The Coroutine Frame Allocator Timing Problem

Document Number:	P4127R0
Date:	2026-05-01
Intent:	Inform
Audience:	SG1, LEWG

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

Table of Contents

Abstract

The coroutine frame allocator must arrive before the coroutine frame exists. Exactly two mechanisms can deliver it.

C++20 coroutines allocate their frame in `promise_type::operator new`, which the compiler calls before the coroutine body executes. Any mechanism that delivers the frame allocator after the coroutine is invoked arrives too late. This paper enumerates every customization point the language provides at or before `operator new` and shows that each one either reduces to passing the allocator in the coroutine's parameter list, reduces to ambient state, or fires after the frame already exists. The design space is closed. The two solutions are `allocator_arg_t` (parameter passing) and thread-local propagation (ambient state). A conforming promise type can provide both as `operator new` overloads, letting the user choose per call site.

Revision History

R0: May 2026 (pre-Brno mailing)

- Initial version.
- Clarified scope: the timing problem applies to child coroutines, not to leaf I/O awaitables (Section 2). Thanks to Michael Hava for identifying the gap.
- Added hybrid option: both paths coexist in a single promise type via `operator new` overload resolution (Section 8).
- Added structured analysis of TLS concerns: platforms, performance, action at a distance, silent failure, precedent, and categorical rejection (Section 9).

---

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 Corosio [2] and Capy [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.

P4003R3 [4], "A Minimal Coroutine Execution Model," uses thread-local propagation for the frame allocator. This paper documents why that choice is one of exactly two possibilities, not a preference among many. The analysis stands independent of the author's library work.

This paper asks for nothing.

---

2. The Constraint

Not every awaitable has a frame. Leaf I/O operations - `sock.read_some(buf)`, timer waits, DNS lookups - return plain awaitable structs. No coroutine, no frame, no `operator new`. The parent's `await_transform` receives them, injects the execution environment, and no frame allocation is involved. The timing problem does not apply to these operations.

The timing problem applies to **child coroutines** - functions that contain `co_await`:

```
task<void> handle_request(socket& s)
{
    auto hdr = co_await parse_header(s);  // coroutine - has a frame
    co_await send_response(s, hdr);        // coroutine - has a frame
}
```

`parse_header` contains `co_await`. It has local variables that persist across suspension points. It is a coroutine. It has a frame. The call expression `parse_header(s)` allocates that frame before the parent's `await_transform` runs. There is nothing for `await_transform` to wrap - the task already exists.

To make `parse_header` a plain awaitable instead of a coroutine, one would have to hand-write the state machine the compiler generates. That is what coroutines exist to avoid.

When a coroutine is called, the compiler allocates the frame first, then begins execution:

```
task<int> fetch(socket& s, buffer& b);

auto t = fetch(sock, buf);    // operator new HERE
co_await t;                   // too late
```

`promise_type::operator new` receives the frame size and the coroutine's own parameters. It executes before the coroutine body, before `initial_suspend`, before any user code inside the coroutine runs. The frame allocator must be available at this point. Any mechanism that delivers context later - after the call expression returns, after `co_await` begins, after the coroutine body starts - cannot influence frame allocation.

The question is: what mechanisms can deliver a runtime value to `promise_type::operator new`?

---

3. The Enumeration

C++20 provides a finite set of customization points for coroutines. Each one fires at a specific point in the coroutine lifecycle. The following table places every customization point in time relative to `operator new`:

Customization point	When it fires	Before `operator new`?
`promise_type::operator new`	Frame allocation	Is `operator new`
`std::coroutine_traits`	Compile time (selects `promise_type`)	Compile time only
`initial_suspend`	After frame creation, before coroutine body	No
`await_transform`	When parent `co_await`s the child's return value	No
`operator co_await`	Same as `await_transform` (operand already evaluated)	No
`await_ready` / `await_suspend` / `await_resume`	After `await_transform`	No
`final_suspend`	After coroutine body completes	No
`return_value` / `return_void`	At `co_return`	No
`yield_value`	At `co_yield`	No
`unhandled_exception`	On exception	No

Every customization point except `operator new` itself fires after the frame exists. `coroutine_traits` operates at compile time - it selects which `promise_type` to use, but cannot inject a runtime value.

The compiler passes exactly one set of runtime values to `operator new`: the frame size and the coroutine's parameter list. No other runtime information is available at that point.

---

4. The Two Paths

4.1 Path A: The Parameter List

The coroutine's parameters are forwarded to `promise_type::operator new`. If the allocator appears in the parameter list, `operator new` can use it. This is the `allocator_arg_t` pattern:

```
task<int> fetch(std::allocator_arg_t,
                MyAllocator alloc,
                socket& s, buffer& b);
```

The promise provides overloads that detect the tag:

```
struct promise_type {
    template<typename Alloc, typename... Args>
    static void* operator new(std::size_t sz,
        std::allocator_arg_t, Alloc& a,
        Args&&...)
    {
        return a.allocate(sz);
    }

    template<typename T, typename Alloc,
             typename... Args>
    static void* operator new(std::size_t sz,
        T&, std::allocator_arg_t, Alloc& a,
        Args&&...)
    {
        return a.allocate(sz);
    }
};
```

This works. The allocator reaches `operator new` at the right time. The cost is that every coroutine in a call chain must carry the allocator in its parameter list. P4003R3 [4] Section 5.2 documents the ergonomic consequences.

4.2 Path B: Ambient State

If the allocator is not in the parameter list, `operator new` must read it from somewhere else. The only "somewhere else" available to a function at call time is ambient state: global, thread-local, or fiber-local storage.

Global state is too coarse - a single allocator for every coroutine chain in the process. Thread-local state scoped per-chain is the minimum viable mechanism. P4003R3 [4] Section 5.3 documents the thread-local write-through cache that delivers the allocator to `operator new` and restores it on every resume.

---

5. The Rejected Alternatives

Each alternative that appears to offer a third path reduces to Path A, Path B, or arrives too late.

5.1 `std::coroutine_traits`

`coroutine_traits<R, Args...>` selects the `promise_type` based on the coroutine's return type and parameter types. It operates at compile time. It can change *which* `operator new` runs. It cannot inject a runtime value that is not already in the parameter list.

A `coroutine_traits` specialization that detects an allocator parameter type and selects an allocator-aware promise is a compile-time optimization of Path A. The allocator must still appear in the parameter list. The signature pollution remains.

5.2 `await_transform`

The parent coroutine's `await_transform` fires when the parent evaluates `co_await child_expr`. The child expression is the operand. The operand is fully evaluated - including the child's `operator new` - before `await_transform` runs.

```
co_await fetch(sock, buf);
//       ^^^^^^^^^^^^^^^^
//       evaluated first: operator new fires here
// then await_transform runs on the returned task
```

`await_transform` can inject context into the child's `await_suspend`. It cannot influence the child's `operator new`. Too late.

5.3 `operator co_await`

`operator co_await` converts the operand into an awaiter. It runs after the operand is evaluated, at the same point as `await_transform`. The child's frame already exists. Too late.

5.4 `initial_suspend`

`initial_suspend` runs inside the child coroutine, after the frame is allocated. It can *capture* a value from TLS (as the IoAwaitable protocol does), but it cannot *deliver* a value to `operator new`. It is useful for Path B's capture step, not as an independent delivery mechanism.

5.5 Deferred Invocation

One could defer the child's construction so that the parent's `await_transform` invokes it:

```
co_await deferred([&]{ return fetch(sock, buf); });
```

The parent's `await_transform` would set the allocator, invoke the lambda, and forward to the child's `await_suspend`. This eliminates TLS in principle. In practice:

- Every call site requires a lambda wrapper. The ergonomic cost exceeds `allocator_arg_t`.
- The lambda captures references to the parent's locals. The child coroutine outlives the lambda. Use-after-free at every call site.
- The child's `operator new` still needs to read the allocator from somewhere. If not from the parameter list, then from ambient state. The lambda approach defers the invocation but does not eliminate the delivery problem - it moves it inside `await_transform`, where the same two paths apply.

Deferred invocation is Path A with worse ergonomics, or Path B with an extra indirection. It is not a third path.

5.6 Promise-Based Injection

The parent's promise could store the allocator and the child's `operator new` could read it from the parent's promise. But `operator new` does not receive the parent's promise. It receives the child's own parameters. The parent's promise is not accessible to the child's `operator new` through any standard mechanism.

One could pass the parent's promise as a parameter to the child coroutine. This is Path A again - the parameter list carries the context.

5.7 Global `pmr::get_default_resource()`

`std::pmr::get_default_resource()` is a process-wide ambient allocator. It is Path B with process scope instead of per-chain scope. A single allocator for every coroutine chain in the process is insufficient for deployments that need per-tenant budgets, bounded pools, or allocation tracking scoped to individual request chains.

5.8 Sender-Environment Bridge Wrappers

A wrapper object can read the allocator from the sender execution environment and inject `allocator_arg_t` into the coroutine's parameter list at the launch site. The wrapper intercepts the call, reads the allocator via `read_env(get_allocator)`, and invokes the coroutine with `std::allocator_arg, alloc, args...`. The coroutine's `operator new` sees the allocator at the right time.

This automates the first hop - from the sender world into the coroutine's parameter list. Inside the coroutine, every `co_await` of a child coroutine still requires manual forwarding:

```
task<void, env> handle_request(
    std::allocator_arg_t, auto alloc,
    socket& s)
{
    buffer buf;
    // wrapper helped at the launch site,
    // but here we are back to manual:
    auto hdr = co_await parse_header(
        std::allocator_arg, alloc, s, buf);
    co_await send_response(
        std::allocator_arg, alloc, s, hdr);
}
```

The wrapper is not present at internal call sites. It solves the sender-to-coroutine boundary but not the coroutine-to-coroutine chain. Every child coroutine still needs `allocator_arg_t` in its signature. Every internal `co_await` still forwards the allocator manually.

A sender-environment bridge wrapper is Path A with automation at one boundary. The viral signature pollution remains for the entire chain below.

---

6. The Design Space

Mechanism	Delivers to `operator new`?	Reduces to
Parameter list	Yes	Path A
`coroutine_traits`	Compile time only	Path A
`await_transform`	No (too late)	-
`operator co_await`	No (too late)	-
`initial_suspend`	No (too late)	-
Deferred invocation	Indirectly	Path A or B
Promise-based injection	No (parent inaccessible)	Path A
Sender-env bridge wrapper	First hop only	Path A
Thread-local storage	Yes	Path B
Global `pmr` resource	Yes	Path B
Fiber-local storage	Yes	Path B

Every row that delivers to `operator new` is either Path A (the allocator is in the parameter list) or Path B (the allocator is in ambient state). Every row that is neither Path A nor Path B does not deliver to `operator new`.

---

7. The Tradeoff

The choice between Path A and Path B is a choice between two costs:

Property	Path A (`allocator_arg_t`)	Path B (thread-local)
Allocator reaches `operator new`	Yes	Yes
Signature pollution	Every coroutine in the chain	None
Call-site pollution	Every `co_await` forwards the allocator	None
Hidden state	None	Thread-local write-through cache
Thread migration	No concern	Requires restore on resume
Per-chain scoping	Explicit in parameters	Requires protocol discipline
Ergonomic cost	High (viral parameter threading)	Low (transparent to coroutine authors)

Path A makes the allocator visible in every signature. Path B makes it invisible. Path A requires every coroutine author to thread the allocator through every call site. Path B requires the protocol machinery to maintain the thread-local invariant. Both are correct. The tradeoff is signature pollution versus ambient state.

7.1 The Ergonomic Cost of Path A

The table above summarizes the tradeoff. The following example makes it concrete. Consider a three-deep coroutine chain: `accept_loop` calls `handle_request` calls `parse_header`.

Path A - every signature carries the allocator, every call site forwards it:

```
task<header, env> parse_header(
    std::allocator_arg_t, auto alloc,
    socket& s, buffer& buf)
{
    auto line = co_await s.read_some(buf);
    co_return parse(line);
}

task<void, env> handle_request(
    std::allocator_arg_t, auto alloc,
    socket& s)
{
    buffer buf;
    auto hdr = co_await parse_header(
        std::allocator_arg, alloc, s, buf);
    co_await s.write(make_response(hdr));
}

task<void, env> accept_loop(
    std::allocator_arg_t, auto alloc,
    acceptor& acc)
{
    while (true) {
        auto s = co_await acc.accept();
        co_await handle_request(
            std::allocator_arg, alloc, s);
    }
}
```

Path B - clean signatures, plain calls:

```
task<header> parse_header(
    socket& s, buffer& buf)
{
    auto line = co_await s.read_some(buf);
    co_return parse(line);
}

task<void> handle_request(socket& s)
{
    buffer buf;
    auto hdr = co_await parse_header(
        s, buf);
    co_await s.write(make_response(hdr));
}

task<void> accept_loop(acceptor& acc)
{
    while (true) {
        auto s = co_await acc.accept();
        co_await handle_request(s);
    }
}
```

With Path A, the first two parameters of every coroutine have nothing to do with what the function does. `parse_header` parses a header from a socket. The allocator is infrastructure, not interface. Every tutorial, every example, every code review must explain the leading parameter pair. Forgetting `allocator_arg` at one call site is not a compile error - the code compiles, runs correctly, and silently falls back to the default allocator.

With Path B, the coroutine signatures are pure domain logic. The allocator propagates through three coroutine frames without any coroutine author writing any allocator-related code.

7.2 Compilation Model Consequences

Path A with a generic allocator parameter (`auto alloc`) makes every coroutine a function template. This has consequences beyond ergonomics:

- **Separate compilation.** A template coroutine must be visible at the point of instantiation. It cannot be compiled in one translation unit and called from another without exposing the full definition in a header.
- **Virtual functions.** Templates cannot be virtual member functions. A coroutine that takes `auto alloc` cannot participate in a class hierarchy through virtual dispatch.

- **Type erasure.** A template coroutine cannot be stored behind `std::function` or a vtable without first fixing the allocator type.

Fixing the allocator type recovers separate compilation:

```
using alloc_t =
    std::pmr::polymorphic_allocator<std::byte>;

task<void, env> handle_request(
    std::allocator_arg_t, alloc_t alloc,
    socket& s);
```

This is no longer a template. It can be separately compiled and virtual. But the allocator type is now hardcoded into every signature. Code that uses a different allocator type cannot call these coroutines without wrapping the allocator in a `polymorphic_allocator` at every call site.

Choice	Separately compiled?	Any allocator?	Virtual?
`auto alloc`	No	Yes	No
Fixed allocator	Yes	PMR only	Yes
Path B (no param)	Yes	Yes	Yes

Path B sidesteps the entire question. The coroutine signature has no allocator parameter. It is not a template for allocator reasons. It can be separately compiled, virtual, and type-erased. The allocator type is the promise type's concern, not the function signature's concern.

---

8. The Hybrid

The two paths are not mutually exclusive. A conforming promise type can provide both, and the compiler's overload resolution dispatches automatically:

```
struct promise_type {
    // Path A: allocator_arg_t in parameter list
    template<typename Alloc, typename... Args>
    static void* operator new(std::size_t sz,
        std::allocator_arg_t, Alloc& a,
        Args&&...)
    {
        return allocate_frame(sz, a);
    }

    // Path A: member function variant
    template<typename T, typename Alloc,
             typename... Args>
    static void* operator new(std::size_t sz,
        T&, std::allocator_arg_t, Alloc& a,
        Args&&...)
    {
        return allocate_frame(sz, a);
    }

    // Path B: ambient state fallback
    static void* operator new(std::size_t sz)
    {
        auto* mr =
            get_cached_frame_allocator();
        if (!mr)
            mr = std::pmr::new_delete_resource();
        return allocate_frame(sz, mr);
    }
};
```

If the coroutine's parameter list begins with `allocator_arg_t`, the first overload wins. Otherwise the parameterless overload wins and reads the current frame allocator from ambient state. The dispatch is entirely mechanical - no user intervention, no protocol change, no new types.

The consequences:

- **Users who reject ambient state** pass `allocator_arg_t` at every call site. The thread-local is never read. They pay the signature pollution documented in Section 7, but they get fully explicit control.
- **Users who want clean signatures** use the IoAwaitable protocol as-is. The ambient state propagates the allocator transparently. They never touch `allocator_arg_t`.
- **Users who want both** can pass `allocator_arg_t` at specific boundaries and let ambient state handle the rest of the chain.

Both overloads live in the same promise type. A conforming promise mixin (P4003R3 [4] Section 7) provides both paths. Task types that inherit from the mixin support both paths automatically. No ecosystem fragmentation occurs because the choice is per-call-site, not per-task-type.

---

9. Addressing TLS Concerns

Thread-local storage has a well-deserved reputation for creating hidden coupling and brittle behavior. The concerns are familiar. This section examines each one as it applies to the specific case of frame allocator propagation.

9.1 Platforms Without Thread-Local Storage

The concern is that TLS excludes embedded systems, bare-metal targets, GPUs, and some RTOS environments. A survey of platforms reveals that this concern, while intuitive, defends a platform combination that does not exist in practice.

Frame allocator customization requires three capabilities simultaneously: C++20 coroutines (the language feature), `<memory_resource>` (the standard PMR header, hosted-only [5]), and a heap worth customizing. The following table categorizes platforms by these capabilities:

Platform category	Coroutines	`thread_local`	PMR	Heap
Desktop (Linux, Windows, macOS)	Yes	Yes	Yes	Yes
Mobile (iOS, Android)	Yes	Yes	Yes	Yes
Game consoles (Xbox, PS5)	Yes	Yes	Yes	Yes
Full RTOS (QNX, Zephyr, VxWorks)	Yes	Yes	Hosted	Yes
Lightweight RTOS (FreeRTOS, Pico)	Partial	No	No	Yes
Bare metal (Cortex-M, RISC-V)	Partial	No	No	Rare
GPU device code (CUDA, SYCL)	No	No	No	No

Every platform in the top four rows - where all three capabilities are present - has `thread_local`. Every platform in the bottom three rows lacks at least one of the other prerequisites.

Two cases deserve specific attention:

- **Raspberry Pi Pico (RP2040).** Has C++20 coroutines (via community libraries), has `malloc`, but `thread_local` support remains an open pull request (targeted for SDK 2.3.0). However, the Pico SDK uses newlib in freestanding mode - `<memory_resource>` is not available. There is no `std::pmr::memory_resource` to propagate.
- **Pigweed** (Google's embedded framework [6]). Has C++20 coroutines via `pw::async2::Coro`. Uses its own `pw::Allocator`, not PMR. Every coroutine takes a `CoroContext` (wrapping an allocator) as its first parameter - Path A by design. Pigweed

chose Path A not because TLS is unavailable, but because it operates in freestanding environments where PMR does not exist.

The `<memory_resource>` header is hosted-only [5]. Freestanding implementations are not required to provide it. Platforms without TLS are overwhelmingly freestanding. The intersection of "has `<memory_resource>`" and "lacks `thread_local`" is empty across every platform surveyed.

The hybrid (Section 8) addresses the theoretical case: the `allocator_arg_t` overload is always available. On any hypothetical platform without TLS, the Path B overload falls back to `std::pmr::new_delete_resource()` - the same behavior as not customizing the frame allocator at all. But no such platform has been identified that also provides the hosted standard library features needed to make frame allocator customization meaningful.

9.2 Performance

TLS has earned a reputation for overhead in two cases: variables with dynamic initializers (which require a guard check on every access) and variables in dynamically loaded shared libraries (which may require a slow `__tls_get_addr` call).

The frame allocator thread-local is neither. It is a zero-initialized pointer with no dynamic initializer and no destructor. The loader zeros it as part of the static TLS block at thread creation. On Linux, reading it is a single `fs:`-relative load - one instruction. On Windows, even the DLL case is a few pointer chases - single-digit nanoseconds. For I/O coroutines where the operation itself takes microseconds, the TLS access is not measurable.

9.3 Invisible Action at a Distance

The standard objection to ambient state is that a function's behavior depends on state not in its signature. This concern is valid for state that affects correctness. The frame allocator does not affect correctness.

The frame allocator is a performance policy, not a correctness invariant. A coroutine that receives the wrong frame allocator - or no frame allocator at all - produces the correct result. Its frame is allocated from a different memory pool. The program runs. The output is identical. The difference is where the bytes came from.

This distinguishes the frame allocator from the other two values in `io_env`:

Context	Wrong value	Consequence
Executor	Coroutine runs on the wrong thread	Data races, UB
Stop token	Cancellation does not propagate	Hangs, deadlocks
Frame allocator	Frame allocated from a different memory pool	Suboptimal allocation strategy

A protocol violation for the frame allocator degrades to the default allocation strategy - the same behavior every coroutine has when no frame allocator is customized at all.

9.4 Silent Failure on Protocol Violation

The concern is that a broken TLS chain silently delivers the wrong allocator. This is true. It is equally true of `allocator_arg_t`.

With `allocator_arg_t`, the protocol violation is forgetting to pass the allocator at a call site:

```
task<void> handle_request(socket& s)
{
    // forgot allocator_arg here
    auto hdr = co_await parse_header(s);

    // remembered here
    co_await send_response(
        std::allocator_arg, alloc, s, hdr);
}
```

The code compiles. It runs. `parse_header`'s frame is allocated from `new_delete_resource()`. No error. Silent fallback.

Property	`allocator_arg_t`	Thread-local
Failure mode	Wrong allocator, program correct	Wrong allocator, program correct
Who can cause the failure	Every application developer at every call site	Library author writing a custom promise type
Frequency of opportunity	Every `co_await` in every coroutine	Once per promise type implementation
Prevention	Programmer discipline at every call site	Use the standard promise mixin

The failure surface with `allocator_arg_t` is every call site in every coroutine written by every application developer. The failure surface with TLS is the promise type implementation - written once by a library author, typically by inheriting from the standard mixin. The silent failure objection applies to both paths. The failure is more likely with Path A.

9.5 Precedent

The concern is that standardizing TLS for frame allocators normalizes TLS as a general propagation mechanism, inviting TLS for executors, logging contexts, error handlers, and other cross-cutting concerns.

The `operator new` timing gap is unique to the frame allocator. The executor arrives through `await_suspend`, after the frame exists. The stop token arrives through `await_suspend`. Logging contexts, error handlers, and container allocators can be passed as parameters or stored in `io_env` - none face the constraint that `operator new`'s fixed signature imposes on the frame allocator.

TLS for the frame allocator does not generalize because the constraint does not generalize. A proposal to use TLS for executors would be answered: `await_suspend` delivers the executor, no TLS needed. The precedent is narrow and structurally bounded.

9.6 Categorical Rejection

Some hold that thread-local storage is categorically inappropriate for standard library mechanisms, regardless of the specific use case. This is a legitimate position. The hybrid (Section 8) provides the answer: users who hold this position use `allocator_arg_t` exclusively. The thread-local is never read. `std::pmr::get_default_resource()` provides a process-wide allocator channel through similar ambient state, adopted in C++17.

---

10. Anticipated Objections

Q: Return an awaitable instead of a task, and let `await_transform` create the frame.

A: This works for leaf I/O operations, which are already plain awaitables (Section 2). It does not work for child coroutines. A function that contains `co_await` is a coroutine - it has a frame, allocated by the call expression, before `await_transform` runs. The only way to avoid the frame is to not use a coroutine, which means hand-writing the state machine the compiler generates.

Q: A future language change could add a new delivery mechanism.

A: A language change that provides a new way to pass runtime state to `operator new` without putting it in the parameter list would be a new form of ambient state - a new instance of Path B. The two-path structure is a consequence of `operator new`'s fixed interface, not of the current set of customization points. A language change that alters `operator new`'s interface would be a different analysis.

Q: HALO (Heap Allocation eLision Optimization) eliminates the frame allocation entirely.

A: When HALO applies, the frame is stack-allocated and the allocator is irrelevant. This paper addresses the case where HALO does not apply - the general case for I/O coroutines, where the frame outlives the caller.

Acknowledgments

The author thanks Michael Hava for identifying that R0 failed to distinguish leaf I/O awaitables from child coroutines, prompting the scoping clarification in Section 2; Steve Gerbino and Mungo Gill for Capy and Corosio implementation work, where the frame allocator timing constraint was first encountered in practice; and Klemens Morgenstern for discussions on coroutine frame allocation and thread-local propagation.

References

[1] P4100R0 - "Coroutine-Native I/O for C++29 (The Network Endeavor)" (Vinnie Falco, Steve Gerbino, Michael Vandeberg, Mungo Gill, Mohammad Nejati, 2026).

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

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

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

[5] Freestanding and hosted implementations - cppreference.com. `<memory_resource>` is not required for freestanding implementations.

[6] Pigweed pw_async2: Coroutines - Google's embedded C++20 coroutine framework. Every coroutine takes a `CoroContext` (wrapping a `pw::Allocator`) as its first parameter.