A Minimal Coroutine Execution Model

Document Number:	P4003R2
Date:	2026-04-17
Intent:	Ask
Audience:	LEWG
Reply-to:	Vinnie Falco vinnie.falco@gmail.com Steve Gerbino steve@gerbino.co Mungo Gill mungo.gill@me.com

Table of Contents

Abstract

This paper asks the committee to advance the *IoAwaitable* protocol as a standard coroutine execution model.

We start from the minimal use case:

```
co_await f();
```

For this to work, something must decide where the coroutine resumes, whether it should stop, and where its frame is allocated. The *IoAwaitable* protocol provides exactly those three things - executor affinity, stop token propagation, and frame allocator delivery - and nothing more.

A companion paper, P4172R0[1], provides the design rationale, evidence framework, preemptive objections, and analysis of alternative approaches.

Everything in this paper comes from a complete implementation on three platforms: Capy[2] (protocol) and Corosio[3]. A self-contained demonstration is available on Compiler Explorer[4].

Revision History

R2: April 2026 (post-Croydon mailing)

- Title changed from "Coroutines for I/O" to "A Minimal Coroutine Execution Model".
- Reframed as a coroutine execution model. Networking is one consumer, not the identity.
- Motivating example changed from `socket.read_some(buf)` to `co_await f()`.
- Introduction replaced with Disclosure. Paper repositioned within the Network Endeavor.
- Sections 2-7 (Networking's Essentials, The Protocol, Executor concept, The Frame Allocator, Ergonomics of Type Erasure, io_awaitable_promise_base Mixin) replaced with new Sections 2-6 (What We Get, What Coroutines Need, The IoAwaitable Protocol, IoAwaitable Is Structured Concurrency, Why Not `exec::as_awaitable`?).
- Evidence Framework (Section 9), example wording (Section 11), and Appendices A-B removed. Design choices, rationale, and post-adoption retrospectives moved to companion paper P4172R0[1].
- Networking quotes, Kona poll context, and SG14 position moved to P4100R0[5].
- Five straw polls replaced with one.
- References pruned and renumbered.

R1: March 2026 (pre-Croydon mailing)

- Added "Why Standardize" subsection to Section 1: committee record, tower of abstraction argument, P4133 cost/benefit analysis.
- Expanded implementation evidence in Section 3.

- Added Section 9 "Evidence Framework" addressing P4133 requirements: competing designs, case against standardization, decision record, domain coverage, post-adoption metrics, retrospective commitment, prediction registry.
- Expanded sequence diagram with explicit `set_environment`, `set_continuation`, and `handle.resume()` steps.
- Renamed "Boost.Http" to "Http" in body and references.
- Closed TLS spoilage gap in Section 5.4: intervening code between resume and child creation can overwrite the thread-local frame allocator. Introduced `safe_resume` save/restore protocol.
- Added non-normative note to executor concept (Section 11.3.3) requiring event loop pump sites to save and restore TLS around `.resume()` calls.
- Replaced `std::coroutine_handle<>` with `continuation` in the executor interface. The `continuation` struct embeds an intrusive list pointer, eliminating per-post heap allocation.
- Editorial: fixed list formatting, code line wrapping, acknowledgements.

R0: March 2026 (pre-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)[5], a project to bring coroutine-native I/O to C++.

Falco and Gerbino developed and maintain Capy[2] and Corosio[3] and believe 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.

---

2. What We Get

If only this proposal ships and nothing else, we get:

What `std` provides	What users can write
IoAwaitable	Interoperable tasks, awaitables
IoRunnable	Interoperable launch functions
Executor	Interoperable executors
ExecutionContext	User-defined execution contexts
`execution_context`	Platform event loops
`get_cached_frame_allocator`, `set_cached_frame_allocator`	Custom frame allocators
`executor_ref`	(protocol)
`io_env`	(protocol)

The left column is small. The right column is not. Small protocol, big rewards. It earns its keep.

This protocol is a companion to P2300R10[6] `std::execution`. See P4172R0[1] for design rationale and analysis of alternative approaches.

---

3. What Coroutines Need

What follows is the minimum.

3.1 How to Suspend

The minimal statement that suspends a coroutine:

```
co_await f();
```

When this executes, the coroutine suspends and control passes to the awaitable returned by `f()`. That awaitable holds the coroutine handle and must eventually resume it. But on which thread? Under whose control?

```
std::coroutine_handle<> h = /* ...the suspended coroutine... */;

h.resume();  // but WHERE? on which thread? under whose control?
```

This is the question that drives the entire protocol. The awaitable cannot just call `h.resume()` - that resumes on the current thread, possibly while holding a lock, possibly re-entering code that is not re-entrant. Something must decide where and how the coroutine wakes up.

Every awaitable needs three things at the moment of suspension: who resumes me (executor), should I stop (stop token), and where do child frames come from (frame allocator).

The I/O payoff is immediate. Consider the awaitable for a platform operation:

```
auto [ec, n] = co_await stream.read_some(buf);
```

The same three concerns apply: the reactor completes the read and holds a coroutine handle that must resume on the right thread. Synchronous awaitables return `true` from `await_ready` and never suspend at all. The protocol handles both.

3.2 How to Resume

The reactor has a coroutine handle. It cannot just call `resume()`. Something decides where and how the coroutine wakes up. That something is the executor:

```
executor ex = /* ...the coroutine's executor... */;

ex.post( h ); // (notional)
```

The executor queues the coroutine for resumption under the application's control. That is its entire job. The full *Executor* concept is presented in Section 4.

3.3 How to Stop

The stop mechanism should be invisible until you need it. Most coroutines never touch it. But when you need it, it should be obvious:

```
auto token = co_await get_stop_token;

if (token.stop_requested())
    co_return;
```

One line to get the token. One check. Built on `std::stop_token`.

3.4 How to Launch

A regular function cannot `co_await` (see P4035R0[7] for a discussion of coroutine escape hatches). To start a coroutine chain, you call a launch function:

```
run_async( ex )( my_coroutine() );
```

The executor is required. The stop token and frame allocator are optional. This is where the three concerns from 3.1-3.3 come together - the launch function binds them to the coroutine chain.

3.5 How to Allocate

Every coroutine has a frame, and every frame must be allocated. This is why coroutines look slow. Despite the cost, the right frame allocator makes coroutines performant. Thus, the frame allocator must be a first-class citizen.

Frame allocators come in two versions:

1. A classic typed allocator (e.g. `std::allocator`, a custom pool allocator) - the user's own type
2. A `std::pmr::memory_resource*` - type-erased, for when the concrete type does not matter

Platform	Frame Allocator	Time (ms)	Speedup
MSVC	Recycling	1265.2	3.10x
MSVC	mimalloc	1622.2	2.42x
MSVC	`std::allocator`	3926.9	-
Apple clang	Recycling	2297.08	1.55x
Apple clang	`std::allocator`	3565.49	-

The protocol must:

- Provide a reasonable, customizable default
- Propagate the frame allocator to every coroutine frame in the chain automatically
- Keep function signatures clean, unless the programmer needs otherwise
- Allow a coroutine to `co_await` a new chain with a different frame allocator

---

4. The *IoAwaitable* Protocol

What follows is the minimum as well.

4.1 `io_env`

The `io_env` struct contains the three members a coroutine needs: the executor, the stop token, and the frame allocator:

```
struct io_env
{
    executor_ref executor;
    std::stop_token stop_token;
    std::pmr::memory_resource* frame_allocator = nullptr;
};
```

4.2 *IoAwaitable*

Implementations and library authors provide types satisfying *IoAwaitable*:

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

The two-argument `await_suspend` is the mechanism. The caller's `await_transform` injects the environment as a pointer parameter - no templates, no type leakage. A `task` needs only one template parameter. The environment is passed as a pointer because the launch function owns the `io_env` and every coroutine in the chain borrows it - pointer semantics make the ownership model explicit.

The two-argument signature is also a compile-time boundary check. A non-compliant awaitable fails to compile. A compliant awaitable in a non-compliant coroutine fails to compile. Both sides of every suspension point are statically verified. In a world

with multiple coexisting async models, a coroutine that accidentally `co_await`s across model boundaries should fail at compile time, not silently misbehave at runtime. See P4172R0[1] for the alternative design discussion and detailed analysis.

4.3 *Executor*

```
struct continuation
{
    std::coroutine_handle<> h;
    continuation* next = nullptr;
};

template<class E>
concept Executor =
    std::is_nothrow_copy_constructible_v<E> &&
    std::is_nothrow_move_constructible_v<E> &&
    requires( E& e, E const& ce, E const& ce2,
              continuation& c ) {
        { ce == ce2 } noexcept -> std::convertible_to<bool>;
        { ce.context() } noexcept;
        requires std::is_lvalue_reference_v<
            decltype(ce.context())> &&
            std::derived_from<
                std::remove_reference_t<
                    decltype(ce.context())>,
                execution_context>;
        { ce.on_work_started() } noexcept;
        { ce.on_work_finished() } noexcept;
        { ce.dispatch( c ) }
            -> std::same_as< std::coroutine_handle<> >;
        { ce.post(c) };
    };

class executor_ref
{
    void const* ex_ = nullptr;
    detail::executor_vtable const* vt_ = nullptr;

public:
    template<Executor E>
    executor_ref(E const& e) noexcept
        : ex_(&e), vt_(&detail::vtable_for<E>) {}

    std::coroutine_handle<> dispatch(continuation& c) const;
    void post(continuation& c) const;
    execution_context& context() const noexcept;
};
```

The `continuation` struct pairs a `coroutine_handle<>` with an intrusive `next` pointer, allowing executors to queue continuations without allocating a separate node - eliminating the last steady-state allocation in the hot path. `dispatch` returns a `coroutine_handle<>` for symmetric transfer: if the caller is already in the executor's context, it returns `c.h` directly for zero-overhead resumption. Otherwise it queues and returns `noop_coroutine()`. `post` always defers. The `executor_ref` type-erases any *Executor* as two pointers - one indirection (~1-2 nanoseconds[8]) is negligible for I/O operations at 10,000+ nanoseconds. See P4172R0[1] for detailed semantics.

4.4 execution_context

```
class execution_context
{
public:
    class service
    {
    public:
        virtual ~service() = default;
    protected:
        service() = default;
        virtual void shutdown() = 0;
    };

    execution_context( execution_context const& ) = delete;
    execution_context& operator=(
        execution_context const& ) = delete;
    ~execution_context();
    execution_context();

    template<class T> T& use_service();
    template<class T, class... Args>
        T& make_service( Args&&... args );

    std::pmr::memory_resource*
        get_frame_allocator() const noexcept;
    void set_frame_allocator(
        std::pmr::memory_resource* mr ) noexcept;

protected:
    void shutdown() noexcept;
    void destroy() noexcept;
};

template<class X>
concept ExecutionContext =
    std::derived_from<X, execution_context> &&
    requires(X& x) {
        typename X::executor_type;
        requires Executor<typename X::executor_type>;
        { x.get_executor() } noexcept
            -> std::same_as<typename X::executor_type>;
    };
```

An executor's `context()` returns the `execution_context` - the base class for anything that runs work. The platform reactor lives here. Services provide singletons with ordered shutdown. I/O objects hold a reference to their execution context, not to an executor. This design borrows from Boost.Asio[9].

The execution context holds the default frame allocator. The user can optionally override it, and every coroutine chain launched through that context uses it. This is how the "reasonable, customizable default" from Section 3.5 works in practice.

4.5 Frame Allocator Delivery

There are exactly two ways to deliver the allocator to `operator new`:

1. **The parameter list.** This is `allocator_arg_t` - it always works and is always available as a fallback, but it should not be the only option.
2. **Out of band.** The allocator is temporarily stashed somewhere the `operator new` can find it.

The protocol specifies accessor functions but leaves the storage mechanism to the implementer:

```
std::pmr::memory_resource*
get_cached_frame_allocator() noexcept;

void
set_cached_frame_allocator(
    std::pmr::memory_resource* mr) noexcept;
```

See P4172R0[1] for the timing constraint analysis, execution window, `safe_resume` protocol, and responses to common concerns.

4.6 `IoRunnable` and Launch Functions

Someone has to launch the first coroutine. Within a coroutine chain, *IoAwaitable* alone is sufficient - `co_await` handles lifetime, result extraction, and exception propagation natively. But launch functions cannot `co_await`. They need access to the promise to manage lifetime and extract results. This paper does not propose launch functions - the ecosystem provides them. *IoRunnable* is the concept that supports them.

Launch functions come in two kinds:

From regular code: `run_async`. You cannot `co_await` in `main()`. This is the entry point - the place where synchronous code starts an asynchronous chain. The executor is required. The stop token and frame allocator are optional. Without handlers, the result is discarded and exceptions rethrow on the executor thread. With handlers, both outcomes are explicitly routed:

```
// Fire and forget
run_async( ex )( server_main() );

// Structured: both outcomes routed
run_async( ex,
    [](int result)         { /* use result */   },
    [](std::exception_ptr) { /* handle error */ }
)( compute() );
```

From a coroutine: `run`**.** Switches executor, stop token, or allocator for a subtask. Always `co_await`ed. The parent suspends and resumes only when the child completes - the lexical boundary is enforced by the language. There is no unstructured path through `run`:

```
co_await run( worker_ex )( compute() );
co_await run( source.get_token() )( sensitive_op() );
co_await run( pool )( alloc_heavy_op() );
co_await run( worker_ex, source.get_token(), pool )(
    compute() );
```

Both kinds use two-phase invocation to ensure the frame allocator is cached before the child coroutine's frame is allocated. *IoRunnable* provides the interface both need:

```
template<typename T>
concept IoRunnable =
    IoAwaitable<T> &&
    requires { typename T::promise_type; } &&
    requires( T& t, T const& ct,
              typename T::promise_type const& cp,
              typename T::promise_type& p )
    {
        { ct.handle() } noexcept
            -> std::same_as<
                std::coroutine_handle<
                    typename T::promise_type> >;
        { cp.exception() } noexcept
            -> std::same_as< std::exception_ptr >;
        { t.release() } noexcept;
        { p.set_continuation(
            std::coroutine_handle<>{} ) } noexcept;
        { p.set_environment(
            static_cast<io_env const*>(
                nullptr) ) } noexcept;
    } &&
    ( std::is_void_v<
        decltype(
            std::declval<T&>().await_resume()) > ||
      requires( typename T::promise_type& p ) {
          p.result();
      });
```

See P4172R0[1] for detailed examples and implementation guidance.

---

5. *IoAwaitable* Is Structured Concurrency

1. The awaitable owns the suspended coroutine handle.
2. The awaitable submits the operation and transfers ownership to the executor.
3. The executor resumes the coroutine, returning ownership to the coroutine body.

There is always an owner: the coroutine body, the awaitable, or the executor.

This ownership model is possible because the language provides the mechanism:

- `co_await` enforces a lexical boundary. The child completes before the parent continues.
- RAII works inside coroutines. Deterministic destruction is guaranteed.
- Cancellation propagates forward. Destruction propagates backward. Both are automatic.

- The language provides what a library would reimplement.
- The synchronous entry point requires an escape hatch in every async framework. Senders call theirs `sync_wait`.
- `when_all` and `when_any` in Capy[2] (`include/boost/capy/when_all.hpp`, `when_any.hpp`)

```
auto [ec, counts] = co_await when_all(std::move(reads));

auto result = co_await when_any(std::move(reads));
```

The *IoAwaitable* protocol is structured concurrency.

5.1 Structurable Building Blocks Are More Fundamental

When handlers are provided, the launch is structured - both outcomes are explicitly routed by the caller:

```
run_async( ex,
    [](int result)         { /* use result */   },
    [](std::exception_ptr) { /* handle error */ }
)( compute() );
```

Without handlers, the result is discarded and exceptions rethrow on the executor thread. This unstructured path is intentional - P4035R0[7] explains why launch functions cannot `co_await` and therefore need an escape hatch. The protocol provides structure when the caller uses it; it does not forbid escape when the caller needs it.

The handlers are the primitive. Higher-level structured concurrency abstractions are built from them. A `counting_scope` - tracking N concurrent tasks, joining when all complete, propagating exceptions - is a handler policy:

```
class counting_scope {
    /* atomic state machine: idle -> waiting -> done */

public:
    template<Executor Ex, IoRunnable Task>
    void spawn(Ex ex, Task t) {
        run_async(ex,
            [this](auto&&...) noexcept          { on_done(); },
            [this](std::exception_ptr e) noexcept { on_except(e); }
        )(std::move(t));
    }

    [[nodiscard]] /* awaitable */ join() noexcept;

private:
    void on_done() noexcept;  // decrements count, resumes
                              // waiter on zero
    void on_except(std::exception_ptr) noexcept; // stores first ep, then on_done()
};
```

A full implementation is available in Capy[2] (`include/boost/capy/ex/when_all.hpp`).

The protocol provides the building block. `counting_scope` is one policy built on it. The argument that senders provide structured concurrency and coroutines do not has it backwards: the *IoAwaitable* protocol is the layer from which structured concurrency constructs are assembled.

---

6. Why Not `exec::as_awaitable`?

`std::execution` provides `exec::as_awaitable`, which wraps a sender as an awaitable. Both models work. The table shows the cost differential under type erasure:

Property	Returns awaitable	Returns sender
Frame allocations	1	1
Per-operation allocation (under type erasure)	0 (preallocated awaitable)	1 (`op_state` heap-allocated per `connect`)
Inline completion (`await_ready`)	Yes - completes, no suspend	No - `start()` is post-suspend
Synchronous-complete overhead	0 (symmetric transfer)	`connect`/`start` + trampoline

Under type erasure, `connect(sndr, rcvr)` produces a type-dependent `op_state` that must be heap-allocated when either side is erased.

The sender composition algebra does not apply to compound results - such as `[ec, n]` - without data loss or shared state; the sender three-channel model is in tension with `error_code` as a value-channel result.[10,11]

See P4172R0[1] for detailed analysis.

P3482R1[12] ("Design for C++ networking based on IETF TAPS") defines a TAPS-shaped networking API surface. The *IoAwaitable* protocol provides the coroutine execution model beneath it. A TAPS implementation needs coroutines that suspend, resume correctly, cancel, and allocate frames - exactly what this protocol provides. The two are not competing; TAPS is a consumer.

---

7. Conclusion

Executor affinity, stop token propagation, frame allocator delivery. Three concerns. One protocol. A companion to `std::execution`, implemented on three platforms. Nothing in it can be removed.

---

8. Suggested Straw Poll

Poll. The *IoAwaitable* protocol is the minimum vocabulary for coroutines that need executor affinity, a stop token, and a frame allocator.

Acknowledgements

Gor Nishanov - The C++20 coroutines language feature is the foundation every word of this paper rests on. Without the coroutines TS, the *IoAwaitable* protocol has nothing to build on.

Christopher Kohlhoff - The `execution_context`, executor, and service model in Section 4 derives directly from Boost.Asio. The design choices in this paper are possible because Kohlhoff built them first and proved them in production.

Eric Niebler, Bryce Adelstein Lelbach, Michał Dominiak, Lewis Baker, Lee Howes, Kirk Shoop, Jeff Garland, Georgy Evtushenko, and Lucian Radu Teodorescu - The authors of P2300R10. The structured concurrency guarantees, completion channel model, and scheduler design in `std::execution` defined the async vocabulary this paper complements. The bridge papers in the Network Endeavor exist because P2300 is in the standard.

Lewis Baker - The symmetric transfer technique, documented in the published blog post, is the mechanism that makes the zero-overhead coroutine resumption path in Section 4.2 possible.

References

[1] P4172R0 - "Design: IoAwaitable for Coroutine-Native Byte-Oriented I/O" (Vinnie Falco, Steve Gerbino, Mungo Gill, 2026).

[2] Capy - *IoAwaitable* protocol implementation (Vinnie Falco, Steve Gerbino).

[3] Corosio - Coroutine-native I/O library (Vinnie Falco, Steve Gerbino).

[4] Compiler Explorer - Self-contained *IoAwaitable* demonstration.

[5] P4100R0 - "The Network Endeavor: Coroutine-Native I/O for C++29" (Vinnie Falco et al., 2026).

[6] P2300R10 - "`std::execution`" (Dominiak, Baker, Evtushenko, Teodorescu, Howes, Shoop, Garland, Niebler, Lelbach, 2024).

[7] P4035R0 - "Support: The Need for Escape Hatches" (Vinnie Falco, 2026).

[8] Optimizing Away C++ Virtual Functions May Be Pointless - CppCon 2023 (Shachar Shemesh).

[9] Boost.Asio - Asynchronous I/O library (Christopher Kohlhoff).

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

[11] P4091R0 - "Info: Error Models of Regular C++ and the Sender Sub-Language" (Vinnie Falco, 2026).

[12] P3482R1 - "Design for C++ networking based on IETF TAPS" (Thomas Rodgers, Dietmar Kühl, 2024).