Document:	**D4003**
Date:	2026-01-21
Reply-to:	Vinnie Falco <vinnie.falco@gmail.com> Steve Gerbino <steve@gerbino.co> Amlal El Mahrouss <amlalelmahrouss@icloud.com> Mungo Gill <mungo.gill@me.com>
Audience:	Boost, C++

IoAwaitables: A Coroutines-Only Framework

Abstract

This paper asks: *what would an execution model look like if designed from the ground up for coroutine-only asynchronous I/O?*

An execution model answers how asynchronous work is scheduled and dispatched, where operation state is allocated and by whom, and what customization points allow users to adapt the framework's behavior. We propose a **coroutines-only** execution model optimized for CPU-bound I/O workloads. The framework introduces the *IoAwaitable* protocol: a system for associating a coroutine with an executor, stop token, and allocator, and propagating this context forward through a coroutine chain which can end at an operating system API boundary where asynchronous operations are performed.

We compare our use-case-first driven design against `std::execution` (P2300 (https://wg21.link/p2300)) and observe significant divergence. Analysis of P2300 (https://wg21.link/p2300) and its evolution (P3826 (https://wg21.link/p3826)) reveals a framework driven by GPU and parallel workloads—P3826 (https://wg21.link/p3826)'s focus is overwhelmingly GPU/parallel with no networking discussion. Core networking concerns—strand serialization, I/O completion contexts, platform integration—remain unaddressed. The query-based context model that P3826 (https://wg21.link/p3826) attempts to fix is unnecessary when context propagates forward through coroutine machinery.

Our framework demonstrates what emerges when networking requirements drive the design rather than being adapted to a GPU-focused abstraction.

1. Introduction

The C++ standardization effort has produced `std::execution` (P2300 (https://wg21.link/p2300)), a sender/receiver framework for structured concurrency. Its design accommodates heterogeneous computing—GPU kernels, parallel algorithms, and hardware accelerators. The machinery is substantial: domains select algorithm implementations, queries determine completion schedulers, and transforms dispatch work to appropriate backends.

But what does asynchronous I/O actually need?

A network server completes read operations on I/O threads, resumes handlers on application threads, and serializes access to connection state. A file system service batches completions through io_uring, dispatches callbacks to worker pools, and manages buffer lifetimes across suspension points. These patterns repeat across every I/O-intensive application. None of them require domain-based algorithm dispatch. TCP servers do not run on GPUs. Socket reads have one implementation per platform, not multiple hardware backends.

This paper explores what emerges when I/O requirements drive the design.

We find that two operations for resuming coroutines suffice: `dispatch` for continuations and `post` for independent work. The choice between them is correctness, not performance. Using dispatch while holding a mutex invites deadlock. Using post for every continuation wastes cycles bouncing through queues. But primitives alone don't solve the composition problem. In a chain like `async_http_request → async_read → socket`, who decides which allocator to use? Who determines which executor runs the completion? How are operations canceled? The answers are discovered from studying the use case:

- **The application decides executor policy.** Thread affinity, strand serialization, priority scheduling; these are deployment decisions. A read operation doesn't care which thread pool it runs on. The application architecture determines where completions should resume.

- **The application sends stop signals.** Cancellation flows downward from application logic, not upward from I/O operations. A timeout, user abort, or graceful shutdown originates at the application layer and propagates to pending operations. The socket doesn't decide when to cancel itself.

- **The application decides allocation policy.** Memory strategies: recycling pools, bounded allocation, per-tenant budgets; these are a property of the coroutine chain, not the I/O operation. A socket doesn't care about memory policy; the context in which it runs does.

- **The execution context owns its I/O objects.** A socket registered with epoll on thread A cannot have completions delivered to thread B's epoll. The physical coupling is inherent. The call site cannot know which event loop the socket uses—only the socket knows.

Execution context flows naturally *forward* through async operation chains. The executor, stop token, and allocator propagate from caller to callee at each suspension point. This forward flow addresses the late-binding challenge differently than `std::execution`: context is known at launch site, not discovered through receiver queries after the work graph is built. Our goal is not to replace `std::execution` for GPU workloads—it is to demonstrate that networking deserves a purpose-built abstraction rather than adaptation to a framework optimized for different requirements.

> ***Convention.*** *Throughout this document we use the following type alias:*
>
> ```
> using coro = std::coroutine_handle<void>;
> ```

2. Understanding Asynchronous I/O

Not every committee member or library reviewer works with network programming daily, and the challenges that shape I/O library design may not be immediately obvious from other domains. This section provides the background needed to evaluate the design decisions that follow. The concepts presented here draw heavily from Christopher Kohlhoff's pioneering work on Boost.Asio, which has served the C++ community for over two decades, and from Gor Nishanov's C++ coroutines that now enable elegant expression of asynchronous control flow. Readers already familiar with event loops, completion handlers, and executor models may skip ahead to §3.

2.1 The Problem with Waiting

Network I/O operates on a fundamentally different timescale than computation. A CPU executes billions of instructions per second; reading a single byte from a local network takes microseconds, and from a remote server, milliseconds. The disparity is stark:

Operation	Approximate Time
CPU instruction	0.3 ns
L1 cache access	1 ns
Main memory access	100 ns
Local network round-trip	500 µs
Internet round-trip	50-200 ms

When code calls a blocking read on a socket, the thread waits—doing nothing—while the network delivers data. During a 100ms network round-trip, a modern CPU could have executed 300 billion instructions. Blocking I/O wastes this potential.

```
// Blocking I/O: thread waits here
char buf[1024];
ssize_t n = recv(fd, buf, sizeof(buf), 0);  // Thread blocked
process(buf, n);
```

For a single connection, this inefficiency is tolerable. For a server handling thousands of connections, it becomes catastrophic.

2.2 The Thread-Per-Connection Trap

The natural response to blocking I/O is to spawn a thread per connection. Each thread blocks on its own socket; while one waits, others make progress.

```
void handle_client(socket client) {
    char buf[1024];
    while (auto [ec, n] = client.read_some(buf); !ec) {
        process(buf, n);
    }
}

// Spawn a thread for each connection
for (;;) {
    socket client = accept(listener);
    std::thread(handle_client, std::move(client)).detach();
}
```

This works—until it doesn't. Each thread consumes memory (typically 1MB for the stack) and creates scheduling overhead. Context switches between threads cost thousands of CPU cycles. At 10,000 connections, you have 10,000 threads consuming 10GB of stack space, and the scheduler spends more time switching between threads than running actual code.

The C10K problem (http://www.kegel.com/c10k.html)—handling 10,000 concurrent connections—revealed that thread-per-connection doesn't scale. Modern servers handle millions of connections. Something else is needed.

2.3 Event-Driven I/O

The solution is to invert the relationship between threads and I/O operations. Instead of one thread per connection, use a small number of threads that multiplex across many connections. The operating system provides mechanisms to wait for *any* of a set of file descriptors to become ready:

- Linux: `epoll` — register interest in file descriptors, wait for events
- Windows: I/O Completion Ports (IOCP) — queue-based completion notification
- BSD/macOS: `kqueue` — unified event notification

These mechanisms enable the **proactor pattern**: instead of blocking until an operation completes, you *initiate* an operation and receive notification when it finishes. The thread is free to do other work in the meantime.

```
io_context ioc;
socket sock(ioc);
sock.open();

// Initiate an async operation - returns immediately
auto [ec] = co_await sock.connect(endpoint(ipv4_address::loopback(), 8080));
// Execution resumes here when the connection completes
```

The `io_context` is the heart of this model. It maintains a queue of pending operations and dispatches completions as they arrive from the OS. Calling `ioc.run()` processes this queue:

```
io_context ioc;
// ... set up async operations ...
ioc.run();  // Process completions until no work remains
```

A single thread calling `run()` can service thousands of connections. For CPU-bound workloads, multiple threads can call `run()` on the same context, processing completions in parallel.

2.4 Completion Handlers and Coroutines

Early asynchronous APIs used callbacks to handle completions:

```
// Callback-based async (traditional style)
socket.async_read(buffer, [](error_code ec, size_t n) {
    if (!ec) {
        // Process data, then start another read...
        socket.async_read(buffer, [](error_code ec, size_t n) {
            // More nesting...
        });
    }
});
```

This “callback hell” inverts control flow, making code hard to follow and debug. Error handling becomes scattered across nested lambdas. State must be explicitly captured and managed.

C++20 coroutines restore sequential control flow while preserving the efficiency of asynchronous execution:

```
// Coroutine-based async (modern style)
task<> handle_connection(socket sock) {
    char buf[1024];
    for (;;) {
        auto [ec, n] = co_await sock.read_some(buf);
        if (ec)
            co_return;
        co_await process_data(buf, n);
    }
}
```

The `co_await` keyword suspends the coroutine until the operation completes, then resumes execution at that point. The code reads sequentially, but executes asynchronously. The `task<>` return type represents a coroutine that can be awaited by a caller or launched independently.

2.5 The Execution Context

I/O objects must be associated with an execution context that manages their lifecycle and delivers completions. A `socket` created with an `io_context` is registered with that context's platform reactor (epoll, IOCP, etc.). This binding is physical—the socket's file descriptor is registered with specific kernel structures.

```
io_context ioc;
socket sock(ioc);  // Socket bound to this context
sock.open();

// The socket's completions will be delivered through ioc
auto [ec] = co_await sock.connect(endpoint);
```

This binding has implications: - A socket cannot migrate between contexts - Completions are delivered to the context that owns the socket - The context must remain alive while operations are pending

The `io_context` abstracts platform differences. On Windows, it wraps an I/O Completion Port. On Linux, it wraps epoll (or io_uring). Application code remains portable while the implementation leverages platform-specific optimizations.

2.6 Executors

An **executor** determines where and how work runs. It answers: when an async operation completes, which thread should run the completion handler? Should it run immediately, or be queued for later?

```
auto ex = ioc.get_executor();
```

The executor provides two fundamental operations:

`dispatch` — Run work immediately if safe, otherwise queue it. When the I/O context thread detects a completion, it typically dispatches the waiting coroutine inline for minimal latency.

`post` — Always queue work for later execution. Use this when you need a guarantee that the work won't run until after the current function returns—for example, when holding a lock.

```
// Dispatch: may run inline
ex.dispatch(continuation);

// Post: always queued
ex.post(new_work);
```

The distinction matters for correctness. Dispatching while holding a mutex could cause the completion handler to run immediately, potentially deadlocking if it tries to acquire the same mutex. Posting guarantees the handler runs later, after the lock is released.

2.7 Strands: Serialization Without Locks

When multiple threads call `ioc.run()`, completions may execute concurrently. If two coroutines access shared state, you need synchronization. Mutexes work but introduce blocking—the very thing async I/O tries to avoid.

A **strand** provides an alternative: it guarantees that handlers submitted through it never execute concurrently, without using locks.

```
strand my_strand(ioc.get_executor());

// Entire coroutine runs serialized through the strand
run_async(my_strand)(handle_connection(sock));
```

Handlers on a strand execute in FIFO order, one at a time. Multiple strands can make progress concurrently on different threads, but within a single strand, execution is sequential. This enables safe concurrent access to connection state without explicit locking.

2.8 Cancellation

Long-running operations need a way to stop gracefully. A connection might timeout. A user might close a window. A server might be shutting down.

C++20's `std::stop_token` provides cooperative cancellation:

```
std::stop_source source;
std::stop_token token = source.get_token();

// Launch a coroutine with a stop token
run_async(ex, token)(long_running_operation());

// Later, request cancellation
source.request_stop();
```

The stop token propagates through the coroutine chain. At the lowest level, I/O objects observe the token and cancel pending operations with the appropriate OS primitive (`CancelIoEx` on Windows, `IORING_OP_ASYNC_CANCEL` on Linux). The operation completes with an error, and the coroutine can handle it normally.

Cancellation is cooperative—no operation is forcibly terminated. The I/O layer requests cancellation, the OS acknowledges it, and the operation completes with an error code. This keeps resource cleanup predictable and avoids the hazards of abrupt termination.

2.9 Moving Forward

With these fundamentals in hand—event loops, executors, strands, and cancellation—you have the conceptual vocabulary to understand the design decisions in the sections that follow. These patterns form the bedrock of modern C++ networking: every high-performance server, every responsive client application, builds on some combination of non-blocking I/O, completion handlers, and execution contexts.

If you're eager to experiment, the Corosio (https://github.com/cppalliance/corosio) library implements these concepts in production-ready code. It provides sockets, timers, TLS, and DNS resolution—all built on the coroutine-first model we'll explore in depth. The Boost.Asio (https://www.boost.org/doc/libs/release/doc/html/boost_asio.html) documentation and its many community tutorials offer additional paths to hands-on learning. Building a simple echo server or chat application is one of the best ways to internalize how these pieces fit together.

The rest of this paper examines what an execution model looks like when these networking requirements drive the design from the ground up.

3. What Networking Needs

Networking has specific requirements that differ from parallel computation. A socket read has **one implementation per platform**—there's no algorithm selection, no hardware heterogeneity to manage. There are other important concerns:

- **Platform implementation**: Integration with IOCP, epoll, io_uring
- **Strand serialization**: Ordering guarantees for concurrent access
- **Thread affinity**: Handlers must resume on the correct thread
- **Inline vs queued**: Choosing whether continuations run immediately or are deferred
- **Stack depth control**: Bounding recursion through trampolining or deferred dispatch

Strip these requirements down to their essence. What primitive do they all depend on? **Something that runs work**. An *executor*. Completion contexts, strands, and thread affinity are all built on executors. I/O objects must be bound to an executor's context to work correctly. P2762 (https://wg21.link/p2762) acknowledges this reality in §4.1:

> *"It was pointed out networking objects like sockets are specific to a context: that is necessary when using I/O Completion Ports or a TLS library and yields advantages when using, e.g., io_uring(2)."*

3.1 The Executor

An executor is to functions what an allocator is to memory: while an allocator controls *where* objects live, an executor controls *how* functions run. Examining decades of asynchronous I/O patterns reveals that a minimal executor needs only two operations. The first operation handles work that continues the current operation—a completion handler that should run as part of the same logical flow. The second handles work that must begin independently—a task that cannot start until after the initiating call returns. These two cases have distinct correctness requirements, captured by `dispatch` and `post` .

`dispatch`

When an I/O operation completes at the operating system level, the suspended coroutine must resume. The kernel has signaled that data is ready or a connection is established; now user code must process the result. In this context, resuming the coroutine immediately, on the current thread and without queuing, is both safe and efficient. No locks are held by the I/O layer. The completion is a natural continuation of the event loop's work.

This pattern is captured by `dispatch` . It resumes the provided coroutine immediately if doing so is safe within the executor's rules, or queues it otherwise. For a strand, "safe" means no other coroutine is currently executing on that strand. For a thread pool, it might mean the caller is already on a pool thread. The executor decides; the caller simply requests execution.

This is the common case for I/O completions. The event loop calls `dispatch` with the coroutine, and in the typical case the coroutine resumes inline without touching any queue. The result is minimal latency and zero allocation for the dispatch itself.

`post`

Sometimes inline execution is not just inefficient but incorrect. Consider code that initiates an async operation while holding a mutex, expecting the lock to be released before the completion runs. Or a destructor that launches cleanup work and must return before that work begins. These patterns require a guarantee: the coroutine will not execute before the call returns.

The `post` operation provides this guarantee. It always queues coroutines for later execution, never running inline. The caller can rely on control returning before the coroutine begins. This makes `post` suitable for breaking call chains, avoiding reentrancy, and ensuring ordering when the caller's state must settle before continuation.

The cost is queue insertion and eventual dequeue—cycles that `dispatch` avoids when inline execution is safe. But when correctness requires deferred execution, `post` is the only sound choice.

3.2 The `stop_token`

C++20 introduced `std::stop_token` as a cooperative cancellation mechanism. A `stop_source` owns the cancellation state; the token provides a read-only view that can be queried or used to register callbacks. When `stop_source::request_stop()` is called, all associated tokens observe the cancellation and registered callbacks fire.

Cancellation flows downward. The application—not the I/O operation—decides when to stop. A timeout, user abort, or graceful shutdown originates at the application layer and propagates to pending operations. The socket doesn't decide when to cancel itself; it receives a stop request from above.

In our model, the stop token is injected at the coroutine launch site and propagated through the entire chain. Consider:

```
http_client → http_request → write → write_some → socket
```

The top-level caller provides a `stop_token`. Each coroutine in the chain receives it and passes it forward. At the end of the chain, the I/O object—the socket—receives the token and can act on it.

This is where the operating system enters. Modern platforms provide cancellation at the kernel level:

- **Windows IOCP**: `CancelIoEx` cancels pending overlapped operations on a specific handle
- **Linux io_uring**: `IORING_OP_ASYNC_CANCEL` cancels a previously submitted operation by its user data
- **POSIX**: `close()` on the file descriptor, though less graceful, interrupts blocking operations

When the stop token signals cancellation, the I/O object registers a callback that invokes the appropriate OS primitive. The pending operation completes with an error—typically `operation_aborted`—and the coroutine chain unwinds normally through its error path.

This design keeps cancellation cooperative and predictable. No operation is forcibly terminated mid-flight. The I/O layer requests cancellation; the OS acknowledges it; the operation completes with an error; the coroutine handles the error. Each layer respects the boundaries of the next.

3.3 The Allocator

Neither callbacks nor coroutines achieve competitive performance without allocation control. Early testing showed that fully type-erased callbacks—using `std::function` at every composition boundary—allocated hundreds of times per invocation and ran an order of magnitude slower than native templates. Non-recycling coroutine configurations perform so poorly, dominated by heap allocation overhead, that they are not worth benchmarking.

Recycling is mandatory for performance. Thread-local recycling—caching recently freed memory for immediate reuse—enables zero steady-state allocations. Callbacks achieve this through “delete before dispatch”: deallocate operation state before invoking the completion handler. Coroutines achieve it through frame pooling: custom `operator new/delete` recycles frames across operations.

But thread-local recycling optimizes for throughput, not all applications. Different deployments have different needs:

- **Memory conservation**: Embedded systems or resource-constrained environments may prefer bounded pools over unbounded caches
- **Per-tenant limits**: Multi-tenant servers need allocation budgets per client to prevent resource exhaustion
- **Debugging and profiling**: Development builds may want allocation tracking that production builds eliminate

Coroutines are continuously created—every `co_await` may spawn new frames. An execution model must make allocation a first-class customization point that controls *all* coroutine allocations, not just selected operations.

Without this, memory policy becomes impossible to enforce.

The allocator must be present at invocation. Coroutine frame allocation has a fundamental timing constraint: `operator new` executes before the coroutine body. When a coroutine is called, the compiler allocates the frame first, then begins execution. Any mechanism that injects context later—receiver connection, `await_transform`, explicit method calls—arrives too late.

P2762 (https://wg21.link/p2762) acknowledges that the receiver-based model brings scheduler and operation together "rather late"—at `connect()` time, after the work graph is already built:

> *"When the scheduler is injected through the receiver the operation and used scheduler are brought together rather late, i.e., when `connect(sender, receiver)` ing and when the work graph is already built."*

For coroutines, this timing is fundamentally incompatible with frame allocation. The allocator must be discoverable from the coroutine's launch site before the coroutine frame is created.

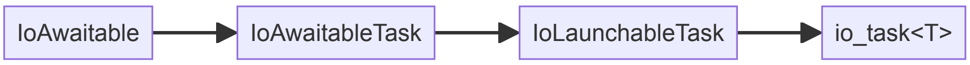
3.4 Backward Flow in `std::execution`

The `std::execution` model (P2300 (https://wg21.link/p2300)) uses a **backward flow** where context—scheduler, allocator, stop token—is discovered by querying the receiver's environment *after* connection. For GPU workloads, where the work graph is constructed before execution begins, this model works well.

For networking, there is a timing problem lurking. We explore the consequences in §4.7.

4. Our Solution: The *IoAwaitable* Protocol

The *IoAwaitable* protocol is a family of concepts layered on top of each other, working together to deliver what coroutines and networking need for correctness and performance:



$$\begin{tikzcd} \text{IoAwaitable} \arrow[r] & \text{IoAwaitableTask} \arrow[r] & \text{IoLaunchableTask} \arrow[r] & \text{io\_task<T>} \end{tikzcd}$$

To help readers understand how these requirements fit together, this paper provides the `io_awaitable_support` CRTP mixin (§8.1) as a non-normative reference implementation. It is not proposed for standardization—implementors may write their own machinery—but examining it clarifies how the protocol works in practice. The mixin also provides the `this_coro::executor` and `this_coro::stop_token` accessors, which allow coroutines to retrieve their bound context without suspending.

4.1 IoAwaitable

The *IoAwaitable* concept allows a coroutine to receive **context**: the three things a coroutine needs for I/O—the executor, the stop token, and the allocator. Context propagates **forward** from caller to callee using a well-defined, easy to understand protocol. Most importantly, this protocol *does not require templates* in the definition of its coroutine machinery. Because the protocol uses type-erased wrappers like `executor_ref` rather than template parameters, context types do not leak into the awaitable's type signature— `task<T>` remains simply `task<T>` , not `task<T, Executor, StopToken, Allocator>` .

```
template< typename A >
concept IoAwaitable =
    requires(
        A a, coro h, executor_ref ex, std::stop_token token )
    {
        a.await_suspend( h, ex, token );
    };
```

The key insight: `await_suspend` receives the executor and stop token as parameters, injected by the caller's `await_transform` . The allocator propagates via `thread_local` storage during a narrow execution window with specific guarantees, ensuring availability at frame allocation time (more on this later).

Satisfying IoAwaitable

The *IoAwaitable* concept is the foundation of the protocol. Any type that can be `co_await` ed and participates in context propagation must satisfy this concept.

Requirements:

1. Implement `await_suspend(std::coroutine_handle<> cont, executor_ref ex, std::stop_token token)`
2. Store or forward the executor and stop token as needed
3. Return a `std::coroutine_handle<>` for symmetric transfer (or `void` / `bool` per standard rules)
4. Implement `await_ready()` and `await_resume()` per standard awaitable requirements

Example implementation:

```
struct my_awaitable
{
    bool await_ready() const noexcept { return false; }

    // This signature satisfies IoAwaitable
    coro await_suspend( coro cont, executor_ref ex, std::stop_token token )
    {
        // Store context for the operation
        cont_ = cont;
        ex_ = ex;
        token_ = token;
        start_async_operation();
        return std::noop_coroutine();
    }

    T await_resume() { return result_; }
};
```

> ***Non-normative note.*** *Implementors may wish to enforce protocol compliance at API boundaries. When a compliant coroutine's* `await_transform` *calls the three-argument* `await_suspend` *, a non-compliant awaitable (lacking this signature) will produce a compile error. Similarly, a compliant awaitable awaited from a non-compliant coroutine will fail to compile. This provides static checking that both sides of each suspension point participate in the protocol:*

```
template<typename A>
auto await_transform(A&& a) {
    static_assert(IoAwaitable<A>,
        "Awaitable does not satisfy IoAwaitable; "
        "await_suspend(coro, executor_ref, stop_token) is required");
    // Return wrapper that forwards to: a.await_suspend(h, executor_, stop_token_)
    ...
}
```

4.2 IoAwaitableTask

Whereas *IoAwaitable* allows *receiving* propagated context, the *IoAwaitableTask* refinement adds the promise interface needed to **store** received context and **propagate** it to child coroutines. This bidirectional capability is what distinguishes a task from a simple awaitable—the task participates fully in the context propagation chain.

```cpp
template<typename T>
concept IoAwaitableTask =
    IoAwaitable<T> &&
    requires { typename T::promise_type; } &&
    requires(
        typename T::promise_type& p,
        typename T::promise_type const& cp,
        executor_ref ex,
        std::stop_token st,
        coro cont )
    {
        { p.set_executor(ex) } noexcept;
        { p.set_stop_token(st) } noexcept;
        { p.set_continuation(cont, ex) } noexcept;
        { cp.executor() } noexcept -> std::same_as< executor_ref >;
        { cp.stop_token() } noexcept -> std::same_as< std::stop_token const& >;
        { cp.complete() } noexcept -> std::same_as< coro >;
    };
```

The concept ensures the promise provides the injection interface (`set_executor` , `set_stop_token`) for receiving context when awaited, and the retrieval interface (`executor` , `stop_token`) for propagating context to children via `await_transform` . The `set_continuation` and `complete` functions work together to implement the same-executor optimization:

```cpp
coro promise_type::complete() const noexcept
{
    if( ! cont_ )
        return std::noop_coroutine();
    if( executor_ == caller_ex_ ) // a single pointer comparison internally
        return cont_;
    caller_ex_.dispatch( cont_ );
    return std::noop_coroutine();
}
```

When a parent `co_await` s a child task, it calls `set_continuation( cont, caller_ex )` , storing both its coroutine handle and its executor. At completion, `complete()` compares the child's executor against the stored caller executor: if they match, it returns the continuation directly for zero-overhead symmetric transfer; if they differ (as with `run`), it dispatches through the caller's executor (which resumes the continuation directly) and returns `noop_coroutine()` to ensure the parent resumes in its expected execution context.

Satisfying IoAwaitableTask

Additional requirements (beyond *IoAwaitable*):

1. Define a nested `promise_type`
2. The promise must provide injection methods:
   - `set_executor(executor_ref)` — stores the executor (must be `noexcept`)
   - `set_stop_token(std::stop_token)` — stores the stop token (must be `noexcept`)
   - `set_continuation(coro, executor_ref)` — stores the continuation handle and caller's executor (must be `noexcept`)
3. The promise must provide retrieval methods:
   - `executor()` — returns `executor_ref` (must be `noexcept`)
   - `stop_token()` — returns `std::stop_token const&` (must be `noexcept`)
   - `complete()` — returns a `coro` for `final_suspend` to use for symmetric transfer (must be `noexcept`)
4. The promise's `await_transform` must intercept child awaitables and inject context
5. Support `operator new` overloads for allocator propagation (read TLS)

Example implementation:

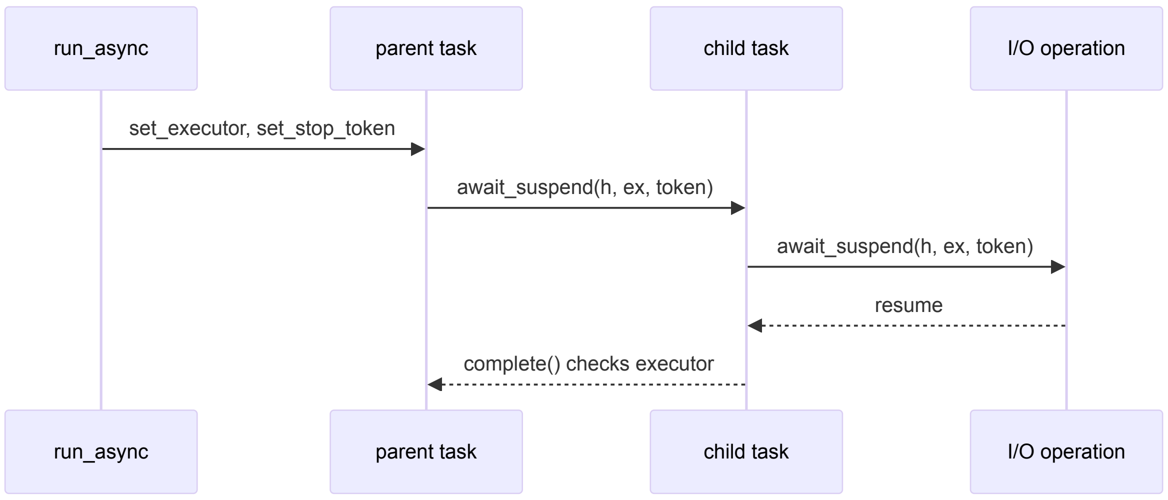
```
template<typename T>
struct task
{
    struct promise_type
    {
        executor_ref executor_;
        executor_ref caller_ex_;
        std::stop_token stop_token_;
        coro cont_;

        void set_executor( executor_ref ex ) noexcept { executor_ = ex; }
        void set_stop_token( std::stop_token st ) noexcept { stop_token_ = st; }
        void set_continuation( coro c, executor_ref ex ) noexcept
        {
            cont_ = c;
            caller_ex_ = ex;
        }

        executor_ref executor() const noexcept { return executor_; }
        std::stop_token const& stop_token() const noexcept { return stop_token_; }

        coro complete() const noexcept
        {
            if( ! cont_ )
                return std::noop_coroutine();
            if( executor_ == caller_ex_ )
                return cont_;  // Same-executor optimization
            caller_ex_.dispatch( cont_ );
            return std::noop_coroutine();
        }

        template<IoAwaitable A>
        auto await_transform( A&& awaitable ) {
            // Return wrapper that forwards to:
            // awaitable.await_suspend(h, executor(), stop_token())
            ...
        }
        // ... result storage, initial/final suspend, etc.
    };

    std::coroutine_handle<promise_type> h_;

    bool await_ready() const noexcept { return false; }
    T await_resume() { return h_.promise().result(); }
    // Satisfies IoAwaitable and enables IoAwaitableTask
    coro await_suspend( coro cont, executor_ref ex, std::stop_token token )
    {
        h_.promise().set_executor( ex );
        h_.promise().set_stop_token( token );
        h_.promise().set_continuation( cont, ex );
        return h_;  // Transfer to child coroutine
    }
};
```

> ***Non-normative note.*** *The* `io_awaitable_support` *CRTP mixin (§8.1) provides all six required promise methods and is offered as a convenience. It is not proposed for standardization—implementors may write the boilerplate themselves.*

4.3 IoLaunchableTask

The *IoLaunchableTask* concept further refines *IoAwaitableTask* with the interface needed by launch functions like `run_async` and `run` . These functions bootstrap context directly into a task—they call `set_executor` and `set_stop_token` on the promise rather than going through the three-argument `await_suspend` . The concept adds the requirements these functions need: `handle()` and `release()` for lifetime management, plus `exception()` and `result()` for completion handling.

```
template<typename T>
concept IoLaunchableTask =
    IoAwaitableTask<T> &&
    requires( T& t, T const& ct, typename T::promise_type const& cp )
    {
        { 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;
    } &&
    ( std::is_void_v< decltype(std::declval<T&>().await_resume()) > ||
     requires( typename T::promise_type& p ) {
         p.result();
     });
```


run_async
parent task
child task
I/O operation
set_executor, set_stop_token
await_suspend(h, ex, token)
await_suspend(h, ex, token)
resume
complete() checks executor
run_async
parent task
child task
I/O operation

- `run_async` is the root of a coroutine chain, launching from non-coroutine code
- `run` performs executor hopping from within coroutine code, binding a child task to a different executor

Because launch functions are constrained on the concept rather than a concrete type, they work with any conforming task:

```
template< Executor Ex, class... Args >
unspecified run_async( Ex ex, Args&&... args );

template< Executor Ex, class... Args >
unspecified run( Ex ex, Args&&... args );
```

This decoupling enables library authors to write launch utilities that work with any conforming task type, and users to define custom task types that integrate seamlessly with existing launchers.

Satisfying IoLaunchableTask

Additional requirements (beyond *IoAwaitableTask*):

1. The task must provide `handle()` returning `std::coroutine_handle<promise_type>` (must be `noexcept`)
2. The task must provide `release()` to transfer ownership without destroying the frame (must be `noexcept`)
3. The promise must provide `exception()` returning any stored `std::exception_ptr` (must be `noexcept`)
4. For non-void tasks, the promise must provide `result()` returning the stored value

Example implementation:

```
template<typename T>
struct task
{
    struct promise_type : io_awaitable_support<promise_type>
    {
        std::exception_ptr ep_;
        std::optional<T> result_;

        std::exception_ptr exception() const noexcept { return ep_; }
        T&& result() noexcept { return std::move(*result_); }

        // ... plus all IoAwaitableTask requirements
    };

    std::coroutine_handle<promise_type> h_;

    std::coroutine_handle<promise_type> handle() const noexcept
    {
        return h_;
    }

    void release() noexcept
    {
        h_ = nullptr;
    }

    // ... plus all IoAwaitableTask requirements
};
```

The `handle()` method provides access to the typed coroutine handle, allowing launch functions to resume the coroutine and access the promise. The `release()` method transfers ownership—after calling it, the task wrapper no longer destroys the frame, leaving lifetime management to the launch function.

For `task<void>`, the `result()` method is not required since there is no value to retrieve. The concept uses a disjunction to handle this:

```
( std::is_void_v< decltype(std::declval<T&>().await_resume()) > ||
  requires( typename T::promise_type& p ) { p.result(); } );
```

> ***Non-normative note.*** *Unlike the* `io_awaitable_support` *mixin which provides promise methods, the* `handle()` *and* `release()` *methods are task-specific. The exception and result storage shown above is illustrative—implementations may use different strategies such as* `std::variant` *for result/exception storage.*

4.4 executor_ref

The type-erasing wrapper `executor_ref` is a **concrete type** that appears uniformly in all protocol signatures—no templates required. The implementation is compact:

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

At just two pointers, `executor_ref` copies cheaply—important since executors propagate through every suspension point in a coroutine chain. Equality comparison reduces to pointer comparison, enabling the same-executor optimization in `complete()` .

The implementation leverages a key property of coroutines: parameter lifetimes in calling coroutines extend until the callee's final suspension. Launch functions preserve a copy of the user's typed *Executor*. The `executor_ref` holds a pointer to the stored value. As the wrapper propagates through the call chain, the original executor remains valid—it cannot go out of scope until all coroutines in the chain are destroyed.

Coroutines can access their context directly using `this_coro::executor` and `this_coro::stop_token` . These never suspend—they return immediately with the stored values:

```
task<void> cancellable_work()
{
    executor_ref ex = co_await this_coro::executor;         // never suspends
    std::stop_token token = co_await this_coro::stop_token; // never suspends

    for (int i = 0; i < 1000; ++i)
    {
        if (token.stop_requested())
            co_return;  // Exit gracefully on cancellation
        co_await process_chunk(ex, i);
    }
}
```

4.5 How Does a Coroutine Start?

Two basic functions are needed to launch coroutine chains, and authors can define their own custom launch functions to suit their needs.

`run_async` — launch from callbacks, main(), event handlers, top level of a coroutine chain.

This uses a two-call syntax where the first call captures context and returns a wrapper. The executor parameter is required. The remaining parameters are optional:

- `std::stop_token` to propagate cancelation signals
- `alloc` used to allocate all frames in the coroutine chain
- `h1` , invoked with the task's value at final suspend
- `h2` , invoked with `std::exception_ptr` on exception

```
// Basic: executor only
run_async( ex )( my_task() );

// Full: executor, stop_token, allocator, success handler, error handler
run_async( ex, st, alloc, h1, h2 )( my_task() );

// Example with handlers
run_async( ioc.get_executor(), source.get_token(),
    [](int result) { std::cout << "Got: " << result << "\n"; },
    [](std::exception_ptr ep) { /* handle error */ }
)( compute_value() );
```

While the syntax is unfortunate, it is *the only way* given the timing constraints of frame allocation. And hey, its better than callback hell. What makes this possible is a small but consequential change in C++17: guaranteed evaluation order for postfix expressions. The standard now specifies:

> *"The postfix-expression is sequenced before each expression in the expression-list and any default argument." — [expr.call]*

In `run_async(ex)(my_task())` , the outer postfix-expression `run_async(ex)` is fully evaluated—returning a wrapper that allocates the trampoline coroutine—before `my_task()` is invoked. This guarantees LIFO destruction order: the trampoline is allocated BEFORE the task and serves as the task's continuation.

`run` — switching executors or customizing context within coroutines.

This binds a child task to a different executor while returning to the caller's executor on completion. Like `run_async` , it uses the two-call syntax to ensure proper allocation ordering:

```
task<void> parent()
{
    // Child runs on worker_ex, but completion returns here
    int result = co_await run( worker_ex )( compute_on_worker() );
}
```

The executor is stored by value in the awaitable's frame, keeping it alive for the operation's duration. Additionally, `run` provides overloads without an executor parameter that inherit the caller's executor while customizing stop_token or allocator:

```
task<void> cancellable()
{
    std::stop_source source;
    // Child inherits caller's executor, but uses a different stop_token
    co_await run( source.get_token() )( subtask() );
}
```

4.6 Implementing a Launcher

A launch function (e.g., `run_async`, `run`) bridges non-coroutine code into the coroutine world or performs executor hopping within a coroutine chain. Launch functions are constrained on *IoLaunchableTask* to work with any conforming task type:

```
template<Executor Ex, class... Args>
unspecified run_async( Ex ex, Args&&... args );  // returns wrapper, caller invokes with
        task

template<Executor Ex, class... Args>
unspecified run( Ex ex, Args&&... args );        // returns wrapper for co_await
```

Requirements:

1. Accept or provide an executor
2. Accept or default a stop token
3. Set thread-local allocator before invoking the child coroutine
4. Bootstrap context via `set_executor` and `set_stop_token` on the promise
5. Manage the task lifetime via `handle()` and `release()`
6. Handle completion via `exception()` and `result()` on the promise

Example implementation sketch:

```
template<Executor Ex, IoLaunchableTask Task>
void run_async( Ex ex, std::stop_token token, Task task )
{   // caller responsible for extending lifetime
    auto& promise = task.handle().promise();

    // Bootstrap context directly into the promise
    promise.set_executor( ex );
    promise.set_stop_token( token );
    promise.set_continuation( /* completion handler */, ex );

    // Transfer ownership and start execution
    task.release();
    ex.post( task.handle() );
}
```

> ***Non-normative note.*** *This simplified example has the allocator ordering problem described in §6.1: the task's frame is allocated before `run_async` is called, so any thread-local allocator setup would arrive too late. A correct implementation uses the two-call syntax shown in §4.5—`run_async(ex)(my_task())`—where the first call returns a wrapper that sets up the allocator before the task expression is evaluated. A complete implementation is beyond the scope of this example.*

Because launch functions are constrained on the concept rather than a concrete type, they work with any conforming task implementation. This decoupling enables library authors to write launch utilities that interoperate with user-defined task types.

4.7 Different Design Tradeoffs

Coroutine frame allocation happens *before* the coroutine body executes. The allocator must be known at invocation, not discovered later through receiver queries. This is the fundamental tension between `std::execution`'s backward query model and coroutine-based I/O.

P2300 (https://wg21.link/p2300) acknowledges this timing in its "Dependently-typed senders" section:

> *"In the sender/receiver model, as with coroutines, contextual information about the current execution is most naturally propagated from the consumer to the producer. In coroutines, that means information like stop tokens, allocators and schedulers are propagated from the calling coroutine to the callee. In sender/receiver, that means that that contextual information is associated with the receiver and is queried by the sender and/or operation state **after** the sender and the receiver are `connect`-ed."*

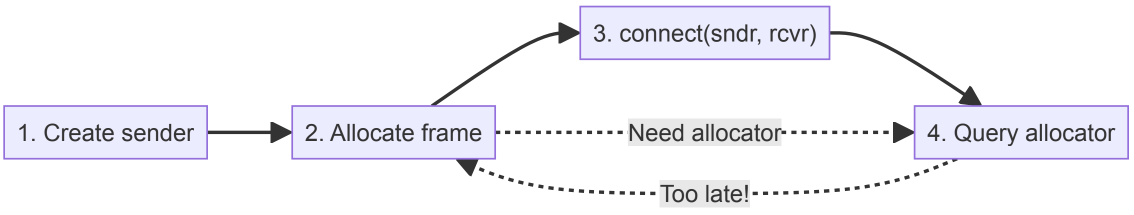
The consequence is stated plainly:

> *"The implication of the above is that the sender alone does not have all the information about the async computation it will ultimately initiate; some of that information is provided late via the receiver."*

This "late" information includes the allocator. Consider what happens with a coroutine-based sender:

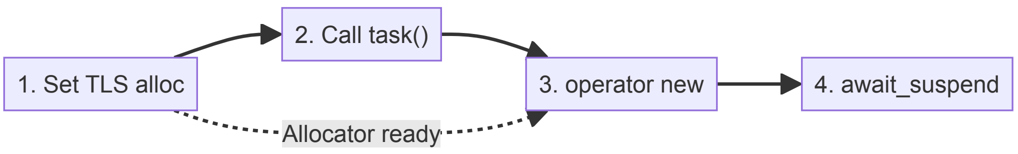
```
// P2300 query model: allocator discovered AFTER connect
template<receiver Rcvr>
struct my_operation_state {
    Rcvr rcvr_;

    void start() & noexcept {
        // Allocator available HERE, via receiver query...
        auto alloc = get_allocator(get_env(rcvr_));
        // ...but coroutine frame was allocated BEFORE connect() returned
    }
};

// The sender is created first, THEN connected:
task<int> async_work();              // Frame allocated NOW
auto sndr = async_work();
auto op = connect(sndr, receiver);   // Allocator available NOW—too late
start(op);
```

The timing mismatch is fundamental:



The frame is allocated at invocation; the allocator isn't queryable until `connect`. No query machinery can retroactively fix an allocation that already happened.

Our forward flow model solves this:



The same timing constraint applies to stop tokens. In `std::execution`, the token is discovered via `get_stop_token(get_env(receiver))`—available only after `connect()`. In our model, the token propagates forward alongside the executor via `await_suspend`, available from the moment the coroutine begins.

4.8 Type Erasure

In `std::execution`, sender types encode the entire operation chain:

```
auto sndr = schedule(sched) | then(f) | then(g);
// Type: then_sender<then_sender<schedule_sender<Sched>, F>, G>
```

Storing these in data structures or passing them through non-templated interfaces requires type erasure—but C++26 `std::execution` provides none. Facebook's libunifex offers `any_sender_of`, but it heap-allocates every erased sender, is hard-coded to `std::exception_ptr`, and is effectively abandoned (https://github.com/facebookexperimental/libunifex/commit/c93740d3c0ee74311c442f55295cd0b9c8bd63cd). Eric Niebler acknowledged (https://github.com/facebookexperimental/libunifex/issues/244#issuecomment-810686094) in 2021 that `any_sender_of` cannot handle error types beyond `exception_ptr`. The fix remains unimplemented.

Coroutines provide structural type erasure at no additional cost:

```
std::coroutine_handle<void> h;  // Type-erased by design—just a pointer
executor_ref ex;                 // Two pointers
task<int> t;                     // Simple type, not task<int, Ex, Alloc, Token>
```

Aspect	IoAwaitable	`std::execution`
Task types	Simple `task<T>`	Encode operation chain
Standard solution	Native `coroutine_handle<>`	None in C++26
Coroutine cost	Frame only	Frame + erasure wrapper
Error types	Any (`error_code` , etc.)	Only `exception_ptr`
Conversion overhead	None	Required

`task<T>` remains `task<T>` regardless of which executor, allocator, or stop token it uses. Context propagates through type-erased channels, not template parameters.

5. Executor concept

Terminology note. We use the term *Executor* rather than *scheduler* intentionally. In `std::execution`, schedulers are designed for heterogeneous computing—selecting GPU vs CPU algorithms, managing completion domains, and dispatching to hardware accelerators. Networking has different needs: strand serialization, I/O completion contexts, and thread affinity. By using *executor*, we signal a distinct concept tailored to networking's requirements. This terminology also honors Christopher Kohlhoff's executor model in Boost.Asio, which established the foundation for modern C++ asynchronous I/O.

```
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, std::coroutine_handle<> h ) {
        { 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;

        // Work submission
        { ce.dispatch( h ) } -> std::convertible_to< std::coroutine_handle<> >;
        { ce.post(h) };
    };
```

Executors are lightweight, copyable handles to execution contexts. Users often provide custom executor types tailored to application needs—priority scheduling, per-connection strand serialization, or specialized logging and instrumentation. An execution model must respect these customizations. It must also support executor composition: wrapping one executor with another. The `strand` we provide, for example, wraps an I/O context's executor to add serialization guarantees without changing the underlying dispatch mechanism.

C++20 coroutines provide type erasure *by construction*—but not through the handle type. `std::coroutine_handle<void>` and `std::coroutine_handle<promise_type>` are both just pointers with identical overhead. The erasure that matters is *structural*:

1. **The frame is opaque**: Callers see only a handle, not the promise's layout
2. **The return type is uniform**: All coroutines returning `task` have the same type, regardless of body
3. **Suspension points are hidden**: The caller doesn't know where the coroutine may suspend

This structural erasure is often lamented as overhead, but we recognize it as opportunity: *the allocation we cannot avoid can pay for the type erasure we need*. In our model, executor type-erasure happens late; only after the API has locked in the executor choice. Executor types are fully preserved at call sites even though they're type-erased internally. This enables zero-overhead composition at the API boundary while maintaining uniform internal representation.

5.1 Dispatch

`dispatch` schedules a coroutine handle for resumption. If the caller is already in the executor's context, the implementation may resume inline; otherwise, the handle is queued.

Unlike general-purpose executors that accept templated callables, `dispatch` takes only `std::coroutine_handle<>`—this is a coroutine-only model. A coroutine handle is a simple pointer: no allocation, no type erasure overhead, no virtual dispatch.

When an I/O context thread dequeues a completion via `epoll_wait`, `GetQueuedCompletionStatus`, or `io_uring_wait_cqe`, it calls `dispatch` to resume the waiting coroutine. The return value enables symmetric transfer: rather than recursively calling `resume()`, the caller returns the handle to the coroutine machinery for a tail call, preventing stack overflow.

Some contexts prohibit inline execution. A strand currently executing work cannot dispatch inline without breaking serialization— `dispatch` then behaves like `post`, queuing unconditionally.

5.2 Post

`post` queues work for later execution. Unlike `dispatch`, it never executes inline—the work item is always enqueued, and `post` returns immediately.

Use `post` for: - New work that is not a continuation of the current operation - **Breaking call chains** to bound stack depth - **Safety under locks**—posting while holding a mutex avoids deadlock risk from inline execution

5.3 The `execution_context`

An executor's `context()` function returns a reference to the `execution_context`, the proposed base class for any object that runs work (often containing the platform reactor or event loop). I/O objects coordinate global state here. Implementations install services—singletons with well-defined shutdown and destruction ordering for safe resource release. This design borrows heavily from Boost.Asio.

```cpp
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> bool has_service() const noexcept;
    template<class T> T* find_service() const noexcept;
    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;

    template<class Allocator>
        requires (!std::is_pointer_v<Allocator>)
    void set_frame_allocator( Allocator const& a );

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

Derived classes can provide: - **Platform reactor**: epoll, IOCP, io_uring, or kqueue integration - **Supporting singletons**: Timer queues, resolver services, signal handlers - **Orderly shutdown**: `stop()` and `join()` for graceful termination - **Work tracking**: `on_work_started()` / `on_work_finished()` for run-until-idle semantics - **Threads**: for example `thread_pool`.

I/O objects hold a reference to their execution context, and do not have an associated executor. A socket needs the context to register with the reactor; the executor alone cannot provide this.

Frame Allocator

The `execution_context` provides `set_frame_allocator` and `get_frame_allocator` as customization points for launchers when no allocator is specified at the launch site. Since every launcher requires an *Executor*, the execution context naturally coordinates frame allocation policy. The default allocator can optimize for speed using recycling with thread-local pools, or for economy on constrained platforms. Using `std::pmr::memory_resource*` allows implementations to change the default without breaking ABI. Applications can set a policy once via `set_frame_allocator`, and all coroutines launched with the default will use it—including those in foreign libraries, without propagating allocator template parameters or recompiling.

5.4 Comparison

Aspect	Executor	Scheduler (`std::execution`)
Purpose	I/O completion, serialization	Algorithm dispatch, GPU/CPU
Context discovery	Forward (at `co_await`)	Backward (query receiver)
Allocation	Early (before frame)	Late (at `connect()`)
Type erasure	Structural (coroutine handles)	Explicit (`any_sender`)
Operations	`dispatch`, `post`	`schedule`, `transfer`

5.5 ExecutionContext concept

While `execution_context` serves as a base class for contexts that manage I/O objects and services, concrete execution contexts that can launch coroutines must also provide an associated executor. The `ExecutionContext` concept captures this requirement: a type must derive from `execution_context`, expose an `executor_type` that satisfies `Executor`, and provide `get_executor()` to obtain an executor bound to the context.

```
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>;
    };
```

The concept formalizes the relationship between execution contexts and their executors. Types like `io_context` and `thread_pool` satisfy `ExecutionContext`—they derive from `execution_context` for service management, and they provide executors for dispatching coroutines:

```
io_context ioc;
auto ex = ioc.get_executor();   // io_context::executor_type
run_async(ex)(my_task());       // Launch coroutine on this context
```

The destructor semantics are also significant: when an `ExecutionContext` is destroyed, all unexecuted function objects that were submitted via an associated executor are also destroyed. This ensures orderly cleanup—work queued but not yet executed does not leak or outlive its context.

6. The Allocator

Achieving high performance levels with coroutines demands allocator customization, yet allocator propagation presents a unique challenge. Unlike executors and stop tokens, which can be injected at suspension points via `await_transform`, the allocator must be available *before* the coroutine frame exists. This section examines why standard approaches fail and presents our solution.

6.1 The Timing Constraint

Coroutine frame allocation has a fundamental timing constraint: `operator new` executes before the coroutine body. When a coroutine is called, the compiler allocates the frame first, then begins execution. Any mechanism that injects context later—receiver connection, `await_transform`, explicit method calls—arrives too late.

```
auto t = my_coro(sock);  // operator new called HERE
co_await t;              // await_transform kicks in HERE (too late)

spawn( my_coro(sock) );  // my_coro(sock) evaluated BEFORE calling spawn (too late)
```

6.2 The Awkward Approach

C++ provides exactly one hook at the right time: `promise_type::operator new`. The compiler passes coroutine arguments directly to this overload, allowing the promise to inspect parameters and select an allocator. The standard pattern uses `std::allocator_arg_t` as a tag to mark the allocator parameter:

```
// Free function: allocator intrudes on the parameter list
task<int> fetch_data( std::allocator_arg_t, MyAllocator alloc,
                      socket& sock, buffer& buf ) { ... }

// Member function: same intrusion
task<void> Connection::process( std::allocator_arg_t, MyAllocator alloc,
                                request const& req) { ... }
```

The promise type must provide multiple `operator new` overloads to handle both cases:

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

    // For member functions (this is first arg)
    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 approach works, but it violates encapsulation. The coroutine's parameter list—which should describe the algorithm's interface—is polluted with allocation machinery unrelated to its purpose. A function that fetches data from a socket shouldn't need to know or care about memory policy. Worse, every coroutine in a call chain must thread the allocator through its signature, even if it never uses it directly. The allocator becomes viral, infecting interfaces throughout the codebase.

6.3 Our Solution: Thread-Local Propagation

Thread-local propagation is the only approach that maintains clean interfaces while respecting the timing constraint. The premise is simple: **allocator customization happens at launch sites**, not within coroutine algorithms. Functions like `run_async` and `run` accept allocator parameters because they represent application policy decisions. Coroutine algorithms don't need to "allocator-hop"—they simply inherit whatever allocator the application has established.

Our approach:

1. **Receive the allocator at launch time.** The launch site (`run_async` , `run`) accepts a fully-typed *Allocator* parameter, or a `std::pmr::memory_resource*` at the caller's discretion.

2. **Type-erase it.** Typed allocators are stored as `std::pmr::memory_resource*` , providing a uniform interface for all downstream coroutines.

3. **Maintain lifetime via frame extension.** The allocator lives in the launch coroutine's frame. Because coroutine parameter lifetimes extend until final suspension, the allocator remains valid for the entire operation chain.

4. **Propagate through thread-locals.** Before any child coroutine is invoked, the current allocator is set in TLS. The child's `promise_type::operator new` reads it. This is an example implementation (non-normative):

```
// Thread-local accessor (returns reference to enable setting)
inline std::pmr::memory_resource*&
current_frame_allocator() noexcept {
    static thread_local std::pmr::memory_resource* mr = nullptr;
    return mr;
}

// In promise_type::operator new
static void* operator new( std::size_t size ) {
    auto* mr = current_frame_allocator();
    if(!mr)
        mr = std::pmr::get_default_resource();

    // Store allocator pointer at end of frame for correct deallocation
    std::size_t total = size + sizeof(std::pmr::memory_resource*);
    void* raw = mr->allocate(total, alignof(std::max_align_t));
    *reinterpret_cast<std::pmr::memory_resource**>(
        static_cast<char*>(raw) + size) = mr;
    return raw;
}

static void operator delete( void* ptr, std::size_t size ) {
    // Read the allocator pointer from the end of the frame
    auto* mr = *reinterpret_cast<std::pmr::memory_resource**>(
        static_cast<char*>(ptr) + size);
    std::size_t total = size + sizeof(std::pmr::memory_resource*);
    mr->deallocate(ptr, total, alignof(std::max_align_t));
}
```

This design keeps allocator policy where it belongs—at the application layer—while coroutine algorithms remain blissfully unaware of memory strategy. The propagation happens during what we call "the window": a narrow interval of execution where the correct state is guaranteed in thread-locals.

6.4 The Window

Thread-local propagation relies on a narrow, deterministic execution window. Consider:

```
task<void> parent() {          // parent is RUNNING here
    co_await child();          // child() called while parent is running
}
```

When `child()` is called: 1. `parent` coroutine is **actively executing** (not suspended) 2. `child()` 's `operator new` is called 3. `child()` 's frame is created 4. `child()` returns task 5. THEN `parent` suspends

The window is the period while the parent coroutine body executes. If `parent` sets TLS when it resumes and `child()` is called during that execution, `child` 's `operator new` sees the correct TLS value.

TLS remains valid between `await_suspend` and `await_resume` :

```
auto initial_suspend() noexcept {
    struct awaiter {
        promise_type* p_;
        bool await_ready() const noexcept { return false; }
        void await_suspend(coro) const noexcept {
            // Capture TLS allocator while it's still valid
            p_->set_frame_allocator( current_frame_allocator() );
        }
        void await_resume() const noexcept {
            // Restore TLS when body starts executing
            if( p_->frame_allocator() )
                current_frame_allocator() = p_->frame_allocator();
        }
    };
    return awaiter{this};
}
```

Every time the coroutine resumes (after any `co_await`), it sets TLS to its allocator. When `child()` is called, TLS is already pointing to `parent` 's allocator. The flow:

```
parent resumes → TLS = parent.alloc
    ↓
parent calls child()
    ↓
child operator new → reads TLS → uses parent.alloc
    ↓
child created, returns task
    ↓
parent's await_suspend → parent suspends
    ↓
child resumes → TLS = child.alloc (inherited value)
    ↓
child calls grandchild() → grandchild uses TLS
```

This is safe because: - TLS is only read in `operator new` - TLS is set by the currently-running coroutine - Single-threaded: only one coroutine runs at a time per thread - No dangling: the coroutine that set TLS is still on the stack when `operator new` reads it

7. Comparing Design Approaches

Integrating with `std::execution` imposes a complexity tax on I/O libraries whether they benefit from the framework or not.

7.1 Integration Considerations

Networking code that participates in the sender/receiver ecosystem must implement `get_env`, `get_domain`, `get_completion_scheduler`, and satisfy sender/receiver concepts—even when only returning defaults. A socket returning `default_domain` still runs through the P3826 (https://wg21.link/p3826) dispatch protocol, finds no customization, and falls through to the default.

7.2 Type Leakage Through connect_result_t

The sender/receiver model's compile-time call graph enables real optimization for GPU workloads: inlining kernel launches, eliminating intermediate buffers, selecting memory transfer strategies. For networking, where I/O latency dwarfs dispatch overhead by orders of magnitude, this visibility provides no benefit.

Yet the model requires type visibility regardless:

```
// From P2300: operation state types leak into composed operations
template<class S, class R>
struct _retry_op {
    using _child_op_t = stdexec::connect_result_t<S&, _retry_receiver<S, R>>;
    optional<_child_op_t> o_;  // Nested operation state, fully typed
};
```

For networking, this creates the template tax discussed in §3.1—N×M instantiations, compile time growth, implementation details exposed at every API boundary—without corresponding optimization payoff. Our design achieves zero type leakage; composed algorithms expose only concrete *Task* return types.

7.3 The Core Question

P2300 (https://wg21.link/p2300)/P3826 (https://wg21.link/p3826) don't break networking code—defaults work. The question is whether networking should pay for abstractions it doesn't use.

Abstraction	Networking Need	GPU/Parallel Need
Domain-based dispatch	None	Critical
Completion scheduler queries	Unused	Required
Sender transforms	Pass-through only	Algorithm selection
Typed operation state	ABI liability	Optimization opportunity

A simpler, networking-native design achieves the same goals without the tax.

8. Miscellaneous

This section is non-normative and demonstrates some aspects which may be required by implementors.

9.1 The `io_awaitable_support` Mixin

This utility simplifies promise type implementation by providing all machinery required for *IoAwaitableTask* compliance:

```
template<typename Derived>
class io_awaitable_support
{
    executor_ref executor_;
    std::stop_token stop_token_;
    std::pmr::memory_resource* alloc_ = nullptr;
    executor_ref caller_ex_;
    coro cont_;

public:
    static void* operator new( std::size_t size );
    static void operator delete( void* ptr, std::size_t size );

    void set_frame_allocator( std::pmr::memory_resource* alloc ) noexcept;
    std::pmr::memory_resource* frame_allocator() const noexcept;

    void set_continuation( coro cont, executor_ref caller_ex ) noexcept;
    coro complete() const noexcept;

    void set_stop_token( std::stop_token token ) noexcept;
    std::stop_token const& stop_token() const noexcept;

    void set_executor( executor_ref ex ) noexcept;
    executor_ref executor() const noexcept;

    template<typename A>
    decltype(auto) transform_awaitable( A&& a );

    template<typename T>
    auto await_transform( T&& t );
};
```

Promise types inherit from this mixin to gain:

- Frame allocation: `operator new` / `delete` using the thread-local frame allocator, with the allocator pointer stored at the end of each frame for correct deallocation
- Frame allocator storage: `set_frame_allocator` / `frame_allocator` for propagation to child tasks
- Continuation support: `set_continuation` / `complete` implementing the same-executor optimization
- Stop token storage: `set_stop_token` / `stop_token` for cancellation propagation
- Executor storage: `set_executor` / `executor` for executor affinity
- Awaitable transformation: `await_transform` intercepts `get_stop_token_tag` and `get_executor_tag`, delegating all other awaitables to `transform_awaitable`

The `await_transform` method uses `if constexpr` to dispatch tag types to immediate awaiters (where `await_ready()` returns `true`), enabling `co_await get_executor()` and `co_await get_stop_token()` without suspension. Other awaitables pass through to `transform_awaitable`, which derived classes can override to add custom transformation logic.

> ***Non-normative note.*** *Derived promise types that need additional `await_transform` overloads should override `transform_awaitable` rather than `await_transform` itself. Defining `await_transform` in the derived class shadows the base class version, silently breaking `this_coro::executor` and `this_coro::stop_token` support. If a separate `await_transform` overload is truly necessary, import the base class overloads with a using-declaration:*
>
> ```
> struct promise_type : io_awaitable_support<promise_type>
> {
>     using io_awaitable_support<promise_type>::await_transform;
>     auto await_transform(my_custom_type&& t); // additional overload
> };
> ```

This mixin encapsulates the boilerplate that every *IoLaunchableTask*-compatible promise type would otherwise duplicate.

9. Conclusion

We have presented an execution model designed from the ground up for coroutine-driven asynchronous I/O:

1. **Minimal executor abstraction**: Two operations— `dispatch` and `post` —suffice for I/O workloads.
2. **Clear responsibility model**: The application decides execution, allocation, and stop policy.
3. **Complete type hiding**: Executor types do not leak into public interfaces. Platform I/O types remain hidden in translation units. Composed algorithms expose only concrete Task return types. This directly enables ABI stability.
4. **Forward context propagation**: Execution context flows with control flow, not against it. No backward queries. *IoAwaitableTask* injects context through `await_transform`.
5. **Conscious tradeoff**: One pointer indirection per I/O operation (~1-2 nanoseconds) buys encapsulation, ABI stability, and fast compilation. For I/O-bound workloads where operations take 10,000+ nanoseconds, this cost is negligible.
6. **Borrows from existing practice**: Our design is heavily inspired by Boost.Asio. It gets most things right.

The comparison with `std::execution` (§3.4, §7) is instructive: that framework's complexity serves GPU workloads, not networking. The number of papers in flight for `std::execution` reflects the complexity inherent in its ambitious scope. This complexity introduces risk for networking: as design attention focuses on GPU and parallel workloads, I/O-specific requirements may become secondary considerations. P3826 (https://wg21.link/p3826) adds machinery that provides limited benefit for networking: domain-based algorithm dispatch, completion scheduler queries, sender transforms. Our design sidesteps these issues entirely.

In October 2021, LEWG polled:

> *"Networking should be based on the sender/receiver model" — Weak consensus in favor*

This direction was reasonable given the information available at the time. However, the evidence presented in this paper—the timing incompatibility with coroutine allocation, the forward-vs-backward context flow mismatch, the complexity overhead without corresponding benefit for I/O—suggests the committee may wish to revisit this guidance. A "weak consensus" is not a mandate; it is an invitation to reconsider as new information emerges. The future of asynchronous C++ need not be a single universal abstraction—purpose-built frameworks can excel at their primary use cases while remaining interoperable at the boundaries.

10. Closing Thoughts

A reference implementation of this protocol exists as a complete library: Capy (https://github.com/cppalliance/capy). It is also the foundation for the Corosio (https://github.com/cppalliance/corosio) library which offers sockets, timers, signals, DNS resolution, and integration on multiple platforms. A self-contained demonstration of the protocol is available on Compiler Explorer (https://godbolt.org/z/GPrvxfEGh). These libraries arose from use-case-first driven development with a simple mandate: produce a networking library built only for coroutines. Every design decision: forward context propagation, type-erased executors, the thread-local allocation window, emerged from solving real problems in production I/O code.

The future of C++ depends less on papers and more on practitioners who ship working code. Open source library authors are the true pioneers—they discover what works by building systems that people actually use. Standards should follow implementations, not the reverse. The *IoAwaitable* protocol is offered in that spirit: not as a theoretical construct, but as a distillation of patterns proven in practice.

11. Wording

> ***Non-normative note.*** *The wording below is not primarily intended for standardization. Its purpose is to demonstrate how a networking-focused, use-case-first design produces a dramatically leaner specification footprint. Compare this compact specification against the machinery required by P2300/P3826—domains, completion schedulers, sender transforms, query protocols—and observe how much simpler an execution model becomes when designed specifically for I/O workloads.*

12.1 Header `<io_awaitable>` synopsis [ioawait.syn]

```
namespace std {
  // [ioawait.concepts], concepts
  template<class A> concept io_awaitable = see-below;
  template<class T> concept io_awaitable_task = see-below;
  template<class T> concept io_launchable_task = see-below;
  template<class E> concept executor = see-below;
  template<class X> concept execution_context = see-below;

  // [ioawait.execref], class executor_ref
  class executor_ref;

  // [ioawait.execctx], class execution_context
  class execution_context;

  // [ioawait.launch], launch functions
  template<executor Ex, class... Args>
    unspecified run_async(Ex const& ex, Args&&... args);

  template<executor Ex, class... Args>
    unspecified run(Ex const& ex, Args&&... args);

  template<class... Args>
    unspecified run(Args&&... args);  // inherits caller's executor

  // [ioawait.thiscoro], namespace this_coro
  namespace this_coro {
    struct executor_tag {};
    struct stop_token_tag {};
    inline constexpr executor_tag executor{};
    inline constexpr stop_token_tag stop_token{};
  }
}
```

12.2 Concepts [ioawait.concepts]

12.2.1 Concept `io_awaitable` [ioawait.concepts.awaitable]

```
template<class A>
concept io_awaitable =
  requires(A a, coroutine_handle<> h, executor_ref ex, stop_token token) {
    a.await_suspend(h, ex, token);
  };
```

1 A type `A` meets the `io_awaitable` requirements if it satisfies the syntactic requirements above and the semantic requirements below.

2 In Table 1, `a` denotes a value of type `A` , `h` denotes a value of type `coroutine_handle<>` representing the calling coroutine, `ex` denotes a value of type `executor_ref` , and `token` denotes a value of type `stop_token` .

Table 1 — io_awaitable requirements

expression	return type	assertion/note pre/post-conditions
`a.await_suspend(h, ex, token)`	`void` , `bool` , or `coroutine_handle<>`	*Effects:* Initiates the asynchronous operation represented by `a` . The executor `ex` and stop token `token` are propagated to the operation. If the return type is `coroutine_handle<>` , the returned handle is suitable for symmetric transfer. *Preconditions:* `h` is a suspended coroutine. `ex` refers to a valid executor. *Synchronization:* The call to `await_suspend` synchronizes with the resumption of `h` or any coroutine to which control is transferred.

3 [*Note:* The three-argument `await_suspend` signature distinguishes `io_awaitable` types from standard awaitables. A compliant coroutine's `await_transform` calls this signature, enabling static detection of protocol mismatches at compile time. — *end note*]

12.2.2 Concept `io_awaitable_task` [ioawait.concepts.task]

```
template<class T>
concept io_awaitable_task =
  io_awaitable<T> &&
  requires { typename T::promise_type; } &&
  requires(typename T::promise_type& p,
           typename T::promise_type const& cp,
           executor_ref ex, stop_token st, coroutine_handle<> cont) {
    { p.set_executor(ex) } noexcept;
    { p.set_stop_token(st) } noexcept;
    { p.set_continuation(cont, ex) } noexcept;
    { cp.executor() } noexcept -> same_as<executor_ref>;
    { cp.stop_token() } noexcept -> same_as<stop_token const&>;
    { cp.complete() } noexcept -> same_as<coroutine_handle<>>;
  };
```

1 A type `T` meets the `io_awaitable_task` requirements if it satisfies `io_awaitable<T>`, has a nested type `promise_type`, and the promise type satisfies the semantic requirements below.

2 In Table 2, `p` denotes an lvalue of type `typename T::promise_type`, `cp` denotes a const lvalue of type `typename T::promise_type`, `ex` denotes a value of type `executor_ref`, `caller_ex` denotes a value of type `executor_ref` representing the caller's executor, `st` denotes a value of type `stop_token`, and `cont` denotes a value of type `coroutine_handle<>`.

Table 2 — io_awaitable_task promise requirements

expression	return type	assertion/note pre/post-conditions
`p.set_executor(ex)`	`void`	*Effects:* Stores `ex` as the executor associated with this coroutine. Shall not exit via an exception. *Postconditions:* `p.executor() == ex`.
`p.set_stop_token(st)`	`void`	*Effects:* Stores `st` as the stop token associated with this coroutine. Shall not exit via an exception. *Postconditions:* `p.stop_token()` returns a reference to the stored token.
`p.set_continuation(cont, caller_ex)`	`void`	*Effects:* Stores `cont` as the continuation to resume when this coroutine completes, and `caller_ex` as the executor on which to resume it. Shall not exit via an exception.
`cp.executor()`	`executor_ref`	*Returns:* The executor previously set via `set_executor`. Shall not exit via an exception. *Preconditions:* `set_executor` has been called.
`cp.stop_token()`	`stop_token const&`	*Returns:* A reference to the stop token previously set via `set_stop_token`. Shall not exit via an exception. *Preconditions:* `set_stop_token` has been called.
`cp.complete()`	`coroutine_handle<>`	*Returns:* A coroutine handle for symmetric transfer at final suspension. If no continuation was set, returns `noop_coroutine()`. If `cp.executor() == caller_ex`, returns `cont` directly (same-executor optimization). Otherwise, dispatches `cont` through `caller_ex` and returns the result. Shall not exit via an exception.

3 The promise's `await_transform` shall intercept child awaitables and inject the current executor and stop token via the three-argument `await_suspend` signature.

4 [*Note:* The `set_continuation` function receives both the continuation handle and the caller's executor to enable the same-executor optimization: when executors match, `complete()` returns the continuation directly for zero-overhead symmetric transfer. — *end note*]

12.2.3 Concept `io_launchable_task` [ioawait.concepts.launch]

```
template<class T>
concept io_launchable_task =
  io_awaitable_task<T> &&
  requires(T& t, T const& ct, typename T::promise_type const& cp) {
    { ct.handle() } noexcept -> same_as<coroutine_handle<typename T::promise_type>>;
    { cp.exception() } noexcept -> same_as<exception_ptr>;
    { t.release() } noexcept;
  } &&
  (is_void_v<decltype(declval<T&>().await_resume())> ||
   requires(typename T::promise_type& p) { p.result(); });
```

1 A type `T` meets the `io_launchable_task` requirements if it satisfies `io_awaitable_task<T>` and the additional semantic requirements below.

2 In Table 3, `t` denotes an lvalue of type `T`, `ct` denotes a const lvalue of type `T`, `cp` denotes a const lvalue of type `typename T::promise_type`, and `p` denotes an lvalue of type `typename T::promise_type`.

Table 3 — io_launchable_task requirements

expression	return type	assertion/note pre/post-conditions
`ct.handle()`	`coroutine_handle<typename T::promise_type>`	*Returns:* The typed coroutine handle for this task. Shall not exit via an exception.
`cp.exception()`	`exception_ptr`	*Returns:* The exception captured during coroutine execution, or a null `exception_ptr` if no exception occurred. Shall not exit via an exception.
`t.release()`	`void`	*Effects:* Releases ownership of the coroutine frame. After this call, the task object no longer destroys the frame upon destruction. Shall not exit via an exception. *Postconditions:* The task object is in a moved-from state.
`p.result()`	*unspecified*	*Returns:* The result value stored in the promise. *Preconditions:* The coroutine completed with a value (not an exception). *Remarks:* This expression is only required when `await_resume()` returns a non-void type.

3 [*Note:* The `handle()` and `release()` methods enable launch functions to manage task lifetime directly. After `release()`, the launch function assumes responsibility for destroying the coroutine frame. — *end note*]

12.2.4 Concept `executor` [ioawait.concepts.executor]

```
template<class E>
concept executor =
  is_nothrow_copy_constructible_v<E> &&
  is_nothrow_move_constructible_v<E> &&
  requires(E& e, E const& ce, E const& ce2, coroutine_handle<> h) {
    { ce == ce2 } noexcept -> convertible_to<bool>;
    { ce.context() } noexcept -> see-below;
    { ce.on_work_started() } noexcept;
    { ce.on_work_finished() } noexcept;
    { ce.dispatch(h) };
    { ce.post(h) };
  };
```

1 A type `E` meets the `executor` requirements if it is nothrow copy and move constructible, and satisfies the semantic requirements below.

2 No comparison operator, copy operation, move operation, swap operation, or member functions `context`, `on_work_started`, and `on_work_finished` on these types shall exit via an exception.

3 The executor copy constructor, comparison operators, and other member functions defined in these requirements shall not introduce data races as a result of concurrent calls to those functions from different threads. The member function `dispatch` may be recursively reentered.

4 Let `ctx` be the execution context returned by the executor's `context()` member function. An executor becomes invalid when the first call to `ctx.shutdown()` returns. The effect of calling `on_work_started`, `on_work_finished`, `dispatch`, or `post` on an invalid executor is undefined. [*Note:* The copy constructor, comparison operators, and `context()` member function continue to remain valid until `ctx` is destroyed. — *end note*]

5 In Table 4, `x1` and `x2` denote (possibly const) values of type `E`, `mx1` denotes an xvalue of type `E`, `h` denotes a value of type `coroutine_handle<>`, and `u` denotes an identifier.

Table 4 — executor requirements

expression	return type	assertion/note pre/post-conditions
`E u(x1);`		Shall not exit via an exception. *Postconditions:* `u == x1` and `addressof(u.context()) == addressof(x1.context())`.
`E u(mx1);`		Shall not exit via an exception. *Postconditions:* `u` equals the prior value of `mx1` and `addressof(u.context())` equals the prior value of `addressof(mx1.context())`.
`x1 == x2`	`bool`	*Returns:* `true` only if `x1` and `x2` can be interchanged with identical effects in any of the expressions defined in these type requirements. [*Note:* Returning `false` does not necessarily imply that the effects are not identical. — *end note*] `operator==` shall be reflexive, symmetric, and transitive, and shall not exit via an exception.
`x1 != x2`	`bool`	Same as `!(x1 == x2)`.
`x1.context()`	`execution_context&`, or `C&` where `C` is publicly derived from `execution_context`	Shall not exit via an exception. The comparison operators and member functions defined in these requirements shall not alter the reference returned by this function.
`x1.on_work_started()`	`void`	Shall not exit via an exception.
`x1.on_work_finished()`	`void`	Shall not exit via an exception. *Preconditions:* A preceding call `x2.on_work_started()` where `x1 == x2`.
`x1.dispatch(h)`	`void`	*Effects:* Schedules `h` for resumption. If the caller is already in the executor's context and inline execution is safe, resumes `h` immediately by calling `h.resume()`. Otherwise, queues `h` for later execution. *Synchronization:* The invocation of `dispatch` synchronizes with the resumption of `h`.
`x1.post(h)`	`void`	*Effects:* Queues `h` for later execution. The executor shall not block forward progress of the caller pending resumption of `h`. The executor shall not resume `h` before the call to `post` returns. *Synchronization:* The invocation of `post` synchronizes with the resumption of `h`.

6 [*Note:* Unlike the Networking TS executor requirements, this concept operates on `coroutine_handle<>` rather than arbitrary function objects. This restriction enables zero-allocation dispatch in the common case and leverages the structural type erasure that coroutines already provide. — *end note*]

12.2.5 Concept `execution_context` [ioawait.concepts.execctx]

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

1 A type `X` meets the `execution_context` requirements if it is publicly and unambiguously derived from `std::execution_context` , and satisfies the semantic requirements below.

2 In Table 5, `x` denotes a value of type `X` .

Table 5 — execution_context requirements

expression	return type	assertion/note pre/post-conditions
`X::executor_type`	type	A type meeting the `executor` requirements.
`x.get_executor()`	`X::executor_type`	*Returns:* An executor object that is associated with the execution context. Shall not exit via an exception.
`x.~X()`		*Effects:* Destroys all unexecuted function objects that were submitted via an executor object that is associated with the execution context.

3 [*Note:* The destructor requirement ensures orderly cleanup—work queued but not yet executed does not leak or outlive its context. Types such as `io_context` and `thread_pool` satisfy this concept. — *end note*]

12.3 Class `executor_ref` [ioawait.execref]

1 Class `executor_ref` is a type-erasing wrapper for executors satisfying the `executor` concept. It provides a uniform, non-templated interface for executor operations.

```
namespace std {
  class executor_ref {
    void const* ex_ = nullptr;                    // exposition only
    unspecified const* vt_ = nullptr;             // exposition only

  public:
    executor_ref() = default;
    executor_ref(executor_ref const&) = default;
    executor_ref& operator=(executor_ref const&) = default;

    template<executor E>
      executor_ref(E const& e) noexcept;

    explicit operator bool() const noexcept;
    bool operator==(executor_ref const&) const noexcept = default;

    execution_context& context() const noexcept;
    void on_work_started() const noexcept;
    void on_work_finished() const noexcept;
    void dispatch(coroutine_handle<> h) const;
    void post(coroutine_handle<> h) const;
  };
}
```

2 The class `executor_ref` satisfies `copy_constructible` and `equality_comparable`. Copies of an `executor_ref` refer to the same underlying executor.

3 [*Note:* At two pointers in size, `executor_ref` is designed for efficient propagation through coroutine chains. Equality comparison reduces to pointer comparison, enabling the same-executor optimization in `complete()`. — *end note*]

12.3.1 `executor_ref` constructors [ioawait.execref.cons]

```
executor_ref() = default;
```

1 *Postconditions:* `bool(*this) == false`.

```
template<executor E>
  executor_ref(E const& e) noexcept;
```

2 *Effects:* Constructs an `executor_ref` that refers to `e`.

3 *Postconditions:* `bool(*this) == true`. `addressof(context())` equals `addressof(e.context())`.

4 *Remarks:* The behavior is undefined if `e` is destroyed or becomes invalid while `*this` or any copy of `*this` still exists. [*Note:* In typical usage, the referenced executor is stored in a launch function's frame or a coroutine promise, which outlives all `executor_ref` copies propagated through the coroutine chain. — *end note*]

12.3.2 `executor_ref` observers [ioawait.execref.obs]

```
explicit operator bool() const noexcept;
```

1 *Returns:* `true` if `*this` refers to an executor, otherwise `false`.

```
bool operator==(executor_ref const& other) const noexcept;
```

2 *Returns:* `true` if `*this` and `other` were both constructed from the same executor object (by address), or if both are empty. Otherwise `false`.

3 *Remarks:* Two `executor_ref` objects constructed from different executor objects compare unequal even if those executors would compare equal via their own `operator==`.

12.3.3 `executor_ref` operations [ioawait.execref.ops]

```
execution_context& context() const noexcept;
```

1 *Preconditions:* `bool(*this) == true`.

2 *Returns:* A reference to the execution context of the referenced executor, as if by calling `e.context()` where `e` is the referenced executor.

```
void on_work_started() const noexcept;
```

3 *Preconditions:* `bool(*this) == true`.

4 *Effects:* Equivalent to `e.on_work_started()` where `e` is the referenced executor.

```
void on_work_finished() const noexcept;
```

5 *Preconditions:* `bool(*this) == true`. A preceding call to `on_work_started()` on `*this` or on an `executor_ref` that compares equal to `*this`.

6 *Effects:* Equivalent to `e.on_work_finished()` where `e` is the referenced executor.

```
void dispatch(coroutine_handle<> h) const;
```

7 *Preconditions:* `bool(*this) == true`. `h` is a valid, suspended coroutine handle.

8 *Effects:* Equivalent to `e.dispatch(h)` where `e` is the referenced executor.

9 *Synchronization:* The invocation of `dispatch` synchronizes with the resumption of `h`.

10 *Throws:* `bad_executor` if `bool(*this) == false`.

```
void post(coroutine_handle<> h) const;
```

12 *Preconditions:* `bool(*this) == true`. `h` is a valid, suspended coroutine handle.

13 *Effects:* Equivalent to `e.post(h)` where `e` is the referenced executor.

14 *Synchronization:* The invocation of `post` synchronizes with the resumption of `h`.

15 *Throws:* `bad_executor` if `bool(*this) == false`.

12.4 Class `execution_context` [ioawait.execctx]

1 Class `execution_context` is the base class for objects that manage a set of services and provide an execution environment for I/O operations. Derived classes typically provide platform-specific reactor integration (epoll, IOCP, io_uring, kqueue).

```
namespace std {
  class execution_context {
  public:
    class service;

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

    template<class T> bool has_service() const noexcept;
    template<class T> T* find_service() const noexcept;
    template<class T> T& use_service();
    template<class T, class... Args> T& make_service(Args&&... args);

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

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

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

2 Access to the services of an `execution_context` is via the function templates `use_service` , `make_service` , `find_service` , and `has_service` .

3 In a call to `use_service<Service>` , the type argument chooses a service from the set in the `execution_context` . If the service is not present, an object of type `Service` is created and added. A program can check if an `execution_context` contains a particular service with `has_service<Service>` .

4 Service objects may be explicitly added using `make_service<Service>` . If the service is already present, `make_service` throws an exception.

5 Once a service reference is obtained from an `execution_context` by calling `use_service` or `make_service` , that reference remains usable until a call to `destroy()` .

6 The functions `use_service`, `make_service`, `find_service`, and `has_service` do not introduce data races as a result of concurrent calls from different threads.

12.4.1 `execution_context` constructors and destructor [ioawait.execctx.cons]

```
execution_context();
```

1 *Effects:* Creates an object of class `execution_context` which contains no services. [*Note:* An implementation may preload services of internal service types for its own use. — *end note*]

```
~execution_context();
```

2 *Effects:* Destroys an object of class `execution_context`. Performs `shutdown()` followed by `destroy()`.

12.4.2 `execution_context` protected operations [ioawait.execctx.protected]

```
void shutdown() noexcept;
```

1 *Effects:* For each service object `svc` in the `execution_context` set, in reverse order of addition to the set, performs `svc->shutdown()`. For each service in the set, `svc->shutdown()` is called only once irrespective of the number of calls to `shutdown` on the `execution_context`.

```
void destroy() noexcept;
```

2 *Effects:* Destroys each service object in the `execution_context` set, and removes it from the set, in reverse order of addition to the set.

12.4.3 `execution_context` service access [ioawait.execctx.services]

```
template<class Service> bool has_service() const noexcept;
```

1 *Returns:* `true` if an object of type `Service` is present in `*this`, otherwise `false`.

```
template<class Service> Service* find_service() const noexcept;
```

2 *Returns:* A pointer to the service of type `Service` if present in `*this`, otherwise `nullptr`.

```
template<class Service> Service& use_service();
```

3 *Effects:* If an object of type `Service` does not already exist in the `execution_context` set, creates an object of type `Service`, initialized as `Service(*this)`, and adds it to the set.

4 *Returns:* A reference to the corresponding service of `*this`.

5 *Remarks:* The reference returned remains valid until a call to `destroy()`.

```
template<class Service, class... Args> Service& make_service(Args&&... args);
```

6 *Preconditions:* An object of type `Service` does not already exist in the `execution_context` set.

7 *Effects:* Creates an object of type `Service`, initialized as `Service(*this, forward<Args>(args)...)`, and adds it to the `execution_context` set.

8 *Returns:* A reference to the new service.

9 *Throws:* `service_already_exists` if a corresponding service object of type `Service` is already present in the set.

10 *Remarks:* The reference returned remains valid until a call to `destroy()`.

12.4.4 `execution_context` frame allocator [ioawait.execctx.alloc]

```
pmr::memory_resource* get_frame_allocator() const noexcept;
```

1 *Returns:* The memory resource set via `set_frame_allocator`, or `pmr::get_default_resource()` if none was set.

2 *Remarks:* This function provides the default allocator for coroutine frames launched with executors from this context when no allocator is specified at the launch site.

```
void set_frame_allocator(pmr::memory_resource* mr) noexcept;
```

3 *Effects:* Sets the memory resource to be returned by subsequent calls to `get_frame_allocator()`.

4 *Remarks:* This function does not affect coroutine frames that have already been allocated.

5 [*Note:* The frame allocator is a quality of implementation concern. A conforming implementation may ignore the allocator parameter entirely, and programs should still behave correctly. The allocator mechanism is provided to enable performance optimizations such as thread-local recycling pools or bounded allocation strategies, but correct program behavior must not depend on a specific allocation strategy being used. — *end note*]

12.4.5 Class `execution_context::service` [ioawait.execctx.service]

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

1 A class is a service if it is publicly and unambiguously derived from `execution_context::service`.

2 A service's `shutdown` member function shall destroy all copies of function objects that are held by the service.

12.5 Launch functions [ioawait.launch]

1 Launch functions bootstrap execution context into a coroutine chain. They are the bridge between non-coroutine code and the `io_awaitable` protocol.

12.5.1 Function template `run_async` [ioawait.launch.async]

```
template<executor Ex, class... Args>
  unspecified run_async(Ex const& ex, Args&&... args);
```

1 *Returns:* A callable object `f` such that the expression `f(task)` is valid when `task` satisfies `io_launchable_task`.

2 *Effects:* When `f(task)` is invoked:

- (2.1) Sets the thread-local frame allocator to the allocator specified in `Args`, or to `ex.context().get_frame_allocator()` if no allocator is specified.
- (2.2) Evaluates `task` (which allocates the coroutine frame using the thread-local allocator).
- (2.3) Calls `task.handle().promise().set_executor(ex)`.
- (2.4) If a `stop_token` is specified in `Args`, calls `task.handle().promise().set_stop_token(token)`.
- (2.5) Sets up completion handling: if completion handlers are specified in `Args`, arranges for them to be invoked when `task` completes.
- (2.6) Calls `task.release()` to transfer ownership.
- (2.7) Resumes the coroutine via the executor.

3 *Remarks:* `Args` may include:

- A `stop_token` to propagate cancellation signals.

- An allocator satisfying the *Allocator* requirements, or a `pmr::memory_resource*`, used to allocate all coroutine frames in the chain.
- A completion handler invoked with the task's result value upon successful completion.
- An error handler invoked with `exception_ptr` if the task completes with an exception.

4 *Synchronization:* The call to `run_async(ex, args...)(task)` synchronizes with the invocation of the completion handler (if any) and with the resumption of the task coroutine.

5 [*Note:* The two-call syntax `run_async(ex)(task())` is required because of coroutine allocation timing. The outer expression `run_async(ex)` must complete—returning the callable and establishing the thread-local allocator—before `task()` is evaluated. This ordering is guaranteed by [expr.call] in C++17 and later: "The postfix-expression is sequenced before each expression in the expression-list." — *end note*]

6 [*Example:*

```
// Basic launch
run_async(ioc.get_executor())(my_task());

// With stop token and allocator
run_async(ex, source.get_token(), my_allocator)(my_task());

// With completion handlers
run_async(ex,
    [](int result) { /* handle success */ },
    [](exception_ptr ep) { /* handle error */ }
)(compute_value());
```

— *end example*]

12.5.2 Function template `run` [ioawait.launch.run]

```
template<executor Ex, class... Args>
  unspecified run(Ex const& ex, Args&&... args);

template<class... Args>
  unspecified run(Args&&... args);
```

1 *Returns:* A callable object `f` such that the expression `f(task)` is valid when `task` satisfies `io_launchable_task`, and returns an awaitable object `a`.

2 *Effects:* When `f(task)` is invoked:

- (2.1) Sets the thread-local frame allocator to the allocator specified in `Args`, or inherits the caller's allocator if none is specified.

- (2.2) Evaluates `task` (which allocates the coroutine frame using the thread-local allocator).
- (2.3) Returns an awaitable `a` that stores the executor (if provided), stop token (if provided), and the task.

3 *Effects:* When `a` is awaited via `co_await a` :

- (3.1) The child task is bound to executor `ex` (if provided) or inherits the caller's executor.
- (3.2) The stop token from `Args` is propagated to `task` , or the caller's stop token is inherited if none is specified.
- (3.3) The child task executes on the bound executor.
- (3.4) Upon completion, the caller resumes on its original executor (via the same-executor optimization in `complete()`).
- (3.5) The result of `co_await a` is the result of `task` .

4 *Preconditions:* The expression appears in a coroutine whose promise type satisfies `io_awaitable_task` .

5 *Remarks:* `Args` may include:

- A `stop_token` to override the caller's stop token.
- An allocator satisfying the *Allocator* requirements, or a `pmr::memory_resource*` , used for coroutine frame allocation.

6 *Remarks:* When no executor is provided, the task inherits the caller's executor directly, enabling zero-overhead symmetric transfer on completion.

7 *Synchronization:* The suspension of the caller synchronizes with the resumption of `task` . The completion of `task` synchronizes with the resumption of the caller.

8 [*Note:* Like `run_async` , `run` uses the two-call syntax `run(ex)(task())` to ensure proper frame allocation ordering. — *end note*]

9 [*Example:*

```
task<int> parent() {
    // Child runs on worker_ex, but completion returns here
    int result = co_await run(worker_ex)(compute_on_worker());
    co_return result * 2;
}

task<void> with_custom_token() {
    std::stop_source source;
    // Child inherits caller's executor, uses different stop_token
    co_await run(source.get_token())(cancellable_work());
}
```

— *end example*]

12.6 Namespace `this_coro` [ioawait.thiscoro]

```
namespace std::this_coro {
  struct executor_tag {};
  struct stop_token_tag {};
  inline constexpr executor_tag executor{};
  inline constexpr stop_token_tag stop_token{};
}
```

1 The `this_coro` namespace provides tag objects that can be awaited within a coroutine to retrieve execution context information without suspension.

12.6.1 `this_coro::executor` [ioawait.thiscoro.executor]

```
inline constexpr executor_tag executor;
```

1 When awaited via `co_await this_coro::executor` inside a coroutine whose promise type satisfies `io_awaitable_task` :

2 *Returns:* The `executor_ref` bound to the current coroutine, as would be returned by `promise.executor()` .

3 *Remarks:* This operation never suspends. The promise's `await_transform` intercepts the `executor_tag` type and returns an immediate awaiter where `await_ready()` returns `true` .

4 *Preconditions:* `set_executor` has been called on the promise.

12.6.2 `this_coro::stop_token` [ioawait.thiscoro.token]

```
inline constexpr stop_token_tag stop_token;
```

1 When awaited via `co_await this_coro::stop_token` inside a coroutine whose promise type satisfies `io_awaitable_task` :

2 *Returns:* The `std::stop_token` propagated to the current coroutine, as would be returned by `promise.stop_token()` .

3 *Remarks:* This operation never suspends. The promise's `await_transform` intercepts the `stop_token_tag` type and returns an immediate awaiter where `await_ready()` returns `true` .

4 *Preconditions:* `set_stop_token` has been called on the promise.

5 [*Example:*

```
task<void> cancellable_work() {
    executor_ref ex = co_await this_coro::executor;
    stop_token token = co_await this_coro::stop_token;

    for (int i = 0; i < 1000; ++i) {
        if (token.stop_requested())
            co_return;  // Exit gracefully
        co_await process_chunk(i);
    }
}
```

— *end example*]

12.7 Threading and synchronization [ioawait.sync]

1 Unless otherwise specified, it is safe to call `const` member functions of the classes defined in this clause concurrently from multiple threads.

2 The execution context, executor, and coroutine handle types do not introduce data races when used according to their documented requirements.

3 Synchronization between asynchronous operations follows the “synchronizes with” relationship defined in [intro.multithread]:

- (3.1) A call to `executor::dispatch` or `executor::post` synchronizes with the resumption of the submitted coroutine handle.
- (3.2) The suspension of a coroutine at a `co_await` expression synchronizes with the resumption of that coroutine.
- (3.3) The completion of a child coroutine (at final suspension) synchronizes with the resumption of the parent coroutine.

4 [*Note:* These synchronization guarantees ensure that modifications made by one coroutine before suspension are visible to the code that resumes it. — *end note*]

Acknowledgements

This paper builds on the foundational work of many contributors to C++ asynchronous programming:

Chris Kohlhoff for Boost.Asio, which has served the C++ community for over two decades and established many of the patterns we build upon—and some we consciously depart from. The executor model in this paper honors his pioneering work.

Lewis Baker for his work on C++ coroutines, the Asymmetric Transfer blog series, and his contributions to P2300 (https://wg21.link/p2300) and P3826 (https://wg21.link/p3826). His explanations of symmetric transfer and coroutine optimization techniques directly informed our design.

Dietmar Kühl for P2762 (https://wg21.link/p2762) and P3552 (https://wg21.link/p3552), which explore sender/receiver networking and coroutine task types. His clear articulation of design tradeoffs—including the late-binding problem and cancellation overhead concerns—helped crystallize our understanding of where the sender model introduces friction for networking.

The analysis in this paper is not a critique of these authors' contributions, but rather an exploration of whether networking's specific requirements are best served by adapting to general-purpose abstractions or by purpose-built designs.

References

1. N4242 (https://wg21.link/n4242) — Executors and Asynchronous Operations, Revision 1 (2014)
2. N4482 (https://wg21.link/n4482) — Some notes on executors and the Networking Library Proposal (2015)
3. P2300R10 (https://wg21.link/p2300) — std::execution (Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Eric Niebler)
4. P2762R2 (https://wg21.link/p2762) — Sender/Receiver Interface for Networking (Dietmar Kühl)
5. P3552R3 (https://wg21.link/p3552) — Add a Coroutine Task Type (Dietmar Kühl, Maikel Nadolski); approved for C++26 at Sofia plenary
6. P3826R2 (https://wg21.link/p3826) — Fix or Remove Sender Algorithm Customization (Lewis Baker, Eric Niebler)
7. Boost.Asio (https://www.boost.org/doc/libs/release/doc/html/boost_asio.html) — Asynchronous I/O library (Chris Kohlhoff)
8. The C10K problem (http://www.kegel.com/c10k.html) — Scalable network programming (Dan Kegel)
9. Capy (https://github.com/cppalliance/capy) — IoAwaitable protocol implementation (Vinnie Falco, Steve Gerbino)