The Sender Sub-Language

Document Number:	D4014R0
Date:	2026-02-17
Audience:	LEWG

Reply-to:	Vinnie Falco vinnie.falco@gmail.com Mungo Gill mungo.gill@me.com

Table of Contents

Abstract

C++26 introduces a rich sub-language for asynchronous programming through `std::execution` (P2300R10). Sender pipelines replace C++'s control flow, variable binding, error handling, and iteration with library-level equivalents rooted in continuation-passing style and monadic composition. This paper is a guide to the Sender Sub-Language: its theoretical foundations, the programming model it provides, and the engineering trade-offs it makes. The trade-offs serve specific domains well. The question is whether other domains deserve the same freedom to choose the model that serves them.

Revision History

R0: March 2026 (pre-Croydon mailing)

- Initial version.

---

1. Introduction

`std::execution` (P2300R10) was formally adopted into the C++26 working paper at St. Louis in July 2024. C++ developers who write asynchronous code will likely encounter it.

C++ has a tradition of sub-languages. Template metaprogramming is one: a Turing-complete compile-time language expressed through the type system. The preprocessor is another, operating before compilation begins with its own textual substitution rules. `constexpr` evaluation is a third, running a subset of C++ at compile time to produce

values. Each operates within C++ but carries its own idioms, patterns, and mental model. ***The Design and Evolution of C++***[21] observes that C++ contains multiple programming paradigms; sub-languages are how those paradigms manifest in practice.

C++26 adds another. The Sender Sub-Language is a continuation-passing-style programming model expressed through `std::execution`, with its own control flow primitives, variable binding model, error handling system, and iteration strategy. It is a complete programming model for asynchronous computation.

Consider the simplicity of the basic unit of composition:

```
auto work = just(42) | then([](int v) { return v + 1; });
```

One value lifted into the sender context, one transformation applied through the pipe operator. The Sender Sub-Language builds from this foundation into an expressive system for describing asynchronous work, and the depth of that system is worth understanding.

This paper is a guide to the Sender Sub-Language. P4007R0 ("Senders and C++") examines the coroutine integration; this paper focuses on what the Sub-Language is, where it came from, and what it looks like in practice.

---

2. The Equivalents

The Sender Sub-Language provides equivalents for most of C++'s fundamental control flow constructs. The following table maps each C++ language feature to its Sub-Language counterpart:

Regular C++	Sender Sub-Language
Sequential statements	`|` pipe chaining
`for` / `while` loop	Recursive `let_value` with type-erased return
`if` / `else`	`let_value` returning different sender types
`switch`	Nested `let_value` with continuation selection
`try` / `catch`	`upon_error` / `let_error`
`throw`	`set_error`
`break` / `continue`	`set_stopped` / continuation selection
`return`	`set_value` into the receiver
Local variables	Lambda captures (with move semantics across boundaries)

Regular C++	Sender Sub-Language
Function call + return	`connect` + `start` + `set_value`
Structured bindings	(not needed)
Range-for	(not needed)
`if` with initializer	(not needed)

The table is largely self-explanatory. Two details bear noting. First, the iteration and branching equivalents (`repeat_effect_until` , `any_sender_of<>` , `variant_sender`) are provided by the stdexec reference implementation but are not yet part of the C++26 working paper. Section 5 illustrates both patterns with working code. Second, the last three rows - structured bindings, range-for, and `if` with initializer - have no equivalent because the Sender Sub-Language does not produce intermediate return values. Values flow forward into continuations as arguments, not backward to callers as returns.

---

3. Why It Looks Like This

The Sender Sub-Language is not merely a fluent API. It is continuation-passing style (CPS) expressed as composable value types, a technique with deep roots in the theory of computation.

Sender concept	Theoretical origin	Source
`just(x)`	Monadic `return` / `pure`	Moggi (1991)
`let_value(f)`	Monadic bind (`>>=`)	Lambda Papers (1975-1980)
`then(f)`	Functor `fmap`	Moggi (1991)
`set_value` / `set_error` / `set_stopped`	Algebraic effect channels	Danvy & Filinski (1990)
`connect(sndr, rcvr)`	CPS reification	Lambda Papers (1975-1980)
`start(op)`	CPS evaluation	
Completion signatures	Type-level sum type	Griffin (1990)

CPS makes control flow, variable binding, and resource lifetime explicit in the term structure. This is why optimizing compilers (SML/NJ, GHC, Chicken Scheme) use it as their intermediate representation, and why the Sender Sub-Language can build zero-allocation pipelines and compile-time work graphs. The names are not arbitrary: `just` echoes Haskell's `Just`, `let_value` mirrors monadic bind, and the three completion channels form a fixed algebraic effect system. The P2300 authors built a framework grounded in four decades of programming language research.

---

4. Multiple Discoveries

Eric Niebler's published writing provides the most complete public record of the design thinking behind `std::execution`. His blog posts document two independent discoveries: coroutines for direct-style async and senders for compile-time work graphs, with candor and intellectual honesty. Both discoveries were valid. The following timeline, drawn from that published record, shows how the relationship between them evolved as the problems being solved changed.

2017. Eric Niebler published "Ranges, Coroutines, and React: Early Musings on the Future of Async in C++". The vision was pure coroutines and ranges: `for co_await`, async generators, range adaptors on async streams. No senders. No receivers. The original async vision for C++ was direct-style and coroutine-native:

> *"The C++ Standard Library is already evolving in this direction, and I am working to make that happen both on the Committee and internally at Facebook."*

2020. Eric Niebler published "Structured Concurrency", as P2300R0 was being developed. Coroutines were the primary model:

> *"I think that 90% of all async code in the future should be coroutines simply for maintainability."*

> *"We sprinkle `co_await` in our code and we get to continue using all our familiar idioms: exceptions for error handling, state in local variables, destructors for releasing resources, arguments passed by value or by reference, and all the other hallmarks of good, safe, and idiomatic Modern C++."*

Senders were positioned as the optimization path for the remaining 10%:

> *"Why would anybody write that when we have coroutines? You would certainly need a good reason."*

> *"That style of programming makes a different tradeoff, however: it is far harder to write and read than the equivalent coroutine."*

2021. Eric Niebler published "Asynchronous Stacks and Scopes" during the P2300 R1/R2 revision period:

> *"The overwhelming benefit of coroutines in C++ is its ability to make your async scopes line up with lexical scopes."*

The post ended with a promise: ***"Next post, I'll introduce these library abstractions, which are the subject of the C++ standard proposal P2300R10."***

2024. Eric Niebler published "What are Senders Good For, Anyway?" one month before the Tokyo meeting where P2300 received design approval. The reference implementation, stdexec, is maintained under NVIDIA's GitHub organization. The framing had changed:

> *"If your library exposes asynchrony, then returning a sender is a great choice: your users can await the sender in a coroutine if they like."*

Senders were now the foundation. Coroutines were one of several ways to consume them.

2025-2026. The coroutine integration shipped via P3552R3 ("Add a Coroutine Task Type"). P3796R1 ("Coroutine Task Issues") catalogued twenty-nine open concerns. D3980R0 ("Task's Allocator Use") reworked the allocator model six months after adoption. P4007R0 ("Senders and C++") documented three structural gaps.

Date	Published writing	Concurrent paper activity	Sender/coroutine relationship
2017	"Ranges, Coroutines, and React"	(ranges work, not P2300)	Vision is pure coroutines
2020	"Structured Concurrency"	P2300R0 ("std::execution") in development	Coroutines 90%, senders for 10% hot paths
2021	"Asynchronous Stacks and Scopes"	P2300R2 revisions	Coroutines overwhelming, senders upcoming
2024	"What are Senders Good For, Anyway?"	P2300R10; P3164R0 ("Improving Diagnostics")	Senders are the foundation

Date	Published writing	Concurrent paper activity	Sender/coroutine relationship
2025-2026	(no blog post)	P3826R3 ("Fix Sender Algorithm Customization"): "irreparably broken"	Design under active rework

Between 2017 and 2024, the emphasis changed. In 2017, the vision was coroutines and ranges. By 2020, coroutines were the primary model and senders were the optimization path for hot code. By 2024, as the problems being solved turned toward heterogeneous computing and GPU dispatch, senders naturally received more attention and the Sender Sub-Language became the foundation of `std::execution`.

Both models were discovered by the same engineer. Both were described, in his own published words, as valuable. His 2020 assessment - "90% of all async code in the future should be coroutines simply for maintainability" - remains a valid description of the coroutine model's strengths for I/O and general-purpose async programming. The Sender Sub-Language's strengths for compile-time work graphs and zero-allocation pipelines are equally real.

The question is not which discovery was right. Both were. Coroutines are already standardized. The question is whether the committee should give asynchronous I/O the same domain-specific accommodation it gave heterogeneous compute.

---

5. The Sub-Language in Practice

The following examples are drawn from the official stdexec repository and the sender-examples collection. Each illustrates a progressively more expressive pattern in the Sender Sub-Language, annotated with the functional programming concepts from Section 3. After each example, the same program is shown in C++.

5.1 A Simple Pipeline

```
auto work = just(42)                                    // pure/return
          | then([](int v) { return v + 1; })           // fmap/functor lift
          | then([](int v) { return v * 2; });          // functor composition
auto [result] = sync_wait(std::move(work)).value();     // evaluate the continuation
```

The same program, expressed in C++:

```
int v = 42;
v = v + 1;
v = v * 2;
```

This is the basic unit of composition in the Sender Sub-Language. A value is lifted into the sender context with `just`, and two transformations are applied through the pipe operator. The entire pipeline is lazy - nothing executes until `sync_wait` evaluates the reified continuation.

5.2 Branching

```
auto work = just(42)                                    // pure/return
          | then([](int v) {                            // fmap/functor lift
               return v > 0 ? v * 2 : -v;               // conditional value
            });
```

The same program, expressed in C++:

```
int v = 42;
if (v > 0)
    v = v * 2;
else
    v = -v;
```

Conditional logic that produces a value (not a new sender) can use `then`. When the branches must return different sender types, `let_value` with `variant_sender` is required.

5.3 Error Handling

```
auto work = just(std::move(socket))                     // pure/return
          | let_value([](tcp_socket& s) {               // monadic bind (>>=)
               return async_read(s, buf)                // Kleisli arrow
                    | then([](auto data) {              // fmap/functor lift
                         return parse(data);            // value continuation
                      });
            })
          | upon_error([](auto e) {                     // error continuation
               log(e);                                  // error channel handler
            })
          | upon_stopped([] {                           // stopped continuation
               log("cancelled");                        // cancellation channel handler
            });
```

The same program, expressed as a C++ coroutine:

```
try {
    auto data = co_await async_read(socket, buf);
    auto result = parse(data);
} catch (const std::exception& e) {
    log(e);
}
```

In the Sub-Language version above, the three-channel completion model dispatches each outcome to its own handler: `then` for the value channel, `upon_error` for the error channel, `upon_stopped` for the cancellation channel. Each channel has a distinct semantic role in the sum type over completion outcomes.

5.4 The Loop

The following example is drawn from loop.cpp in the sender-examples repository:

```
auto snder(int t) {
    int n = 0;                                          // initial state
    int acc = 0;                                        // accumulator
    stdexec::sender auto snd =
        stdexec::just(t)                                // pure/return
      | stdexec::let_value(                             // monadic bind (>>=)
            [n = n, acc = acc](int k) mutable {         // mutable closure state
                return stdexec::just()                  // unit/return
                     | stdexec::then([&n, &acc, k] {    // reference capture across
                           acc += n;                    //   continuation boundary
                           ++n;                         // mutate closure state
                           return n == k;               // termination predicate
                       })
                     | exec::repeat_effect_until()      // tail-recursive effect
                     | stdexec::then([&acc]() {         // accumulator extraction
                           return acc;
                       });
            });
    return snd;
}
```

The same program, expressed in C++:

```
int loop(int t) {
    int acc = 0;
    for (int n = 0; n < t; ++n)
        acc += n;
    return acc;
}
```

The Sender Sub-Language version expresses iteration as tail-recursive continuation composition. The `repeat_effect_until` algorithm repeatedly invokes the sender until the predicate returns true. State is maintained through mutable lambda captures with reference captures into the closure - a pattern that requires careful attention to object lifetime across continuation boundaries. The `repeat_effect_until` algorithm is provided by the stdexec reference implementation; it is not yet part of the C++26 working paper.

5.5 The Fold

The following example is drawn from fold.cpp in the sender-examples repository:

```
struct fold_left_fn {
    template<std::input_iterator I, std::sentinel_for<I> S,
             class T, class F>
    constexpr auto operator()(I first, S last, T init, F f) const
        -> any_sender_of<                                  // type-erased monadic return
            stdexec::set_value_t(
                std::decay_t<
                    std::invoke_result_t<F, T, std::iter_reference_t<I>>>),
            stdexec::set_stopped_t(),
            stdexec::set_error_t(std::exception_ptr)> {
        using U = std::decay_t<
            std::invoke_result_t<F, T, std::iter_reference_t<I>>>;

        if (first == last) {                               // base case
            return stdexec::just(U(std::move(init)));      // pure/return
        }

        auto nxt =
            stdexec::just(                                 // pure/return
                std::invoke(f, std::move(init), *first))
          | stdexec::let_value(                            // monadic bind (>>=)
                [this,
                 first = first,                            // iterator capture in closure
                 last = last,
                 f = f](U u) {
                    I i = first;
                    return (*this)(++i, last, u, f);       // recursive Kleisli composition
                });
        return std::move(nxt);                             // inductive case: bind + recurse
    }
};
```

The same program, expressed as a C++ coroutine:

```
task<U> fold_left(auto first, auto last, U init, auto f) {
    for (; first != last; ++first)
        init = f(std::move(init), *first);
    co_return init;
}
```

The sender version above demonstrates recursive Kleisli composition with type-erased monadic returns. The `fold_left_fn` callable object recursively composes senders: each step applies the folding function and then constructs a new sender that folds the remainder. The return type is `any_sender_of<>` because the recursive type would otherwise be infinite - type erasure breaks the recursion at the cost of a dynamic allocation per step.

The `any_sender_of` type is provided by the stdexec reference implementation, which is ahead of the standard in this area. The C++26 working paper does not yet include a type-erased sender. The stdexec team has implemented the facility that recursive sender composition requires; the standard will presumably follow in a future revision.

5.6 The Backtracker

The following example is drawn from backtrack.cpp in the sender-examples repository:

```
auto search_tree(auto                    test,           // predicate
                 tree::NodePtr<int>      tree,           // current node
                 stdexec::scheduler auto sch,            // execution context
                 any_node_sender&&       fail)           // the continuation (failure recovery)
    -> any_node_sender {                                 // type-erased monadic return
    if (tree == nullptr) {
        return std::move(fail);                          // invoke the continuation
    }
    if (test(tree)) {
        return stdexec::just(tree);                      // pure/return: lift into context
    }
    return stdexec::on(sch, stdexec::just())             // schedule on executor
      | stdexec::let_value(                              // monadic bind (>>=)
            [=, fail = std::move(fail)]() mutable {      // move-captured continuation
                return search_tree(                      // recursive Kleisli composition
                    test,
                    tree->left(),                        // traverse left subtree
                    sch,
                    stdexec::on(sch, stdexec::just())    // schedule on executor
                      | stdexec::let_value(              // nested monadic bind
                            [=, fail = std::move(fail)]() mutable {
                                return search_tree(      // recurse right subtree
                                    test,
                                    tree->right(),       // with failure continuation
                                    sch,
                                    std::move(fail));    // continuation-passing failure recovery
                            }));
            });
}
```

The same program, expressed in C++:

```
auto search_tree(auto test, tree::NodePtr<int> node) -> tree::NodePtr<int> {
    if (node == nullptr) return nullptr;
    if (test(node)) return node;
    if (auto found = search_tree(test, node->left())) return found;
    return search_tree(test, node->right());
}
```

The sender version above demonstrates recursive CPS with continuation-passing failure recovery and type-erased monadic return. The failure continuation is itself a sender pipeline passed as a parameter to a recursive function. Each recursive call nests another `let_value` lambda that captures and moves the failure sender. The reader must trace the `fail` parameter through three levels of `std::move` to understand which path executes. As with the fold example, the `any_node_sender` type alias uses stdexec's `any_sender_of<>`, a type-erased sender facility not yet included in the C++26 working paper.

5.7 The `retry` Algorithm

The following example is drawn from retry.hpp in the official NVIDIA stdexec repository. It implements a `retry` algorithm that re-executes a sender whenever it completes with an error. The complete implementation is reproduced here.

```
// Deferred construction helper - emplaces non-movable types
// into std::optional via conversion operator
template <class Fun>                                        // higher-order function wrapper
    requires std::is_nothrow_move_constructible_v<Fun>
struct _conv {
    Fun f_;                                                 // stored callable
    explicit _conv(Fun f) noexcept                          // explicit construction
        : f_(static_cast<Fun&&>(f)) {}
    operator std::invoke_result_t<Fun>() && {               // conversion operator
        return static_cast<Fun&&>(f_)();                    //   invokes stored callable
    }
};

// Forward declaration of operation state
template <class S, class R>
struct _op;

// Receiver adaptor - intercepts set_error to trigger retry
// All other completions pass through to the inner receiver
template <class S, class R>
struct _retry_receiver                                      // receiver adaptor pattern
    : stdexec::receiver_adaptor<                            // CRTP base for receiver
          _retry_receiver<S, R>> {
    _op<S, R>* o_;                                          // pointer to owning op state

    auto base() && noexcept -> R&& {                        // access inner receiver (rvalue)
        return static_cast<R&&>(o_->r_);
    }
    auto base() const& noexcept -> const R& {               // access inner receiver (const)
        return o_->r_;
    }
    explicit _retry_receiver(_op<S, R>* o) : o_(o) {}       // construct from op state

    template <class Error>
    void set_error(Error&&) && noexcept {                   // intercept error channel
        o_->_retry();                                       //   trigger retry instead
    }                                                       // error is discarded
};

// Operation state - holds sender, receiver, and nested op state
// The nested op is in std::optional so it can be destroyed
// and re-constructed on each retry
template <class S, class R>
struct _op {
    S s_;                                                   // the sender to retry
    R r_;                                                   // the downstream receiver
    std::optional<                                          // optional nested op state
```

```
        stdexec::connect_result_t<                          //   type of connect(S, retry_rcvr)
            S&, _retry_receiver<S, R>>> o_;                 //   re-created on each retry

    _op(S s, R r)                                           // construct from sender + receiver
        : s_(static_cast<S&&>(s))
        , r_(static_cast<R&&>(r))
        , o_{_connect()} {}                                 // initial connection

    _op(_op&&) = delete;                                    // immovable (stable address)

    auto _connect() noexcept {                              // connect sender to retry receiver
        return _conv{[this] {                               // deferred construction via _conv
            return stdexec::connect(                        //   connect sender
                s_,                                         //   to retry receiver
                _retry_receiver<S, R>{this});               //   that points back to this op
        }};
    }

    void _retry() noexcept {                                // retry: destroy and reconnect
        STDEXEC_TRY {
            o_.emplace(_connect());                         // re-emplace the operation state
            stdexec::start(*o_);                            // restart the operation
        }
        STDEXEC_CATCH_ALL {                                 // if reconnection itself throws
            stdexec::set_error(                             //   propagate to downstream
                static_cast<R&&>(r_),
                std::current_exception());
        }
    }

    void start() & noexcept {                               // initial start
        stdexec::start(*o_);                                // start the nested operation
    }
};

// The retry sender - wraps an inner sender and removes
// error completions from the completion signatures
template <class S>
struct _retry_sender {
    using sender_concept = stdexec::sender_t;               // declare as sender
    S s_;                                                   // the inner sender

    explicit _retry_sender(S s)                             // construct from inner sender
        : s_(static_cast<S&&>(s)) {}

    template <class... Args>                                // completion signature transform:
    using _error = stdexec::completion_signatures<>;        //   remove all error signatures
```

```
    template <class... Args>
    using _value =                                          // pass through value signatures
        stdexec::completion_signatures<
            stdexec::set_value_t(Args...)>;

    template <class Env>                                    // compute transformed signatures
    static consteval auto get_completion_signatures()
        -> stdexec::transform_completion_signatures<        // signature transformation
            stdexec::completion_signatures_of_t<S, Env>,    //   from inner sender's sigs
            stdexec::completion_signatures<                 //   add exception_ptr error
                stdexec::set_error_t(std::exception_ptr)>,
            _value,                                         //   pass through values
            _error> {                                       //   remove original errors
        return {};
    }

    template <stdexec::receiver R>                          // connect to a receiver
    auto connect(R r) && -> _op<S, R> {                     // produce operation state
        return {static_cast<S&&>(s_),                       // reify the continuation
                static_cast<R&&>(r)};
    }

    auto get_env() const noexcept                           // forward environment
        -> stdexec::env_of_t<S> {
        return stdexec::get_env(s_);
    }
};

template <class S>                                          // the user-facing function
auto retry(S s) -> stdexec::sender auto {
    return _retry_sender{static_cast<S&&>(s)};              // wrap in retry sender
}
```

A complete sender algorithm implementation demonstrating receiver adaptation, operation state lifecycle management, completion signature transformation, and the deferred construction pattern. The `_retry_receiver` intercepts `set_error` and calls `_retry()`, which destroys the nested operation state, reconnects the sender, and restarts it. The `_retry_sender` uses `transform_completion_signatures` to remove error signatures from the public interface, since the retry loop absorbs errors internally. The sender version accepts a sender directly and re-connects it by lvalue reference on each retry; the coroutine version takes a callable that produces a sender, since `co_await` consumes its operand.

The same algorithm, expressed as a C++ coroutine:

```
// Generic retry: takes a callable that produces a sender,
// co_awaits it in a loop, catches any error, and retries
// until success or stopped.
template<class F>
auto retry(F make_sender) -> task</*value type*/> {
    for (;;) {                                          // loop until success
        try {
            co_return co_await make_sender();           // attempt the operation
        } catch (...) {}                                // absorb the error, retry
    }
}
```

The question is whether Regular C++ developers should write asynchronous code this way, or whether simpler alternatives exist for the common case.

---

6. What Complexity Buys

The complexity documented in Section 5 is not accidental. It is the price of admission to a set of engineering properties that no other C++ async model provides.

6.1 The Trade-off

What you get	What it costs
Full type visibility - the compiler sees the entire work graph as a concrete type	Header-only implementations
Zero allocation in steady state - the operation state lives on the stack	Long compile times
Compile-time work graph construction - `connect` collapses the pipeline into a single type	The programming model of Section 5
Deterministic nanosecond-level execution - HPC Wire reports performance "on par with the CUDA implementation,"	

For GPU dispatch, high-frequency trading, embedded systems, and scientific computing, every party involved has opted in. The question is whether domains that do not need these properties - networking, file I/O, ordinary request handling - should be required to pay the same cost, or whether they deserve the same freedom to choose the model

that serves them.

For some, this is a good trade-off.

6.2 The Precedent

The committee designed `std::execution` to accommodate domain-specific needs. NVIDIA's nvexec demonstrates this accommodation in practice: a GPU-specific sender implementation that lives alongside stdexec in the same repository but in a separate namespace.

The `nvexec` implementation includes GPU-specific reimplementations of the standard sender algorithms - `bulk`, `then`, `when_all`, `continues_on`, `let_value`, `split`, and `reduce` - all in `.cuh` (CUDA header) files rather than standard C++ headers. The GPU scheduler in `stream_context.cuh` uses CUDA-specific types throughout:

```
// From nvexec/stream_context.cuh - CUDA execution space specifiers
STDEXEC_ATTRIBUTE(nodiscard, host, device)               // __host__ __device__
auto schedule() const noexcept {
    return sender{ctx_};
}
```

```
// From nvexec/stream/common.cuh - CUDA kernel launch syntax
continuation_kernel<<<1, 1, 0, get_stream()>>>(          // <<<grid, block, smem, stream>>>
    static_cast<R&&>(rcvr_), Tag(), static_cast<Args&&>(args)...);
```

```
// From nvexec/stream/common.cuh - CUDA memory management
status_ = STDEXEC_LOG_CUDA_API(
    cudaMallocAsync(&this->atom_next_, ptr_size, stream_));
```

These annotations and APIs are non-standard C++ language extensions. The CUDA C/C++ Language Extensions documentation enumerates them:

- **Execution space specifiers:** `__host__`, `__device__`, and `__global__` indicate whether a function executes on the host (CPU) or the device (GPU).
- **Memory space specifiers:** `__device__`, `__managed__`, `__constant__`, and `__shared__` indicate the storage location of a variable on the device.
- **Kernel launch syntax:** The `<<<grid_dim, block_dim, dynamic_smem_bytes, stream>>>` syntax between the function name and argument list is not valid C++.

GPU compute has requirements that standard C++ alone cannot meet. These extensions require a specialized compiler (`nvcc`), and every NVIDIA GPU user has opted in to that requirement.

Eric Niebler acknowledges this directly in P3826R3 (“Fix Sender Algorithm Customization”):

> *“Some execution contexts place extra-standard requirements on the code that executes on them. For example, NVIDIA GPUs require device-accelerated code to be annotated with its proprietary `__device__` annotation. Standard libraries are unlikely to ship implementations of `std::execution` with such annotations. The consequence is that, rather than shipping just a GPU scheduler with some algorithm customizations, a vendor like NVIDIA is already committed to shipping its own complete implementation of `std::execution` (in a different namespace, of course).”*

The committee accommodated a domain that needed its own model. GPU compute got a complete, domain-specific implementation of `std::execution` with non-standard extensions, a specialized compiler, and reimplementations of every standard algorithm.

This is a precedent.

6.3 The Principle

The CUDA C++ Language Support documentation (v13.1, December 2025) lists C++20 feature support for GPU device code. The following rows are representative:

C++20 Language Feature	nvcc Device Code
Concepts	Yes
Consistent comparison (`operator<=>`)	Yes
`consteval` functions	Yes
Parenthesized initialization	Yes
Coroutines	NOT SUPPORTED

Coroutine support for GPU device code may arrive in a future release. In the meantime, this is not a deficiency in either model. GPU compute has requirements that coroutines were not designed to meet, just as I/O has requirements that the Sender Sub-Language’s compile-time work graph was not designed to meet. Not every C++ feature must serve every domain.

Coexistence is principled.

6.4 Can Everyone Win?

The Sender Sub-Language serves specific domains exceptionally well:

- **High-frequency trading and quantitative finance**: where nanosecond-level determinism justifies any compile-time cost
- **GPU dispatch and heterogeneous computing**: where vendor-specific implementations (like nvexec) with non-standard compiler extensions are the norm
- **Embedded systems**: where heap allocation is prohibited and the zero-allocation guarantee is a hard requirement
- **Scientific computing**: where performance matches hand-written CUDA (Section 6.1)

Direct-style coroutines serve other domains equally well: networking, file I/O, request handling - the domains where partial success is normal, allocator propagation matters, and the programming model documented in Section 5 is a cost without a corresponding benefit.

C++ has always grown by adding models that serve specific domains. Templates serve generic programming. Coroutines serve async I/O. The Sender Sub-Language serves heterogeneous compute. The standard is stronger when each domain gets the model it needs, and neither is forced to use the other's tool. P4007R0 ("Senders and C++") examines the boundary where these two models meet.

Everyone can win.

---

7. Conclusion

C++26 has a new programming model. It has its own control flow, its own variable binding, its own error handling, and its own type system. It is grounded in four decades of programming language research, and it is already shipping in production at NVIDIA and Citadel Securities. That is not nothing. That is an achievement.

The Sender Sub-Language is here, and it is not going anywhere. The committee adopted it. The implementation exists. Real users depend on it. We should be proud of it - it solves problems that no other C++ async model can touch.

Yet does it have to solve every problem? The domains it serves - GPU dispatch, HFT, embedded, scientific computing - opted in to the trade-offs documented in this paper. The domains it does not serve - networking, file I/O, the everyday async code that most C++ developers write - deserve their own model, built for their own needs, with the same care and the same respect.

I think we can get there. The precedent exists. The principle is sound. And the committee has done harder things than giving two communities the tools they each need.

Further Reading

For readers who wish to explore the theoretical foundations of the Sender Sub-Language in greater depth, the following works provide essential background:

- *The Lambda Papers* (Steele and Sussman, 1975-1980) - the formalization of continuations and continuation-passing style
- *Notions of Computation and Monads* (Moggi, 1991) - the foundational paper on monads as a structuring principle for computation
- *Abstracting Control* (Danvy and Filinski, 1990) - delimited continuations and the shift/reset operators
- *Haskell 2010 Language Report* - the language whose type system and monadic abstractions most directly influenced the Sender Sub-Language's vocabulary
- *Category Theory for Programmers* (Milewski, 2019) - an accessible introduction to the categorical foundations: functors, monads, and Kleisli arrows
- *The Design Philosophy of the DARPA Internet Protocols* (Clark, 1988) - on how narrow, practice-first abstractions succeed where top-down designs do not

---

Acknowledgements

This document is written in Markdown and depends on the extensions in `pandoc` and `mermaid`, and we would like to thank the authors of those extensions and associated libraries.

The authors would also like to thank John Lakos, Joshua Berne, Pablo Halpern, and Dietmar Kühl for their valuable feedback in the development of this paper.

References

WG21 Papers

1. P2300R10. Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. "std::execution." 2024.
2. P2300R0. Michał Dominiak, Lewis Baker, Lee Howes, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. "std::execution." 2021.
3. P2300R2. Michał Dominiak, Lewis Baker, Lee Howes, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. "std::execution." 2021.
4. P3164R0. Eric Niebler. "Improving Diagnostics for Sender Expressions." 2024.

5. P3552R3. Dietmar Kühl, Maikel Nadolski. "Add a Coroutine Task Type." 2025.
6. P3796R1. Dietmar Kühl. "Coroutine Task Issues." 2025.
7. P3826R3. Eric Niebler. "Fix Sender Algorithm Customization." 2026.
8. D3980R0. Dietmar Kühl. "Task's Allocator Use." 2026.
9. P4007R0. Vinnie Falco, Mungo Gill. "Senders and C++." 2026.

Blog Posts

10. Eric Niebler. "Ranges, Coroutines, and React: Early Musings on the Future of Async in C++". 2017.
11. Eric Niebler. "Structured Concurrency". 2020.
12. Eric Niebler. "Asynchronous Stacks and Scopes". 2021.
13. Eric Niebler. "What are Senders Good For, Anyway?". 2024.
14. Herb Sutter. "Living in the Future: Using C++26 at Work". 2025.

Functional Programming Theory

15. Guy Steele and Gerald Sussman. ***The Lambda Papers***. MIT AI Memo series, 1975-1980.
16. Eugenio Moggi. "Notions of Computation and Monads". ***Information and Computation***, 1991.
17. Olivier Danvy and Andrzej Filinski. "Abstracting Control". ***LFP***, 1990.
18. Timothy Griffin. "A Formulae-as-Types Notion of Control". ***POPL***, 1990.
19. Simon Marlow (ed.). ***Haskell 2010 Language Report***. 2010.
20. Bartosz Milewski. ***Category Theory for Programmers***. 2019.

Books

21. Bjarne Stroustrup. ***The Design and Evolution of C++***. Addison-Wesley, 1994.
22. David Clark. "The Design Philosophy of the DARPA Internet Protocols". ***SIGCOMM***, 1988.

Implementations and Examples

23. stdexec. NVIDIA's reference implementation of `std::execution`.
24. `any_sender_of.hpp`. Type-erased sender facility in stdexec (not part of C++26).
25. `variant_sender.hpp`. Variant sender facility in stdexec (not part of C++26).
26. `repeat_effect_until.hpp`. Iteration algorithm in stdexec (not part of C++26).
27. `retry.hpp`. Retry algorithm example in stdexec.
28. sender-examples. Example code for C++Now talk (Steve Downey).
29. `loop.cpp`. Iteration example in sender-examples.
30. `fold.cpp`. Fold example in sender-examples.

31. `backtrack.cpp`. Backtracking search example in sender-examples.

Background

32. SML/NJ. Standard ML of New Jersey (uses CPS as internal representation).
33. GHC. Glasgow Haskell Compiler.
34. Chicken Scheme. Scheme implementation using CPS compilation.
35. Haskell `Just`. Data.Maybe module.
36. Haskell `Monad`. Control.Monad module.
37. Kleisli category. Wikipedia.
38. Delimited continuation. Wikipedia.
39. Curry-Howard correspondence. Wikipedia.
40. Algebraic effect system. Wikipedia.

NVIDIA CUDA and nvexec

41. nvexec. GPU-specific sender implementation in the stdexec repository.
42. `stream_context.cuh`. GPU stream scheduler and context.
43. `common.cuh`. Shared GPU sender infrastructure.
44. `bulk.cuh`. GPU-specific `bulk` algorithm.
45. `then.cuh`. GPU-specific `then` algorithm.
46. `when_all.cuh`. GPU-specific `when_all` algorithm.
47. `continues_on.cuh`. GPU-specific `continues_on` algorithm.
48. `let_xxx.cuh`. GPU-specific `let_value` / `let_error` / `let_stopped` algorithms.
49. `split.cuh`. GPU-specific `split` algorithm.
50. `reduce.cuh`. GPU-specific `reduce` algorithm.
51. CUDA C/C++ Language Extensions. NVIDIA CUDA Programming Guide.
52. `nvcc`. NVIDIA CUDA Compiler Driver.
53. "New C++ Sender Library Enables Portable Asynchrony". HPC Wire, 2022.
54. `connect`. C++26 draft standard, [exec.connect].
55. CUDA C++ Language Support. NVIDIA CUDA Programming Guide v13.1, Section 5.3 (December 2025).