WG21 PROPOSAL

The Sender Sub-Language

Document Number:	D4014R0
Date:	2026-02-15
Audience:	All of WG21

Reply-to:	Vinnie Falco vinnie.falco@gmail.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, its relationship to C++ language features, and the programming model it provides.

---

1. Introduction

`std::execution` (P2300R10) was formally adopted into the C++26 working paper at St. Louis in July 2024. It is coming. Every C++ developer who writes asynchronous code will 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++*** 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 ("Are Senders Replacing Coroutines?") 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 Mapping

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:

C++ (direct style)	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

C++ (direct style)	Sender Sub-Language
Local variables	Lambda captures (with move semantics across boundaries)
Function call + return	`connect` + `start` + `set_value`
Structured bindings	(not needed)
Range-for	(not needed)
`if` with initializer	(not needed)

Control Flow

In C++, a `for` loop iterates by mutating a counter and testing a condition. In the Sender Sub-Language, iteration takes two forms. Dedicated loop combinators such as `repeat_effect_until` handle simple cases without type erasure. Recursive iteration, where a sender constructs another sender of the same shape on completion, requires type erasure (`any_sender_of<>`) to break the otherwise infinite recursive type. Both facilities 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.

Branching in C++ uses `if` / `else` or `switch` . In the Sub-Language, conditional logic is expressed through `let_value` returning a sender depending on a runtime condition. Both branches must return the same concrete sender type; when they do not, `variant_sender` or type erasure is required to unify the return types.

Error Handling

C++ provides `try` / `catch` for exception handling and return values for error codes. The Sender Sub-Language provides three completion channels (`set_value` , `set_error` , and `set_stopped`) forming a fixed sum type over completion outcomes. Error handling is expressed through `upon_error` and `let_error` , which intercept the error channel and transform or replace it. The `upon_stopped` adaptor handles the cancellation channel.

Variable Binding

Local variables in C++ have lexical scope tied to the enclosing block. In the Sender Sub-Language, state is carried through lambda captures across continuation boundaries. A value produced by one stage of the pipeline reaches the next stage through a captured reference or moved value inside a `let_value` lambda. The lifetime of that state is governed by move semantics and the operation state's lifetime, not by lexical scope.

What Has No Equivalent

Three C++ language features have no counterpart in the Sender Sub-Language: structured bindings, range-for, and `if` with initializer. These are direct-style features introduced in C++17 and C++20 that operate on return values. The Sender Sub-Language does not produce intermediate return values. Values flow forward into continuations as arguments, not backward to callers as returns, so there is no aggregate to destructure at intermediate stages.

---

3. Theoretical Foundations

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.

Continuation-Passing Style

CPS is a style of programming where control is passed explicitly via continuation functions rather than implicitly via the call stack. Every function takes an extra argument (the continuation) representing "what to do next." The function never returns; it invokes the continuation instead. CPS has its origins in denotational semantics and was formalized in the Lambda Papers (1975-1980), where continuations were shown to be the computational equivalent of GOTO with lexical scope and first-class status.

CPS serves as the standard intermediate representation in optimizing compilers (SML/NJ, GHC, Chicken Scheme) because it makes control flow, variable binding, and resource lifetime explicit in the term structure. These are exactly the properties that enable zero-allocation pipelines and compile-time work graph construction in the Sender Sub-Language.

Monadic Composition

The Sender Sub-Language's naming choices pay homage to a rich functional programming heritage. `just` echoes Haskell's `Just` constructor, lifting a pure value into a monadic context (analogous to `return` / `pure` in the `Monad` typeclass). `let_value` mirrors Haskell's monadic bind (`>>=`), the essential operation of Kleisli composition: extracting the value from a computation and threading it into the next. The `let_` prefix recalls ML's `let ... in ...` bindings. These are not arbitrary names. They signal that sender pipelines are monadic composition expressed as C++ value types, a deliberate and well-informed borrowing from decades of functional programming research.

Algebraic Effects and Structured Concurrency

CPS enables delimited continuations and algebraic effect handlers, the theoretical foundation for structured concurrency itself (***Abstracting Control***, 1990). The Sender Sub-Language's three-channel completion model (`set_value` , `set_error` , `set_stopped`) is a fixed algebraic effect system: the completion channels are the effects,

and the receiver is the handler. The Curry-Howard-Lambek correspondence, extended to classical logic (***A Formulae-as-Types Notion of Control***, 1990), provides a proof-theoretic reading: a continuation is a refutation, a proof that the program's current value suffices to complete the computation.

Reification

In the Sender Sub-Language, `connect` reifies the continuation into an operation state, a concrete object that holds all data needed for the async operation at a stable address. The operation state is the CPS transform made manifest. The entire work graph, from the initial `just` through every `then` and `let_value`, collapses into a single deeply-nested template type. Nothing executes until `start` is called. The design makes the entire continuation structure available to the compiler at compile time, enabling the zero-allocation pipelines that make the Sender Sub-Language valuable for GPU dispatch and latency-critical compute.

The P2300 authors built a framework grounded in four decades of programming language research. The question is whether every C++ developer who writes asynchronous code should be expected to navigate these theoretical foundations, or whether simpler alternatives exist for the common case.

---

4. The Evolution

Eric Niebler's published writing provides the most complete public record of the design thinking behind `std::execution`. His blog posts document the evolution with unusual candor and intellectual honesty. The following timeline is drawn from that published record.

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", the same year P2300R0 was submitted. 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 P2300."***

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 shifted:

> *"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 catalogued sixteen open issues. D3980R0 reworked the allocator model six months after adoption. P4007R0 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

Date	Published writing	Concurrent paper activity	Sender/coroutine relationship
2020	"Structured Concurrency"	P2300R0 submitted	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	Senders are the foundation
2025-2026	(no blog post)	P3826R3: "irreparably broken", 50 mods	Design under active rework

Between 2017 and 2024, the relationship between senders and coroutines underwent a complete inversion. In 2017, the vision was pure coroutines and ranges. By 2020, coroutines were the primary model and senders were the optimization path for hot code. By 2024, senders were the primary abstraction and coroutines were one of several ways to consume them.

The natural reading of this trajectory is that Eric Niebler discovered, through implementation experience, that senders are a more powerful foundation than coroutines - that the Sender Sub-Language is, for the purposes of `std::execution`, the preferred programming model, with coroutines serving as syntactic sugar over it.

That is a legitimate design conclusion. The committee should evaluate it on its merits. But it is a conclusion the committee should reach explicitly, not inherit implicitly from a design trajectory. If the Sender Sub-Language is the real programming model and coroutines are a convenience layer, that choice has consequences for every C++ developer who writes async code - and the committee should decide explicitly whether the Language is the future of async C++.

---

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 (auto& 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, for readers who prefer C++ to the Language:

```
// 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
    }
}
```

---

6. What Complexity Buys

The complexity documented in Section 5 is not accidental. It buys real engineering properties that no other C++ async model provides.

6.1 Full Type Visibility

The compiler sees the work graph as a concrete type, except at type-erasure boundaries introduced by `any_sender_of<>` or similar facilities. Every sender, every continuation, every completion signature is a template parameter. The optimizer can inline through the entire pipeline without virtual dispatch, indirect calls, or opaque boundaries. The type of the operation state produced by `connect` encodes the complete structure of the computation.

6.2 Zero Allocation in Steady State

The operation state produced by `connect` is a single stack-allocated object containing the entire pipeline. No `new`, no `malloc`, no coroutine frame allocation. The Sub-Language can execute an arbitrarily deep sender pipeline without touching the heap. P2300R10 section 1.2 lists this as a design priority: the framework must ***"Care about all reasonable use cases, domains and platforms,"*** including embedded systems where dynamic allocation is prohibited.

6.3 Compile-Time Work Graph Construction

Because sender pipelines are lazy and compose as value types, the call to `connect(sndr, rcvr)` collapses the entire pipeline into a single deeply-nested template instantiation. The work graph is a compile-time data structure. The compiler can reason about it as a whole, optimizing across operation boundaries that would be opaque in a

callback or coroutine model. Nothing like this is possible with coroutines, whose frames are allocated at runtime and whose contents are opaque to the optimizer.

6.4 Deterministic, Nanosecond-Level Execution

With the graph constructed at compile time and no allocations at runtime, sender pipelines can execute with deterministic latency. For high-frequency trading and quantitative finance, where nanoseconds matter and allocation jitter is unacceptable, the Sub-Language provides guarantees that no other async model in C++ can offer. The HPC Wire article (2022) reports that parallel performance of the stdexec reference implementation is ***"on par with CUDA code."***

6.5 Header-Only by Necessity

Because the types encode the entire work graph, implementations must appear in headers. Separate compilation is structurally impossible for sender pipelines; the deeply-nested template types cannot cross translation unit boundaries without explicit type erasure. The cost is longer compile times. The payoff is maximal optimization opportunity. For domains that prize runtime performance above all else, that exchange is correct.

6.6 Vendor Extensibility

The Sender Sub-Language's design is extensible enough that hardware vendors can provide their own specialized implementations tailored to their platforms. NVIDIA demonstrates this through nvexec, 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++.

None of these can be compiled by GCC, MSVC, or Clang without CUDA support. Every NVIDIA GPU user already depends on a non-standard compiler (`nvcc`).

Eric Niebler acknowledges this directly in P3826R3 (“Fix Sender Algorithm Customization”), section 5.3:

> *“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 Sender Sub-Language is extensible enough to accommodate an entirely separate implementation for specialized hardware.

6.7 C++20 on the GPU

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

C++20 Language Feature	nvcc Device Code
Parenthesized initialization	Yes
Coroutines	NOT SUPPORTED

Coroutine support for GPU device code may arrive in a future release.

6.8 Who Benefits

These properties - full type visibility, zero allocation, compile-time graph construction, deterministic latency, vendor extensibility - serve 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 6.4)

The committee may wish to consider whether the Sender Sub-Language's complexity - documented in Sections 2 through 5 of this paper - is a price that every C++ developer should pay, or whether the benefits accrue primarily to these specialized domains. P4007R0 ("Are Senders Replacing Coroutines?") examines this question from the coroutine perspective.

---

7. Conclusion

C++26 introduces the Sender Sub-Language, a programming model for asynchronous computation rooted in continuation-passing style, monadic composition, and algebraic effect theory. The Sub-Language provides its own control flow (`let_value` for sequencing, recursive continuation for iteration), its own variable binding (lambda captures across continuation boundaries), its own error handling (three-channel completion dispatch), and its own type system (completion signatures). Appendix A maps the Sub-Language's vocabulary to its theoretical foundations.

We hope this guide helps committee members and users understand the Sender Sub-Language's relationship to the C++ they already know.

Appendix A: Glossary of the Sender Sub-Language

The Sender Sub-Language draws on a rich tradition of functional programming and type theory. The following table maps the Sub-Language's vocabulary to its theoretical foundations, provided as a reference for readers encountering these concepts for the first time.

Sender Sub-Language	Functional Programming Theory
`just(x)`	`return` / `pure` - lifting a plain value into a monadic context without performing any computation
`let_value(f)`	Monadic bind (`>>=`) - extracting the value from a monadic computation and passing it to a Kleisli arrow
`then(f)`	`fmap` / functor map - applying a function to the value inside a context without changing the context structure
`when_all(a, b)`	Applicative `<*>` - parallel composition of independent monadic computations
`upon_error(f)`	`fmap` for the error channel - transforms the error value without leaving the error path (`then` equivalent for errors)
`let_error(f)`	Monadic bind for the error channel - the callable returns a new sender that may switch to the value path (`let_value` equivalent for errors)
`upon_stopped(f)`	`fmap` for the stopped channel - transforms the stopped signal (`then` equivalent for cancellation)
`let_stopped(f)`	Monadic bind for the stopped channel - the callable returns a new sender that may switch to the value path (`let_value` equivalent for cancellation)
`set_value` / `set_error` / `set_stopped`	Algebraic effect channels - a fixed sum type over three completion outcomes
`connect(sndr, rcvr)`	Continuation reification - instantiating the CPS transform into a concrete operation
`start(op)`	Evaluating the reified continuation - initiating the CPS computation
sender	A lazy monadic computation - a description of work that does not execute until bound to a continuation
receiver	The continuation - a three-channel callback representing "the rest of the computation"
operation state	The reified continuation object - holds all data at a stable address for the duration of the computation
completion signatures	Type-level description of the sum type - the possible ways a monadic computation can complete

Sender Sub-Language	Functional Programming Theory
domain	Dispatch tag for algorithm selection - determines which implementation of a sender algorithm is used
`transform_sender`	Sender transformation prior to continuation binding - rewriting the monadic computation before evaluation
early customization	Compile-time dispatch before the continuation is known - selecting algorithm implementation at construction time
late customization	Dispatch at continuation-binding time - selecting implementation when the receiver's environment is available
forwarding query	Environment propagation through the continuation chain - passing contextual information from receiver to sender
completion scheduler	The execution context where the continuation will be invoked
`schedule(sch)`	Scheduling a unit continuation onto an execution context
`any_sender_of<>`	Type erasure of the monadic computation - hiding concrete sender types behind a uniform interface
non-dependent sender	A sender whose completion signatures are knowable without an environment - statically determined monadic type
receiver adaptor	Adapter pattern for continuations - wrapping a receiver to intercept or modify completion signals

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

References

WG21 Papers

1. P2300R10. Michal Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. "std::execution." 2024.
2. P2300R0. Michal Dominiak, Lewis Baker, Lee Howes, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach. "std::execution." 2021.
3. P2300R2. "std::execution." 2021.
4. P3164R0. Eric Niebler. "Improving Diagnostics for Sender Expressions." 2024.
5. P3552R3. Dietmar Kuhl, Maikel Nadolski. "Add a Coroutine Task Type." 2025.
6. P3796R1. Dietmar Kuhl. "Coroutine Task Issues." 2025.
7. P3826R3. Eric Niebler. "Fix Sender Algorithm Customization." 2026.
8. D3980R0. Dietmar Kuhl. "Task's Allocator Use." 2026.
9. P4007R0. Vinnie Falco, Mungo Gill. "Are Senders Replacing Coroutines?" 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.

Functional Programming Theory

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

Books

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

Implementations and Examples

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

Background

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

NVIDIA CUDA and nvexec

39. nvexec. GPU-specific sender implementation in the stdexec repository.
40. `stream_context.cuh`. GPU stream scheduler and context.
41. `common.cuh`. Shared GPU sender infrastructure.
42. `bulk.cuh`. GPU-specific `bulk` algorithm.
43. `then.cuh`. GPU-specific `then` algorithm.
44. `when_all.cuh`. GPU-specific `when_all` algorithm.
45. `continues_on.cuh`. GPU-specific `continues_on` algorithm.
46. `let_xxx.cuh`. GPU-specific `let_value` / `let_error` / `let_stopped` algorithms.
47. `split.cuh`. GPU-specific `split` algorithm.
48. `reduce.cuh`. GPU-specific `reduce` algorithm.
49. CUDA C/C++ Language Extensions. NVIDIA CUDA Programming Guide.
50. `nvcc`. NVIDIA CUDA Compiler Driver.
51. "New C++ Sender Library Enables Portable Asynchrony". HPC Wire, 2022.

52. `connect`. C++26 draft standard, [exec.connect].

53. CUDA C++ Language Support. NVIDIA CUDA Programming Guide v13.1, Section 5.3 (December 2025).