each. The C++20 awaitable protocol is the interop surface. Diversity with interoperability is the design intent (Nishanov P1362R0[8]: "mix and match").

Q5: A default Environment will fix the fragmentation. Even with a default, the parameter exists so users provide non-default environments. The moment they do, fragmentation begins. Adding a default delays the risk; it does not prevent it. Section 5 traces the progression from empty environments through NVIDIA's deployed queries - the default does not change the outcome.

Q6: These issues will be fixed in a future revision. Section 2.1 separates engineering topics from structural findings. The open query set and the absence of a general conversion mechanism are consequences of the design decision to make the Environment an open protocol. They cannot be fixed without closing the protocol - which would break every custom query, including NVIDIA's (Section 5.3).

Q7: A standard task type provides a lingua franca. The benefit is real. The question is whether `task<T, Environment>` delivers it. Section 5 shows that the lingua franca holds only when every library uses the same Environment. The moment two libraries use different environments, the types are incompatible (Section 5.1) - and the parameter exists precisely so that users provide non-default environments. Nishanov's layering model (Section 7) assigns environment customization to the awaitable tier, not the coroutine type. A concept is a stronger lingua franca than a concrete type with a second template parameter that fragments on contact with itself.

---

11. Conclusion

The `Environment` parameter in P3552R3[1] creates a structural risk to the task type diversity that C++20 coroutines were designed to enable. The risk is documented by the specification (the open query set, Section 4), by the reference implementation (NVIDIA's custom queries, Section 5.3), by the only production precedent (Asio, Section 6), and by `task`'s own author (Section 3). Two complementary models - each contributing what the other cannot - make a stronger standard than one model asked to serve both.

Acknowledgments

The author thanks Gor Nishanov for the coroutine model's explicit support for task type diversity; Peter Dimov for identifying the frame allocator propagation gap; Klemens Morgenstern for Boost.Cobalt and the cross_await bridges; Dietmar Kühl for P3552R3[1], P3796R1[3], and `beman::execution`; Jonathan Müller for confirming the symmetric transfer gap in P3801R0[2] and for documenting Google's `Co` design; Aaron Jacobs for the CppNow 2024 presentation on coroutines at scale[21]; Steve Gerbino for the constructed comparison and bridge implementations; and Mungo Gill, Mohammad Nejati, and Michael Vandeberg for feedback.

References

WG21 Papers

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

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

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

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

[5] P4091R0 - "Two Error Models" (Vinnie Falco, 2026).

[6] P4090R0 - "Sender I/O: A Constructed Comparison" (Vinnie Falco, Steve Gerbino, 2026).

[7] P2300R10 - "std::execution" (Michał Dominiak et al., 2024).

[8] P1362R0 - "Incremental Approach: Coroutine TS + Core Coroutines" (Gor Nishanov, 2018).

[9] P0975R0 - "Impact of coroutines on current and upcoming library facilities" (Gor Nishanov, 2018).

[10] P4092R0 - "Consuming Senders from Coroutine-Native Code" (Vinnie Falco, Steve Gerbino, 2026).

[11] P4093R0 - "Producing Senders from Coroutine-Native Code" (Vinnie Falco, Steve Gerbino, 2026).

[12] P4003R0 - "Coroutines for I/O" (Vinnie Falco, Steve Gerbino, Mungo Gill, 2026).

StackOverflow and GitHub

[13] "co_await custom awaiter in boost asio coroutine" StackOverflow #69280674

[14] "Boost Asio: Executors in C++20 coroutines" StackOverflow #78593944

[15] "Boost asio using concrete executor type with c++20 coroutines causes compilation errors" StackOverflow #79115751

[16] "Can I co_await an operation executed by one io_context in a coroutine executed by another in Asio?" StackOverflow #73517163

[17] "How to create custom awaitable functions" GitHub asio #795

[18] Nicol Bolas, "Why is the promise type separated from the coroutine object?" StackOverflow #68167497

[19] Nicol Bolas, "Why is `promise_type::await_transform` greedy?" StackOverflow #76110225

Talks

[20] Gor Nishanov, "Naked coroutines live (with networking)," CppCon 2017

[21] Aaron Jacobs, "C++ Coroutines at Scale - Implementation Choices at Google," CppNow 2024

Other

[22] Capy - Coroutine primitives library (Vinnie Falco).

[23] Corosio - Coroutine-native networking library.

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

[25] P1056R1 - "Add lazy coroutine (coroutine task) type" (Lewis Baker, Gor Nishanov, 2019).

[26] P3552R0 - "Add a Coroutine Lazy Type" (Dietmar Kühl, Maikel Nadolski, 2025).

[27] C++ Working Draft (Richard Smith, ed.).

[28] NVIDIA, nvexec/stream/common.cuh - GPU stream environment queries.

[29] Gor Nishanov, N4287 - "Threads, Fibers and Couroutines (slides deck)" (Gor Nishanov, 2014).

[30] Klemens Morgenstern, cross_await - "co_await one coroutine library from another" (2026).