What C++20 Coroutines Already Buy The Standard

Document Number:	P4088R1
Date:	2026-05-01
Intent:	Inform
Audience:	LEWG

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

Table of Contents

Abstract

Revision History

- R1: May 2026 (pre-Brno mailing)
- R0: April 2026 (post-Croydon mailing)

1. Disclosure
2. Two Models Already Ship
   - 2.1 Three Keywords
   - 2.2 Thirty Algorithms
   - 2.3 The Standard Ships Both
   - 2.4 The Chronology
   - 2.5 The Incumbent
3. What Senders Buy
4. What Coroutines Pay
5. Twenty Years, Nine Lines
   - 5.1 Asio's AsyncReadStream
   - 5.2 The C++20 Concept
   - 5.3 What Changed
6. Why Did It Take Twenty Years?
7. What The Frame Buys
   - 7.1 The Caller Is Erased
   - 7.2 The Operation State Is Concrete
   - 7.3 The Operation State Lives in the Socket
   - 7.4 Zero Per-Operation Allocation
   - 7.5 Compile Once
   - 7.6 Both Modes, One Abstraction
   - 7.7 Forty Years of ABI
   - 7.8 The User Chooses
   - 7.9 The Frame Subsidizes Everything

8. One Allocation Per Operation
9. Anticipated Objections
10. The Design Fork
11. Conclusion
Acknowledgments
References

Abstract

C++ already got an asynchronous model: regular C++20 coroutines.

The programmer writes `for`, `if`, structured bindings, and `co_await` - the idiom they already know, made asynchronous by three keywords. Underneath, five language mechanisms combine to produce type-erased streams, separate compilation, and ABI stability - properties the committee could not ship in twenty-one years of networking attempts. This paper traces the causal chain from mechanism to library and states the price.

Revision History

R1: May 2026 (pre-Brno mailing)

- Corrected N1925 attribution (Gerhard Wesp, not Kohlhoff).
- Corrected `when_all` description (concurrent joins, not sequential statements).
- Corrected St. Louis meeting date (July 2024).
- Formatting corrections.

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 [1]), a project to bring coroutine-native I/O to C++.

The author developed and maintains Capy [2] and Corosio [3] 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.

This paper asks for nothing.

---

2. Two Models Already Ship

C++26 ships two async models: coroutines and senders.

2.1 Three Keywords

Coroutine I/O is not a new programming model. It is `for`, `if`, `while`, `break`, `return`, structured bindings - the language the programmer already writes. Three keywords are new: `co_await`, `co_return`, `co_yield`.

The coroutine "tutorial" is: write regular code, put `co_await` before async operations.

2.2 Thirty Algorithms

The sender model replaces these constructs with library equivalents: `let_value` for local variables, `then` for function calls, `upon_error` for `catch`, `when_all` for concurrent joins, `repeat_effect_until` for `for`. P4014R2 [4] is a progressive tutorial of all thirty sender algorithms in C++26. The `retry` algorithm requires approximately 120 lines of template machinery as a sender; the coroutine equivalent is seven lines. The four-layer composition example in P4014R2 [4] Section 12 collapses from interleaved sender pipelines into a single coroutine.

Model	New vocabulary
Coroutines	`co_await`, `co_return`, `co_yield`
Senders	`just`, `just_error`, `just_stopped`, `sync_wait`, `then`, `upon_error`, `upon_stopped`, `let_value`, `let_error`, `let_stopped`, `schedule`, `starts_on`, `continues_on`, `on`, `affine`, `schedule_from`, `read_env`, `write_env`, `unstoppable`, `when_all`, `when_all_with_variant`, `split`, `into_variant`, `stopped_as_optional`, `stopped_as_error`, `bulk`, `bulk_chunked`, `bulk_unchunked`, `associate`, `spawn_future`

2.3 The Standard Ships Both

P3552R3 [5] "Add a Coroutine Task Type" Section 9.4.1 [task.overview] defines the result:

> *"The `task` class template represents a sender that can be used as the return type of coroutines."*

`std::execution::task` is a coroutine that is also a sender.

On September 28, 2021, the Executors telecon polled (P2453R0 [6] "2021 October Library Evolution Poll Outcomes"):

> *"We believe we need one grand unified model for asynchronous execution in the C++ Standard Library, that covers structured concurrency, event based programming, active patterns, etc."*
>
> *SF:4 / WF:9 / N:5 / WA:5 / SA:1 - No consensus (leaning in favor).*

The "one model" premise did not achieve consensus. C++26 now ships two.

2.4 The Chronology

C++20 was ratified in 2020 with coroutines as a language feature. Every major compiler implements them. Production codebases have used them for six years. `std::execution` was adopted into the working draft at St. Louis in July 2024 [7] and ships in C++26. Networking is not in the C++ standard. Twenty-one years from N1925 [8] "A Proposal to Add Networking Utilities to the C++ Standard Library."

2.5 The Incumbent

Coroutines are the language's computation model for async. `std::execution` is the library's. Both are in the standard. Both must succeed. The question is what each one is for.

Three execution models complement each other in the C++26 standard: parallel algorithms with execution policies, `std::execution` with sender/receiver composition, and coroutines with `co_await`. These three span different points on the abstraction spectrum - execution policies annotate synchronous calls with parallelism, senders compose asynchronous work graphs, coroutines suspend and resume sequential code. These three existed before P0443R14 [9] "A Unified Executors Proposal for C++" tried to unify them. The unification never happened. `std::sort(std::execution::par, first, last)` does not use senders. P2500R2 [10] "C++ parallel algorithms and P2300", the only paper that attempts the bridge, has not been revised since October 2023, was never adopted, and leaves the customization mechanism unspecified. The existence of P2500R2 [10] is itself the evidence: a universal execution model does not require a bridge paper to reach the execution model already in the standard. The parallel algorithms shipped in C++17. `std::execution` ships in C++26. Coroutines shipped in C++20. The committee accepted complementary execution models when it shipped all three.

Coroutine-native I/O does not introduce a fourth model. It completes the third. C++20 gave the committee `co_await`, `coroutine_handle<>`, and `promise_type`. It did not give the committee standard I/O operations that use them. This paper documents what happens when you build I/O on the model the language already provides.

---

3. What Senders Buy

The sender/receiver model is an achievement.

Eric Niebler described the philosophical foundation in 2020 [11]:

> *"It brings the Modern C++ style to our async programs by making async lifetimes correspond to ordinary C++ lexical scopes, eliminating the need for reference counting to manage object lifetime."*

Child operations complete before their parents. Lexical scopes govern async lifetimes. The same discipline that makes synchronous C++ safe - RAII, deterministic destruction, nested scopes - extends to asynchronous code. This is the sender model's deepest conviction.

The practical motivation is equally clear. Niebler wrote in 2024 [12]:

> *"There's nothing wrong with the callback API. What's wrong is that every library that exposes asynchrony uses a slightly different callback API. If you want to chain two async operations from two different libraries, you're going to need to write a bunch of glue code to map this async abstraction to that async abstraction. It's the Tower of Babel problem."*

One standard async abstraction solves the interoperability problem. Senders provide the common vocabulary.

Senders and coroutines are not either/or. Niebler framed this directly [12]:

> *"If your library exposes asynchrony, then returning a sender is a great choice: your users can await the sender in a coroutine if they like, or they can avoid the coroutine frame allocation and use the sender with a generic algorithm like `then()` or `when_all()`. The lack of allocations makes senders an especially good choice for embedded developers."*

The sender model provides zero-allocation pipelines through a specific design choice: `connect(sender, receiver)` produces an operation state that aggregates all data before `start()` is called. Niebler described the consequence [12]:

> *"That means we can launch lots of async work with complex dependencies with only a single dynamic allocation or, in some cases, no allocations at all."*

Separating construction from launch lets the pipeline aggregate all state before any work starts. The optimizer sees the full pipeline. This is possible because the operation state is parameterized on the receiver type - the compiler knows the complete type at every stage.

Senders also provide completion signatures as type-level contracts. The sender declares how it can complete. A type mismatch between pipeline stages is a compile error. The three-channel model - `set_value`, `set_error`, `set_stopped` - routes results by channel, and generic algorithms like `retry`, `when_all`, and `upon_error` dispatch on the channel without knowing the

concrete sender type.

These are deployed at scale. P2470R0 [13] "Slides for presentation of P2300R2" documented the deployments: Facebook ("monthly users number in the billions"), NVIDIA ("fully invested in P2300... we plan to ship in production"), and Bloomberg (experimentation). GPU dispatch, infrastructure, HPC - the domains where compile-time work graphs, zero-allocation pipelines, and heterogeneous composition deliver their full value.

---

4. What Coroutines Pay

Coroutines are not free. Three costs are irreducible.

Frame allocation. When a function becomes a coroutine, the compiler moves everything that would normally live on the stack - every local variable, every function parameter, the suspension point that records where execution left off, and the awaitable machinery that manages resumption - into a heap-allocated block called the coroutine frame. Every coroutine that suspends must allocate this frame through `operator new`. The frame size is determined by the compiler. The caller cannot `sizeof` it, cannot stack-allocate it, cannot embed it in a struct. HALO (P0981R0 [14] "Halo: coroutine Heap Allocation eLision Optimization") can elide the allocation when the compiler proves the frame's lifetime is bounded by the caller's scope, but no compiler guarantees HALO. The recycling allocator (recycling_memory_resource [2]) amortizes the cost to a thread-local pool lookup - nanoseconds instead of microseconds - but the allocation still happens. Senders do not pay this cost. Sender operation states can be stack-allocated or embedded in the parent's operation state.

Opaque resume. The compiler cannot see through `std::coroutine_handle<>::resume()`. Every suspension point is an optimization barrier. The optimizer cannot inline across it. In tight inner loops this is measurable. This is the fundamental cost of type erasure through the handle. Senders do not pay this cost. Sender operation states are fully visible to the optimizer within a pipeline.

Reference lifetime hazard. Coroutine parameters are copied into the frame at the call site, but references are copied as references, not as values. A `const std::string&` parameter stores the reference in the frame. If the caller's string goes out of scope before the first suspension point, the reference dangles. Lambda captures by reference have the same hazard. This is a correctness cost, not a performance cost. Google built `Co<T>` as immovable and prvalue-only specifically to prevent it (P3801R0 [15] "Concerns about the design of `std::execution::task`"). Senders do not share this hazard in the same way - the operation state owns copies of everything passed through `connect`.

At the baseline, the price does not make coroutines slower. A benchmark [16] of 100,000,000 `read_some` calls on concrete streams measures both models at ~30-31 ns/op with zero allocations. The frame allocation is a cost that buys something. It is not a cost that slows you down.

You pay the price once. A coroutine that does fifty reads pays one frame allocation and fifty zero-cost resumptions. The frame that you cannot avoid is the same frame that holds the operation state, the local variables, and the result. The type erasure that blocks inlining is the same type erasure that gives you `any_stream`, `task<T>` with one parameter, and non-template operation states.

The frame you cannot avoid is the frame that pays for everything in Section 7.

---

5. Twenty Years, Nine Lines

The committee has been trying to standardize networking since N1925 [8] (2005). The contract that every attempt has been built on comes from Asio.

5.1 Asio's `AsyncReadStream`

The Boost.Asio documentation [17] defines `AsyncReadStream` as a named requirement with two operations:

operation	type	semantics
`a.get_executor()`	satisfying Executor requirements	Returns the associated I/O executor
`a.async_read_some(mb, t)`	determined by async operation requirements	completion signature `void(error_code ec, size_t n)`

Two operations. Twenty years. The contract has not changed. The Networking TS formalized it. The committee could not ship it. But the contract survived because it is correct.

5.2 The C++20 Concept

ReadStream [2] formalizes the same contract as a C++20 concept:

```
template<typename T>
concept ReadStream =
    requires(T& stream,
             mutable_buffer_archetype buffers)
    {
        { stream.read_some(buffers) }
            -> IoAwaitable;
        requires awaitable_decomposes_to<
            decltype(stream.read_some(buffers)),
            std::error_code, std::size_t>;
    };
```

`read_some` takes a buffer. The result satisfies `IoAwaitable`. The result decomposes to `(error_code, size_t)` via structured bindings.

Nine lines. Twenty years of contract, nine lines of concept.

The semantic requirements match Asio's: if `buffer_size(buffers) > 0` and `!ec`, then `n >= 1 && n <= buffer_size(buffers)` - at least one byte was read. If `ec`, then `n` is the number of bytes read before the I/O condition arose. I/O conditions are reported via the `error_code` component. Library failures (such as allocation failure) are reported via exceptions. The caller must ensure the buffer memory remains valid until the `co_await` expression returns.

5.3 What Changed

Two things vanished.

The completion token disappeared. In Asio, `async_read_some(mb, t)` takes a completion token `t` that determines the async model - callback, future, coroutine, or use_awaitable. In the coroutine model, the coroutine *is* the completion mechanism. There is no token. There is no choice. The coroutine suspends and resumes. The simplification is the trade-off.

`get_executor()` disappeared entirely. I/O objects do not carry executors. The caller provides the executor through `io_env` at `await_suspend` time. This resolves a long-standing Asio confusion where the I/O object has an executor and the completion token has a different executor, and the user must understand which one governs resumption. In the coroutine model there is one executor, it comes from the caller, and the I/O object uses it.

What stayed: `read_some` takes a buffer, returns `(error_code, size_t)`. The named requirement became a concept. The language caught up to the contract.

---

6. Why Did It Take Twenty Years?

Networking stalled because networking is not the hard problem. Asynchrony is the hard problem, and asynchrony has three domains: bulk-parallel execution, heterogeneous work-graph composition, and serial stream I/O. C++ already standardized the first two. The parallel algorithms in C++17 serve the bulk-parallel domain through execution policies. `std::execution` in C++26 serves the work-graph and heterogeneous domain through sender/receiver composition. The serial I/O domain - networking, files, pipes, TLS - has no standard facility. P0443R14 [9] "A Unified Executors Proposal for C++" spent fourteen revisions from 2016 to 2020 trying to unify all three domains into a single executor model, reconciling a work-executor suited to bulk dispatch with a continuation-executor suited to I/O chaining (the retrospective series P4094R0 [18] through P4099R0 [19] documents the full history). The unification imperative - the unexamined assumption that one model must cover all three domains - was the obstacle. The answer was three models, each contributing what the others cannot.

The domains that attracted institutional investment shipped. Bulk-parallel execution had compiler vendors. Heterogeneous dispatch had GPU manufacturers. Networking had one independent author without institutional backing. This is how volunteer-based standardization works when some volunteers are funded and some are not.

C++20 coroutines did not exist when the networking debate was at its peak. The Networking TS was designed for a language without `co_await`. By the time compilers shipped coroutines and production codebases validated them, the committee had committed to `std::execution` as the universal async model. The window where coroutine-native I/O was both technically possible and the parallel domain was already resolved did not open until 2025. The coroutine-native approach could not have been proposed earlier because its prerequisites had not converged.

Asio supports every async model - callbacks, futures, coroutines, and completion tokens. That universality served users but created a standardization obstacle: the I/O operations must be templates, the operation states must be parameterized on the completion handler type, and the stream model cannot be type-erased without per-operation allocation. The coroutine-native approach succeeds by committing to one completion mechanism and giving up the rest. Nobody could demonstrate what that commitment produces until someone made it. Section 7 is that demonstration.

The committee has heard networking proposals before. It would be reasonable to ask why this attempt should succeed where others have not. The answer is structural. `std::execution` shipping in C++26 resolves the parallel and heterogeneous domain, which means the serial I/O domain can now be addressed without reopening the parallel question. The Network Endeavor (P4100R0 [1]) is backed by a non-profit with engineering resources - not a solo effort by an independent author. Two libraries ship today (Capy [2] and Corosio [3]) with a Boost review scheduled. The team has implementation experience across the stream model - the author built Boost.Beast on Asio's architecture. Thirteen papers document the technical case, the retrospective record, and the bridge to `std::execution`. If coroutine I/O ships, C++ arrives at three companions - each contributing what the others cannot. P0443R0 tried to unify three into one. The answer, it turns out, was three.

---

7. What The Frame Buys

Five C++20 language mechanisms underlie the chain: the coroutine frame, type-erased `coroutine_handle<>`, the awaitable protocol (`await_ready`/`await_suspend`/`await_resume`), the `promise_type` customization point, and symmetric transfer (`await_suspend` returning `coroutine_handle<>`). None was designed for I/O. Combined, they produce a substrate that was not anticipated when C++20 shipped.

The commitment from Section 6 - coroutines as the only completion mechanism, giving up callbacks, futures, and completion tokens - unlocks properties that no other trade-off can deliver. The operation state becomes concrete. The stream becomes type-erasable. The library becomes separately compilable. The API becomes ABI-stable. None of these are possible when the completion mechanism is a template parameter.

The frame allocation is the cost every design avoids. This design accepts it. The frame the compiler already built is a type-erased storage location that the awaitable reuses at zero per-operation cost. A sender-based design can achieve the same downstream properties - type-erased operation states, pre-allocated storage, ABI-stable vtables - but pays a per-operation allocation because `connect(sender, receiver)` must heap-allocate the type-erased operation state each time the concrete type is unknown at compile time. The coroutine frame provides for free what the sender model must purchase per operation. Each consequence in the causal chain that follows is possible only because the previous one holds. Remove any link and the rest collapse. The chain is not a list of independent benefits. It is a single argument, nine steps long, and it begins with the frame.

7.1 The Caller Is Erased

Awaitable	Sender

`struct read_awaitable`
`{`
`  bool await_ready();`
`  void await_suspend(`
`  std::coroutine_handle<> h);`
`  // caller erased`
`  io_result<size_t>`
`  await_resume();`
`};`
	`template<class Receiver>`
	`struct read_operation`
	`{`
	`  Receiver rcvr_;`
	`  // caller stamped in`
	`  void start() noexcept;`
	`};`

Same operation. Left: the caller is erased behind `coroutine_handle<>`. Right: the caller's type is stamped into the operation state as `Receiver`. The `<>` is the fork in the road. Everything in 7.2 through 7.9 follows from this difference.

`std::function` erases a callable. `coroutine_handle<>` erases a resumable. One is a library convention. The other is a language primitive. Both hide the concrete type behind a fixed interface. The difference is that `std::function` must allocate its own storage. `coroutine_handle<>` points to a frame the compiler already built.

When a coroutine `co_await`s an awaitable, the awaitable's `await_suspend` receives `std::coroutine_handle<>`. That handle is type-erased. The awaitable does not know what the caller's return type is, what the caller's promise type is, what local variables the caller holds, or where the caller will resume. The caller's entire identity - its variables, its suspension point, its future - is behind an opaque pointer. The awaitable sees one thing: a handle it can resume or destroy.

Any function that returns an awaitable gets structural type erasure of the caller for free. The I/O operation's state does not depend on who is waiting for it. A `read_op` struct is the same whether the caller is `task<int>`, `task<void>`, or any other coroutine type. The caller is erased. The operation state is concrete.

The cost is real. Because the handle is opaque, the compiler cannot optimize through `resume()`. It cannot inline the caller into the awaitable or the awaitable into the caller. The optimization barrier from Section 4 is the same mechanism that provides the type erasure. You do not get one without the other.

In the sender model, `connect(sender, receiver)` stamps the receiver's type into the operation state. The optimizer sees the full pipeline - it can inline across operation boundaries, eliminate dead code, and propagate constants through the entire chain. That is the strength for GPU pipelines (Section 3). The cost of that visibility is that the I/O operation's state depends on who is waiting for it. The same `async_read` connected to two different receivers produces two different operation state types.

Senders and coroutines are two different ways to express computation. One contributes optimization visibility. The other contributes structural type erasure. The sender model's causal chain diverges here. The receiver-parameterized operation state gives the optimizer full pipeline visibility - the strength documented in Section 3. The chain that follows in 7.2 through 7.9 requires a concrete operation state. The sender model can achieve this by type-erasing the receiver, but each `connect` call must heap-allocate the operation state because the concrete type is unknown at compile time. The coroutine model achieves

it at zero per-operation cost - the frame the compiler already built is the storage the awaitable reuses. One unnecessary allocation per operation is one unnecessary allocation per operation, regardless of what else is in the I/O path.

From here forward, the paper walks the coroutine chain at zero per-operation cost. The sender model can follow the same chain at one allocation per operation. Section 8 documents the price.

7.2 The Operation State Is Concrete

Windows (IOCP). overlapped_op [3] and read_op [3]:

```
struct overlapped_op : OVERLAPPED
{
    std::coroutine_handle<> h;
    capy::executor_ref      ex;
    std::error_code*        ec_out;
    std::size_t*            bytes_out;
    DWORD                   bytes_transferred;
};

struct read_op : overlapped_op
{
    WSABUF wsabufs[16];
    DWORD  wsabuf_count;
    DWORD  flags;
    win_socket_internal& internal;
};
```

Linux (epoll). epoll_op.hpp [3]:

```
struct epoll_read_op final
    : reactor_read_op<epoll_op> {};

struct epoll_write_op final
    : reactor_write_op<
        epoll_op, epoll_write_policy> {};
```

Not templates. Not parameterized on the caller. Known at library-build time. The same `read_op` serves every coroutine that reads from the socket, regardless of the caller's type.

7.3 The Operation State Lives in the Socket

win_socket_internal [3]:

```
class win_socket_internal
{
    connect_op conn_;
    read_op    rd_;
    write_op   wr_;
    SOCKET     socket_ = INVALID_SOCKET;
    int        family_  = AF_UNSPEC;
};
```

Three operation states. Members of the socket. Pre-allocated when the socket is created. Not allocated per-operation. 10,000 sockets means 10,000 `read_op` instances of one known type, allocated once, reused for every read.

7.4 Zero Per-Operation Allocation

any_read_stream [2] type-erases any `ReadStream`. The erasure is on the awaitable, not the stream. The vtable [2] dispatches `await_ready`, `await_suspend`, `await_resume` through function pointers:

```
struct vtable
{
    void (*construct_awaitable)(
        void*, void*,
        std::span<mutable_buffer const>);
    bool (*await_ready)(void*);
    std::coroutine_handle<> (*await_suspend)(
        void*, std::coroutine_handle<>,
        io_env const*);
    io_result<std::size_t> (*await_resume)(void*);
    void (*destroy_awaitable)(void*) noexcept;
    std::size_t awaitable_size;
    std::size_t awaitable_align;
    void (*destroy)(void*) noexcept;
};
```

The awaitable storage is pre-allocated at construction time and reused for every `read_some` call. `read_some` constructs the real awaitable into cached storage, dispatches through the vtable, and destroys it on resume. One allocation at construction. Zero per-operation.

The benchmark [16] measures the cost. 100,000,000 `read_some` calls on a single thread, no-op stream, five runs per configuration:

Stream type	capy IoAwaitable		P2300 sender	
	ns/op	al/op	ns/op	al/op
Native	31.4	0	30.0	0
Abstract	32.1	0	53.5	1
Type-erased	36.4	0	53.4	1

At the native level, both models are equivalent. Under type erasure, awaitables add +5 ns and zero allocations. Senders add +23 ns and one allocation per operation. The cost is structural: `await_suspend` takes a type-erased `coroutine_handle<>`, so the awaitable's size is known at construction time. `connect(receiver)` produces an operation state whose type depends on both the sender and the receiver. When either side is type-erased, the operation state must be heap-allocated per operation.

For any individual I/O operation, the +23 ns dispatch difference is negligible - a `read()` syscall costs 1-10 us, and network round-trips cost orders of magnitude more. The difference is not speed per operation but infrastructure burden. To eliminate the per-operation allocation, senders need per-connection pools sized for the operation state, threaded through the API, and managed across 10,000 concurrent connections. Coroutines need nothing - the frame the compiler already built is the storage the awaitable already uses. At scale, the allocation compounds: 10,000 connections at 100 operations per second is 1,000,000 allocations per second that must be pooled or accepted.

7.5 Compile Once

Because `any_read_stream` has a fixed layout, a function accepting `any_read_stream&` goes in a `.cpp` file:

```
// dump.hpp
#include <boost/capy/task.hpp>
#include <boost/capy/io/any_read_stream.hpp>

capy::task<> dump(capy::any_read_stream& in);
```

```cpp
// dump.cpp
#include "dump.hpp"
#include <boost/capy/buffers.hpp>
#include <iostream>

capy::task<> dump(capy::any_read_stream& in)
{
    char buf[1024];
    for(;;)
    {
        auto [ec, n] = co_await in.read_some(
            capy::mutable_buffer(buf, sizeof buf));
        if(ec)
            break;
        std::cout.write(buf, n);
    }
}
```

The header includes only Capy [2]. No platform headers. No Corosio [3]. No sockets. The `.cpp` compiles once. Consumers include the header and link. The stream behind `any_read_stream` could be a TCP socket, a TLS session, a file, or a test mock. Nothing recompiles.

This is also the Capy [2]/Corosio [3] split point. Capy delivers the abstract layer: `task<T>`, `any_read_stream`, `any_write_stream`, `any_stream`, buffer concepts, stream concepts, `when_all`, `when_any`, the frame allocator. Pure C++20. No platform dependency. Corosio delivers the platform layer: `tcp_socket`, `tls_stream`, timers, DNS, signals. Capy delivers value on its own - sans-I/O protocols, buffered streams, test mocks, all compile against Capy without Corosio. The architecture that made separate compilation possible is the same architecture that made the library split possible. One frame. Two libraries. Zero recompilation.

7.6 Both Modes, One Abstraction

`co_await` checks `await_ready()` first. If the awaitable returns `true`, no suspension happens - the coroutine continues synchronously. A memory buffer, a test mock, a zlib decompressor, a base64 decoder - they all satisfy `ReadStream` by returning immediately-ready awaitables. The same `dump` function from Section 7.5 works whether the stream suspends for kernel I/O or returns instantly from a buffer. The algorithm does not know the difference.

A pipeline of `tcp_socket` -> `tls_stream` -> `decompression_stream` -> HTTP parser works regardless of which layers suspend. The synchronous layers pay zero suspension cost. No `is_async` flag. No separate sync API. One abstraction.

7.7 Forty Years of ABI

The vtable layout of `any_read_stream` does not change. Libraries compiled today work with new transports tomorrow. A `tls_stream` implementation compiled against OpenSSL 3.0 satisfies `ReadStream`. A future implementation compiled against a post-quantum TLS library will satisfy the same concept, plug into the same `any_read_stream`, and work with every library that was compiled against the old transport. No recompilation. No relinking. The forty-year-old contract - `read_some` takes a buffer, returns `(error_code, size_t)` - is the ABI.

ABI commitments are permanent. `std::regex` is slow because its ABI cannot change. The C++11 `std::string` ABI break was traumatic. The question is whether this vtable will need to change.

The contract behind it:

```
read_some(buffer) -> (error_code, size_t)
```

This is the POSIX `read()` contract with a C++ interface. Six change vectors exist. None breaks it.

- **New buffer types.** The `ReadStream` concept is already generic over buffer type. New buffer representations satisfy the existing concept. The vtable is unaffected.
- **New error conditions.** `std::error_code` is extensible via error categories. QUIC errors, post-quantum TLS errors, io_uring-specific errors - each registers a new category. The return type does not change.
- **New return values.** A read operation takes a buffer and reports how many bytes were transferred and whether an error occurred. No I/O model in any language returns anything else.
- **Cancellation.** Stop tokens propagate through `io_env`, not through the return type. Cancellation does not touch the contract.
- **New I/O patterns.** `read_some` is the primitive. Scatter/gather, vectored I/O, and read-until are composed from `read_some` by algorithms. The primitive does not change because the compositions do.
- **Environment evolution.** `io_env const*` is part of the `await_suspend` signature. If the environment needs new capabilities - priority hints, NUMA affinity, allocator awareness - the type must evolve. The mitigation is that `io_env` is accessed through pointer indirection: new fields append to the structure without changing the vtable's function signatures. The ABI risk is in `io_env`'s internal layout, not the vtable. This is the same evolution model as `OVERLAPPED` on Windows and `iocb` on Linux - pointer-to-extensible-struct.

The contract has survived POSIX, BSD sockets, IOCP, Asio, the Networking TS, io_uring, and every transport from TCP to QUIC.

7.8 The User Chooses

The user chooses the trade-off. io_stream [3], tcp_socket [3], native_tcp_socket [3]:

```
io_stream                       // abstract (Layer 3)
    |
tcp_socket                      // concrete (Layer 2)
    |
native_tcp_socket<Backend>      // native   (Layer 1)
```

Abstract (`io_stream`): virtual dispatch, ABI-stable, separately compiled. The compilation firewall. Business logic accepts `any_stream&` and never sees a platform header. Maximum compilation speed. Maximum insulation. The cost is virtual dispatch per I/O operation - nanoseconds against a microsecond syscall.

Concrete (`tcp_socket`): protocol-specific API - bind, connect, shutdown - still virtual dispatch, still separately compiled. Application code lives here. The full socket API without platform headers in the caller's translation unit.

Native (`native_tcp_socket<Backend>`): templated on the platform backend. Member function shadowing eliminates the vtable. Full inlining. Zero overhead. The cost is that the platform backend header is included and the code is in a header. Hot paths and benchmarks live here.

The user does not choose once for the whole application. Different layers coexist. A library accepts `any_stream&` (abstract). An application creates `tcp_socket` (concrete). A benchmark uses `native_tcp_socket<epoll>` (native). All three interoperate through the inheritance chain. The user gets the best of everything - compilation firewall where they want insulation, zero overhead where they want performance.

7.9 The Frame Subsidizes Everything

The coroutine frame paid for in Section 4 holds the local variables, the suspension point, and the result. `any_read_stream` works without per-operation allocation because the caller's frame already exists. The frame allocation you cannot avoid subsidizes the type erasure you want. This is the payoff for the price.

Per-operation allocations by execution model and stream type:

Stream type	`capy::task`	`beman::task`	sender pipeline
Native	0	0	0
Abstract	0	1	1
Type-erased	0	1	1

The frame you already paid for is the reason the other two rows are zero.

Additional properties that ride on the same frame:

- **Compile-time domain gate.** The two-argument `await_suspend(coroutine_handle<>, io_env const*)` is a deliberate trade-off. The alternative was a single-argument `await_suspend` that extracts the environment from the promise, costing one fewer parameter. The two-argument form was chosen because it buys compile-time enforcement: any awaitable that does not accept `io_env const*` is a type error inside an I/O task. Foreign awaitables that do not speak the I/O protocol are rejected by the compiler, not by a runtime mismatch. The pointer is the cost. The domain gate is the benefit. IoAwaitable [2].
- **Compound result preservation.** `auto [ec, n] = co_await sock.read_some(buf)`. Both values visible. No channel split. No data loss. The three-channel model routes results by channel. Compound results must choose a channel, losing data on the error path (P4090R0 [20], P4091R0 [21]).
- **Symmetric transfer.** `await_suspend` returns `coroutine_handle<>`. O(1) stack depth regardless of chain length.
- **One-parameter** `task<T>`. task.hpp [2]: `template<typename T = void> struct task`. One parameter. No Environment. The promise carries the environment. P4089R0 [22] documents why coroutine task type diversity is both inevitable and desirable.
- **Structured concurrency.** when_all [2] and when_any [2]. Both return `task<>`. Stop tokens propagate through `io_env`. Children complete before the parent resumes.

---

8. One Allocation Per Operation

Senders can achieve type-erased I/O. They can get ABI stability. The benchmark in Section 7.4 documents the cost: one allocation per operation, +23 ns. One allocation per read. Completely reasonable.

A custom allocator can pool the operation states. The pool must know each operation state's size at construction time - the size depends on both the sender and the receiver, so the pool is parameterized on the pipeline shape. The pool must be threaded through the API - the I/O object, the connect call, or the execution context must carry it. The pool must be managed per-connection - 10,000 connections means 10,000 pools, each sized for the operation states that connection's pipeline produces.

Small buffer optimization (SBO) is another mitigation. SBO works for `std::function` because callable objects are often small - a few pointers. Sender operation states are not small: the state includes captured data from the sender, the receiver's continuation and environment, and any intermediate storage. Under type erasure, the size depends on both the sender and receiver types, neither of which is known at compile time. A fixed SBO buffer must be sized for the worst case or fall back to heap allocation. The coroutine frame avoids this problem entirely - the compiler sizes it at coroutine-creation time, once, and the awaitables reuse storage the frame already provides.

The allocation purchased something. The receiver-parameterized operation state gives the optimizer full pipeline visibility - the strength documented in Section 3. That visibility is what makes senders the right choice for GPU dispatch, HPC, and compile-time work graphs. The allocation is the price of stamping the receiver into the operation state when the receiver is type-erased.

The coroutine model provides zero-allocation type erasure as a language consequence. The frame the compiler already built is the storage the awaitable already uses. No pool. No size calculation. No API threading. No per-connection management. The language feature provides what the workaround stack reconstructs.

---

9. Anticipated Objections

Q: Why not type-erase senders with `any_sender`?

A: `any_sender` type-erases the sender, not the receiver. `connect(any_sender, receiver)` still stamps the receiver type into the operation state. The operation state remains a template. `any_sender` erases the sender's identity from the caller. The coroutine model erases the caller's identity from the I/O operation. Section 7.1 documents the distinction. Section 8 documents the cost.

Q: Does `std::execution::task` not already bridge both models?

A: It does. `task` is a coroutine that is also a sender. But serving both models in one type is where the friction originates - two template parameters, open issues documented in P4007R3 [23], constraints that neither model alone requires. The companion approach accepts the design fork: each model does what it does best, and bridges connect them at ~10-14 ns with zero allocations. The question is whether the coroutine side carries I/O facilities that exploit the properties `coroutine_handle<>` provides: concrete operation states, type-erased streams, separate compilation, ABI stability. `task` bridges the models. It does not provide I/O.

Q: Two async models are harder to teach.

A: C++ already teaches three execution models: parallel algorithms with execution policies, `std::execution` with sender algorithms, and coroutines with `co_await`. These three complement each other in C++26. The teachability cost was paid when the committee shipped all three. Coroutine-native I/O does not add a fourth model. It completes the third. The programmer already writes `co_await` - coroutines are the code they already know, with three keywords added. The sender model requires thirty algorithms, each replacing a language construct the programmer already knows. P4014R2 [4] is a progressive tutorial of all thirty. The `retry` algorithm: approximately 120 lines as a sender, seven as a coroutine. The four-layer composition in P4014R2 [4] Section 12 collapses from interleaved sender pipelines into a single coroutine.

The teachability gap extends beyond the committee. On the `stdexec` issue tracker, a user reported [28] that `let_error([](int) { ... })` does not compile when the upstream sender can also complete with `std::exception_ptr`. The response: *"This is the design of stdexec."* The user replied: *"The obvious solution was just use `let_error([](int) ...`. But it's not working because of 'This is the design'."* Another user summarized the sentiment: *"Is it fundamentally impossible? May `let_*` senders be implemented that way in future?"* The answer was no.

Jonathan Müller wrote [29]: *"One particular complexity I don't like is the idea of environments... If we compose two senders from different libraries in a third library using the environment from a fourth place, it can require a lot of digging around to figure out what exactly went wrong."*

One anonymous r/cpp commenter reported [30]: *"I seriously investigated [libunifex] for roughly 2 work-days. I didn't like it, and couldn't use it to do what I wanted. That doesn't mean that libunifex couldn't - it means that I couldn't. The cognitive load was too high."*

Independent production confirmation: the derivatives exchange described in P4125R1 [31] - a team with a mature message-passing pipeline architecture - reported that the sender/receiver expression syntax caused many mental "trips" when reasoning about behaviour, and did not scale beyond simple examples (Section 7.2).

Q: Coroutines were not designed for I/O.

A: Correct. The five properties were designed for generality - async patterns, lazy evaluation, generators. Section 7 documents what they produce when applied to I/O. The substrate is emergent.

Q: The fragmentation argument is revisionist history.

A: The chronology is a fact. C++20 ratified 2020. `std::execution` adopted July 2024 [7]. The paper documents which model shipped first.

Q: `std::execution` is a library, not a second computation model.

A: `std::execution::task` (P3552R3 [5]) is a coroutine type. The library requires language coroutine support to function. Whether the second model is a "language model" or a "library model" does not change the chronology or the benchmark data.

Q: This picks the coroutine model as the winner for I/O.

A: The `IoAwaitable` concept (P4003R3 [24]) requires one function: `await_suspend(coroutine_handle<>, io_env const*)`. The protocol is two concepts, one type-erased executor, and a frame allocator cache. The specification fits in six pages. It says nothing about sockets, streams, TCP, TLS, or HTTP. Remove `await_ready` and you lose the synchronous fast path. Remove `io_env` and you lose the executor and stop token. Nothing is removable. This is the narrowest enabler. What gets built on top is the committee's decision.

Q: The domain split is artificial. Senders compose across domains.

A: C++26 ships three complementary execution models. P2500R2 [10] "C++ parallel algorithms and P2300" has not been revised since October 2023, was never adopted, and leaves the customization mechanism unspecified. `std::sort(std::execution::par, first, last)` does not use senders and no paper proposes that it should. The design fork (Section 10) is structural: `coroutine_handle<>` erases the caller, `connect(sender, receiver)` stamps the caller into the operation state, and the resulting property sets are mutually exclusive at zero per-operation cost. The bridges (P4092R0 [25], P4093R0 [26]) are how the companions connect - the same role `extern "C"` serves between C and C++. The complementary specializations follow from the design fork the committee already made.

Q: If the domains are separate, why do you need bridges?

A: Different computation models must interact at boundaries. C and C++ interact through `extern "C"`. CPU and GPU interact through memory copies. The bridges (P4092R0 [25], P4093R0 [26]) are how sender composition and coroutine I/O connect.

They are evidence of a clean interface between models. The crossing has a cost, and it is the right tradeoff. Each model becomes first class.

Q: The prerequisite has not shipped - `task` is still being iterated.

A: `std::execution` shipped: the scheduler abstraction, the sender composition algebra, the three-channel completion model, and the structured concurrency guarantees are in the C++26 working draft. `task`'s open issues are classified in P4007R3 [23] Section 2 as fixable post-ship - the `promise_type` is a class template instantiated in user code, so its allocation strategy, error handling, and environment forwarding can change between standard revisions without binary incompatibility. The prerequisite is the execution vocabulary, not one coroutine task type.

Q: P2300 was designed to serve I/O too. Its motivating example is a TCP server.

A: The motivating example demonstrates the gap. P2300R10 [27] "`std::execution`" Section 1.4.1.3 is analyzed in P4007R3 [23] Section 3 and P4090R0 [20]. The echo server loses the byte count on the error path. The I/O types cannot be type-erased without per-operation allocation. The stream cannot be separately compiled. The design intent included I/O. The design properties serve parallel and heterogeneous dispatch. This paper documents the difference between intent and outcome.

Q: io_uring's batch submission model favors senders.

A: The coroutine model does not submit one syscall per `co_await`. The event loop batches submissions from multiple suspended coroutines between `io_uring_enter` calls. The coroutine suspends into the submission queue; the event loop flushes the queue in bulk when it re-enters the kernel. Batching is an event loop implementation detail, invisible to the coroutine. Corosio's epoll and IOCP backends already batch this way. An io_uring backend would follow the same structure.

Q: This is too much scope for C++29.

A: P4003R3 [24] is two concepts, one executor, and a frame allocator cache. Six pages. The Network Endeavor (P4100R0 [1]) is modular - each paper stands independently. If one paper ships, that is progress. If four ship, that is a foundation. The team's target is pencils down by December 2028: reviewed, stable proposals with implementation experience.

---

10. The Design Fork

Six objections to the coroutine-native model are well-known. We concede every one.

1. Every coroutine that suspends pays a heap allocation. Senders do not.
2. `coroutine_handle<>::resume()` is an optimization barrier. Senders give the optimizer full pipeline visibility.
3. References captured in the coroutine frame can dangle. Sender operation states own copies.
4. Coroutines do not provide compile-time work graphs, static completion signature checking, or heterogeneous child composition.
5. Coroutines do not provide the generic sender composition algebra - `retry`, `upon_error`, `let_value` as reusable channel-routing adapters.
6. Two async models in the standard library are harder to teach and maintain than one.

These are real costs. Each model provides something in return. The fork is one design choice:

```
// Awaitable                    // Sender
void await_suspend(             template<class Receiver>
    coroutine_handle<> h);      struct read_operation {
    // caller erased                Receiver rcvr_;
                                    // caller stamped in
                                };
```

`coroutine_handle<>` erases the caller. `connect(sender, receiver)` stamps the caller into the operation state. Each choice unlocks a different set of strengths:

Senders	Coroutines
Full pipeline visibility to optimizer	Type-erased streams at zero per-op cost
Zero-allocation composition	Concrete, non-template operation states
Compile-time work graphs	Operation state lives in the I/O object
Static completion signature checking	Separate compilation of I/O algorithms
Heterogeneous child composition	ABI stability across transport changes
Generic composition algebra	Compound result preservation (ec + n)

Both companions provide structured concurrency (`when_all`, `when_any`), stop token propagation, non-exception error channels, and production deployment at scale. The bridge crossing cost is ~10-14 ns with zero allocations (P4092R0 [25], P4093R0 [26]).

At zero per-operation cost, the two property sets are mutually exclusive consequences of the design fork. Senders can achieve type erasure by heap-allocating the operation state. Coroutines can achieve partial pipeline visibility when HALO fires. Neither achieves both property sets without cost. That is the structural constraint. The sender strengths cluster around parallel and heterogeneous dispatch - compile-time work graphs, full pipeline optimization, and zero-allocation composition. The coroutine strengths cluster around serial stream I/O - type erasure, separate compilation, and ABI stability. The complementary specializations are not imposed by this paper. They are a consequence of the design fork and the standard the committee already shipped.

---

11. Conclusion

Networking stalled for twenty-one years because every proposal had to solve all of asynchrony at once. The parallel and heterogeneous domains now have standard facilities. The serial I/O domain does not.

The committee designed five language mechanisms for generality: the coroutine frame, type-erased `coroutine_handle<>`, the awaitable protocol, `promise_type`, and symmetric transfer. None was designed for I/O. Combined, they produce a substrate that resolves the problems that have kept networking out of the standard: template explosion, compile-time cost, allocation control, and ABI instability. The causal chain runs from one frame allocation through concrete operation states, type-erased streams, separate compilation, and ABI stability to a three-layer architecture where the user chooses the trade-off.

C++20 shipped the language. The programmer already knows how to use it. This paper documents the library it produces.

Acknowledgments

The author thanks Chris Kohlhoff for Asio's stream model, buffer sequences, and executor architecture - twenty years of production deployment is the foundation this work builds on; Eric Niebler, Kirk Shoop, Lewis Baker, and their collaborators for `std::execution`; Gor Nishanov for the coroutine model's explicit support for task type diversity; Dietmar Kühl for `beman::execution` and P3552R3; Ian Petersen for identifying an asymmetry in an earlier draft and for confirming the equivalence between sender and coroutine dispatch; Ville Voutilainen for broadening the compound-result problem beyond I/O and for P2464R0; Jens Maurer for framing the design spectrum; Herb Sutter for identifying the need for tutorials and documentation; Jonathan Müller for confirming the symmetric transfer gap in P3801R0; Peter Dimov for the refined channel mapping; Klemens Morgenstern for Boost.Cobalt and the cross-library bridges; Steve Gerbino for co-developing the constructed comparison, bridge implementations, and Corosio; and Mungo Gill, Mohammad Nejati, and Michael Vandeberg for feedback.

References

[1] P4100 - "The Network Endeavor: Coroutine-Native I/O for C++29" (Vinnie Falco, Steve Gerbino, Michael Vandeberg, Mungo Gill, Mohammad Nejati, 2026).

[2] cppalliance/capy - Coroutine I/O primitives library.

[3] cppalliance/corosio - Coroutine-native networking library.

[4] P4014R2 - "The Sender Sub-Language For Beginners" (Vinnie Falco, Mungo Gill, 2026).

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

[6] P2453R0 - "2021 October Library Evolution Poll Outcomes" (Bryce Adelstein Lelbach, Fabio Fracassi, Ben Craig, 2022).

[7] Herb Sutter, "Trip report: Summer ISO C++ standards meeting (St Louis, MO, USA)," 2024

[8] N1925 - "Networking proposal for TR2 (rev. 1)" (Gerhard Wesp, 2005).

[9] P0443R14 - "A Unified Executors Proposal for C++" (Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, Michael Wong, 2020).

[10] P2500R2 - "C++ parallel algorithms and P2300" (Ruslan Arutyunyan, Alexey Kukanov, 2023).

[11] Eric Niebler, "Structured Concurrency," 2020

[12] Eric Niebler, "What are Senders Good For, Anyway?" 2024

[13] P2470R0 - "Slides for presentation of P2300R2" (Eric Niebler, 2021).

[14] P0981R0 - "Halo: coroutine Heap Allocation eLision Optimization" (Gor Nishanov, 2018).

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

[16] I/O Read Stream Benchmark

[17] Boost.Asio AsyncReadStream requirements

[18] P4094 - "Retrospective: The Unification of Executors and P0443" (Vinnie Falco, 2026).

[19] P4099 - "Twenty-One Years: The Arc of Networking in C++" (Vinnie Falco, 2026).

[20] P4090 - "Sender I/O: A Constructed Comparison" (Vinnie Falco, Steve Gerbino, 2026).

[21] P4091 - "Two Error Models" (Vinnie Falco, 2026).

[22] P4089 - "On the Diversity of Coroutine Task Types" (Vinnie Falco, 2026).

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

[24] P4003R3 - "A Minimal Coroutine Execution Model" (Vinnie Falco, Steve Gerbino, Mungo Gill, 2026).

[25] P4092 - "Consuming Senders from Coroutine-Native Code" (Vinnie Falco, Steve Gerbino, 2026).

[26] P4093 - "Producing Senders from Coroutine-Native Code" (Vinnie Falco, Steve Gerbino, 2026).

[27] P2300R10 - "`std::execution`" (Eric Niebler, Michał Dominiak, Lewis Baker, Kirk Shoop, Lucian Radu Teodorescu, Lee Howes, 2024).

[28] NVIDIA/stdexec issue #1564: type issue of let_error - @taooceros, @BartolomeyKant, July 2025 / March 2026.

[29] Trip Report: Summer ISO C++ Meeting in St. Louis, USA - Jonathan Müller, July 2024.

[30] r/cpp: C++ committee polling results for asynchronous programming - Oct 2021.

[31] P4125R1 - "Coroutine-Native I/O at a Derivatives Exchange" (Mungo Gill, 2026).