The Sender Sub-Language For Beginners

Document Number:	P4014R2
Date:	2026-05-01
Intent:	Inform
Audience:	LEWG

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

Table of Contents

Abstract

Every sender algorithm in C++26 - all thirty - explained, demonstrated, and mapped to its plain-C++ equivalent.

This paper is a progressive tutorial. It introduces every sender algorithm in C++26 [1], one at a time, from the simplest to the most complex. Each algorithm is defined, demonstrated in a working example, and explained. After the explanation, the equivalent C++ program appears without commentary - the equivalents compute the same results but do not preserve concurrency or execution context semantics. The theoretical foundations are presented first - the lambda calculus, continuation-passing style, and monadic composition that give the Sub-Language its structure - followed by the algorithms themselves, grouped by function and ordered by escalating complexity. The final sections cover the `task` coroutine type (P3552R3, "Add a Coroutine Task Type") [2] and the composition patterns that emerge when senders and coroutines interleave.

Revision History

R2: May 2026 (pre-Brno mailing)

- Formatting corrections.

R1: April 2026 (post-Croydon mailing)

- Complete rewrite as a progressive tutorial.
- Covers all thirty sender algorithms in the C++26 working draft.
- Adds coverage of `task` and sender-coroutine composition.
- Adds real-world examples (backtracker, retry) from stdexec and sender-examples.
- Updated for Croydon: `affine_on` renamed to `affine` (P4151R1) and made unary (P3941R4), `task_scheduler` uses `get_start_scheduler`, allocator description updated for P3980R1 two-tier model.

R0: March 2026 (pre-Croydon mailing)

- Initial version.

---

1. Disclosure

The author provides information and serves at the pleasure of the committee.

Falco developed and maintains Capy [3] and Corosio [4] and believes coroutine-native I/O is a practical foundation for networking in C++.

Coroutine-native I/O and `std::execution` are complementary. Each serves the domain where its design choices pay off.

This paper asks for nothing.

---

2. Theoretical Foundations

This section is optional. Readers who prefer to learn by example may skip to Section 3 and return here when the theoretical connections become interesting.

The Sender Sub-Language is not merely a fluent API. It is continuation-passing style expressed as composable value types, drawing on techniques refined across four decades of programming language research [5] [6]. Understanding where the pieces come from makes the tutorial that follows easier to absorb.

2.1 Continuations and CPS

The Lambda Papers [5] (Steele and Sussman, 1975-1980) formalized continuation-passing style: every function receives an explicit continuation representing "what happens next." Instead of returning a value to its caller, a function passes its result forward into the continuation. The sender protocol works the same way. `connect(sndr, rcvr)` reifies the continuation - it binds a sender to its receiver, producing an operation state that holds the entire work graph as a concrete type. `start(op)` evaluates it.

Gordon Plotkin's 1975 paper on call-by-value lambda calculus [7] established that CPS makes evaluation order explicit in the term structure. This is why optimizing compilers - SML/NJ [8], Chicken Scheme [9] - use CPS as their intermediate representation, why GHC [10] uses a closely related continuation-based model in its STG machine, and why the Sender Sub-Language can build zero-allocation pipelines and compile-time work graphs.

2.2 Monads

Eugenio Moggi's 1991 paper "Notions of Computation and Monads" [6] showed that monads structure computation with effects (see also Milewski [11] for an accessible treatment). Two operations are sufficient: `return` (lift a value into the computational context) and `bind` (sequence one computation into the next). In the Sender Sub-Language, `just(x)` is monadic return and `let_value(f)` is monadic bind. `then(f)` is the functor lift - `fmap` - a specialization where the function returns a plain value rather than a new sender.

2.3 Delimited Continuations and Algebraic Effects

Olivier Danvy and Andrzej Filinski's 1990 paper "Abstracting Control" [12] introduced delimited continuations and the shift/reset operators, enabling multiple effect channels within a single computation. The sender three-channel model - `set_value`, `set_error`, `set_stopped` - applies the algebraic-effects pattern with a fixed channel set. Each channel carries a distinct semantic role, and the composition algebra dispatches on which channel a result arrives on.

Timothy Griffin's 1990 paper "A Formulae-as-Types Notion of Control" [13] connected control operators to type systems. The sender completion signatures - the type-level declaration of which channels a sender may complete on and with what argument types - are this connection made concrete in C++.

2.4 The Mapping

Sender concept	Theoretical origin
`just(x)`	Monadic return / `pure`
`let_value(f)`	Monadic bind (`>>=`)
`then(f)`	Functor `fmap`
`set_value` / `set_error` / `set_stopped`	Algebraic effect channels
`connect(sndr, rcvr)`	CPS reification
`start(op)`	CPS evaluation
Completion signatures	Type-level signature sets
Scheduler algorithms	Indexed monad / delimited continuations
`read_env` / `write_env`	Reader monad (`ask` / `local`)
`when_all`	Parallel composition (CCS)
`split`	Contraction (linear logic)
Async scopes	Region-based lifetimes
`bulk` family	Data-parallel map

The names are not arbitrary. The P2300R0 authors chose them with care, and the choices reflect genuine scholarship. `just` echoes Haskell's [14] `Just`. `let_value` mirrors monadic bind. The three completion channels form a closed effect algebra. The framework is grounded in real theory, and the theory is worth knowing before the tutorial begins.

---

3. Value Lifting

The simplest sender algorithms place a value into the pipeline and let it carry forward. Nothing is computed, nothing is fetched. The value is already known, and these algorithms welcome it into the sender world.

Theory: In Moggi's framework [6], this is the unit of the monad - the natural transformation eta : Id -> T that lifts a plain value into a computational context without performing any effect. The sender completion type is a three-way coproduct - Value + Error + Stopped - and each `just` variant is the monadic unit composed with a different coproduct injection. `just` injects into the value summand, `just_error` into the error summand, `just_stopped` into the stopped summand. The construction is the same; only the target channel differs.

3.1 `just`

`just(vs...)` creates a sender that completes immediately with the given values on the value channel, where `vs...` are the values to carry forward.

```
auto sndr = just(42);
```

Put values into the pipeline. Nothing happens to them - they are simply there for the next step. The reader will appreciate the elegance: one value, placed into the sender context, ready to go.

Theory: This is monadic return - the `pure` operation that lifts a value into the sender context. The sender completes synchronously, inline, with no allocation and no scheduling. It is the entry point into every pipeline.

The equivalent program:

```
int v = 42;
```

3.2 `just_error`

`just_error(e)` creates a sender that completes immediately with the error `e` on the error channel.

```
auto sndr = just_error(
    make_error_code(errc::invalid_argument));
```

Signal that something went wrong. The error travels down the pipeline until something handles it. Equally natural - when an error needs to enter the pipeline, `just_error` provides it cleanly and without ceremony.

`just_error` lifts an error into the sender context the way `just` lifts a value. The error is delivered to the first error handler downstream - an `upon_error` or `let_error` in the pipeline.

The equivalent program:

```
throw system_error(
    make_error_code(errc::invalid_argument));
```

3.3 `just_stopped`

`just_stopped()` creates a sender that completes immediately on the stopped channel. It takes no arguments. No value is produced and no error is reported. The operation was cancelled.

```
auto sndr = just_stopped();
```

Signal that the operation was cancelled. This is how the programmer says "never mind" in a sender pipeline. The third channel completes the picture - a dedicated path for a dedicated intent.

The stopped channel is the third path in the sender completion algebra - distinct from both success and failure. It represents cancellation as a first-class concept. The reader will find this a natural extension of the two-channel error model - a dedicated path for a dedicated intent. Regular C++ has no dedicated cancellation channel; the three-channel model is one of the properties the Sub-Language adds.

3.4 `sync_wait`

`sync_wait(sndr)` blocks the current thread until the sender completes, then returns the result as an `optional<tuple<...>>`. The argument `sndr` is the sender to execute. Strictly, `sync_wait` is a consumer, not a lifter - it appears here because every example from this point forward needs it.

```
auto [result] =
    sync_wait(just(42)).value();
// result == 42
```

The reader has built a sender. Now they want the result. `sync_wait` is that moment - the point where the sender world opens and delivers its contents to regular C++ code. It is the only sender consumer in the standard, and nothing in the pipeline executes until `sync_wait` drives it to completion.

`sync_wait` is the bridge between the Sender Sub-Language and regular C++. Everything upstream is lazy - a description of work, not the execution of it. `sync_wait` makes it real.

Theory: In Moggi's framework [6], `sync_wait` is the counit - the elimination form that collapses the monadic layer back to a plain value. The Lambda Papers [5] showed that CPS transforms every computation into a suspended form awaiting its continuation; `sync_wait` provides the final continuation and forces evaluation. The analogous boundaries in Haskell are `runST` and `unsafePerformIO` - the single point where the computational context yields a result to the world outside. There is only one such point in the sender algebra, and this is it.

The equivalent program:

```
int result = 42;
```

---

4. Transformation

Take what came before, apply a function, and carry the result forward. The transformation algorithms are the most natural thing a pipeline can do - they turn one value into another.

Theory: These are the functorial action on morphisms [6] - Moggi's `fmap`, the lifting T(f) : T(A) -> T(B) for a function f : A -> B. The sender monad is a functor on the category of C++ types; the transformation algorithms are its action on arrows. Each variant applies the functor to a different component of the coproduct completion: `then` maps the value injection, `upon_error` maps the error injection, `upon_stopped` maps the stopped injection. The remaining components pass through as the identity natural transformation. Three algorithms, one construction - the same endofunctor action on different summands of the coproduct.

4.1 then

`then(f)` applies `f` to the values produced by the predecessor sender, where `f` is a callable that receives those values and returns a new value. The return value of `f` becomes the new value completion.

```
auto sndr = just(3, 4)
          | then([](int a, int b) {
                return a * a + b * b;
            });
// completes with set_value(25)
```

The most natural operation in the Sub-Language: take what came before, apply a function, and carry the result forward. The reader will find this immediately familiar - it is function application, expressed as a pipeline stage.

Theory: `then` is functor `fmap`. It transforms values without changing the structure of the pipeline - the sender still completes on the value channel, with a different value. Errors and stopped signals pass through untouched. If `f` throws, the exception is caught and delivered as `set_error(current_exception())`.

The equivalent program:

```
int result = 3 * 3 + 4 * 4;
```

4.2 upon_error

`upon_error(f)` applies `f` to the error produced by the predecessor sender, where `f` is a callable that receives the error value and returns a recovery value. The return value of `f` becomes a value completion.

```
auto sndr = just_error(
        make_error_code(errc::connection_reset))
      | upon_error([](error_code ec) {
            return -1;
        });
// completes with set_value(-1)
```

When something goes wrong, `upon_error` provides the recovery. The function receives the error and returns a fallback value. The error is handled; the pipeline continues as if nothing happened.

`upon_error` is the error-channel counterpart of `then`. Where `then` transforms values, `upon_error` transforms errors into values - the result crosses channels, converting an error completion into a value completion. Values and stopped signals pass through untouched.

The equivalent program:

```
int result;
try {
    throw system_error(
        make_error_code(
            errc::connection_reset));
} catch (system_error const&) {
    result = -1;
}
```

4.3 upon_stopped

`upon_stopped(f)` applies `f` when the predecessor sender is cancelled, where `f` takes no arguments and returns a recovery value. The return value of `f` becomes a value completion.

```
auto sndr = just_stopped()
          | upon_stopped([] {
                return 0;
            });
// completes with set_value(0)
```

The cancellation counterpart. If the operation was stopped, provide a fallback value instead. The stopped signal is absorbed; the pipeline moves on.

`upon_stopped` converts a cancellation into a value. It is used when a pipeline needs a fallback result in the event of cancellation rather than propagating the stopped signal. Regular C++ has no cancellation equivalent.

---

5. Monadic Composition

Sometimes the next step in a computation is itself a whole new computation. The `let_*` family handles that moment - when the function returns not just a value, but an entire pipeline.

Theory: A Kleisli arrow [6] [15] is a morphism A -> T(B) - a function from a plain value to a computation. Monadic bind takes a Kleisli arrow and extends it to T(A) -> T(B), composing the outer computation with the inner one through the monad's multiplication mu : T(T(A)) -> T(A). `let_value(f)` is precisely this extension. The Kleisli category of the sender monad is the category whose morphisms are sender-returning functions and whose composition is `let_value` chaining - the pipeline the programmer writes. `let_error` and `let_stopped` are the same Kleisli extension applied to the error and stopped injections of the coproduct. Three algorithms, one categorical construction.

5.1 `let_value`

`let_value(f)` applies `f` to the values produced by the predecessor, where `f` is a callable that receives those values and returns a new sender. That sender's completion becomes the completion of the composed pipeline.

```
auto sndr = just(std::string("hello"))
          | let_value([](std::string s) {
                return just(s + " world")
                     | then([](std::string r) {
                           return r.size();
                       });
            });
// completes with set_value(11)
```

Here the Sub-Language reveals its expressive depth. Where `then` applies a function that returns a value, `let_value` applies a function that returns an entirely new sender - a new pipeline within the pipeline. The reader will find this a natural extension of the composition model.

Theory: `let_value` is monadic bind. The function receives the predecessor's values and returns a sender which may itself be a multi-stage pipeline. The result is a sender whose completion type is determined by the inner sender, not by the function's return type directly. This is what makes `let_value` more powerful than `then`: the next stage of computation is itself a sender. The pattern is straightforward once the distinction between returning a value and returning a sender is clear.

The equivalent program:

```
std::string s = "hello";
s += " world";
size_t result = s.size();
```

5.2 `let_error`

`let_error(f)` applies `f` to the error produced by the predecessor, where `f` is a callable that receives the error and returns a new sender. The error is recovered by routing into a new computation.

```
auto sndr = just_error(
        make_error_code(errc::timed_out))
      | let_error([](error_code ec) {
            return just(ec.message());
        });
// completes with set_value("timed out")
```

The same principle, applied to errors. The recovery is itself an asynchronous operation - a pipeline that does real work to recover from the failure.

`let_error` is the error-channel counterpart of `let_value`. Where `upon_error` transforms an error into a value, `let_error` transforms an error into a sender - enabling recovery paths that are themselves multi-stage pipelines.

The equivalent program:

```
std::string result;
try {
    throw system_error(
        make_error_code(errc::timed_out));
} catch (system_error const& e) {
    result = e.code().message();
}
```

5.3 `let_stopped`

`let_stopped(f)` applies `f` when the predecessor is cancelled, where `f` takes no arguments and returns a new sender that replaces the cancellation.

```
auto sndr = just_stopped()
          | let_stopped([] {
                return just(
                    std::string("recovered"));
            });
// completes with set_value("recovered")
```

And for cancellation. The replacement for a stopped operation is itself a sender - a new computation that takes over where the cancelled one left off.

`let_stopped` converts a cancellation into a new computation. The function receives no arguments - the stopped channel carries no data - and returns a sender whose completion replaces the stopped signal. The reader will recognize the pattern: each `let_*` variant handles one channel and routes its result into a new sender. Regular C++ has no cancellation equivalent.

---

6. Execution Contexts

Work has to run somewhere. These algorithms decide where - which thread, which I/O loop, which GPU stream owns the next step of the computation.

Theory: Plotkin [7] showed that CPS makes evaluation order explicit in the term structure; the scheduler algorithms make the evaluation *context* - the execution resource - explicit as well. In the indexed-monad pattern, a grade or index tracks a resource annotation through a computation; here the index is the scheduler. `schedule` introduces the index. `continues_on` changes it. `on` brackets a sub-computation with a context switch and its inverse - a `reset`/`shift` pair in Danvy and Filinski's [12] framework that delimits the region where the alternate context applies. All six algorithms in this section are morphisms in the

indexed category of execution contexts, composing context transitions the way CPS composes evaluation steps.

6.1 `schedule`

`schedule(sch)` produces a sender that, when started, completes on the execution context of `sch`, the specified scheduler.

```
auto sndr = schedule(pool.get_scheduler())
          | then([] {
                return handle_request();
            });
```

The reader enters an execution context. `schedule` is the door - it produces a sender that, when started, transitions to the scheduler's context. Everything piped after it runs there.

`schedule` creates a sender from a scheduler. The sender completes with no values - it serves purely as a transition point. Work piped after `schedule` runs on the scheduler's context.

The equivalent program:

```
pool.post([&] {
    handle_request();
});
```

6.2 `starts_on`

`starts_on(sch, sndr)` arranges for `sndr` to begin execution on the scheduler `sch`.

```
auto sndr = starts_on(
    pool.get_scheduler(),
    just(request) | then([](auto req) {
        return handle(req);
    }));
```

The reader has a sender and wants it to run somewhere specific. `starts_on` arranges the introduction - the sender begins its work on the chosen scheduler.

The equivalent program:

```
pool.post([&] {
    auto result = handle(request);
});
```

6.3 `continues_on`

`continues_on(sch)` transitions execution to the scheduler `sch` after the predecessor completes.

```
auto sndr =
    starts_on(io_ctx.get_scheduler(),
              async_read(socket, buf))
  | continues_on(pool.get_scheduler())
  | then([](auto data) {
        return parse(data);
    });
```

A graceful transition. When the previous step finishes on one context, `continues_on` moves execution to another. The data carries over; the thread changes. This is how a pipeline moves work between contexts - read on the I/O thread, parse on the thread pool.

The equivalent program:

```
auto data = read(socket);
pool.post([&] {
    auto result = parse(data);
});
```

6.4 on

`on(sch, sndr)` runs `sndr` on the scheduler `sch`, then returns execution to the caller's context.

```
auto sndr = on(
    pool.get_scheduler(),
    just(request) | then([](auto req) {
        return handle(req);
    }));
```

A round trip - work runs on another context and the result comes back. The reader does not need to manage the return; it is handled.

The equivalent program:

```
auto result = handle(request);
```

6.5 `affine`

`affine(sndr)` ensures `sndr` completes on the receiver's start scheduler, skipping the transition if the sender already completes there. The scheduler is obtained from `get_start_scheduler(get_env(rcvr))` at `connect` time.

```
auto sndr = some_operation()
          | affine;
```

The optimization the reader would hope for - if the sender already completes on the start scheduler, the transition is skipped entirely. No wasted work.

`affine` is the scheduler affinity primitive, redesigned by P3941R4 [16] and renamed from `affine_on` by P4151R1 [17]. It behaves like `continues_on` but avoids the scheduling overhead when the predecessor already completes on the start scheduler. The scheduler is not passed as an argument - it comes from the receiver's environment. This is what makes scheduler affinity practical for coroutine `task` - the `await_transform` injects `affine` around every `co_await`ed sender. We will return to this in Section 12.

The equivalent program:

```
auto result = some_operation();
```

6.6 `schedule_from`

`schedule_from(sch, sndr)` ensures that all of `sndr`'s completion signals - value, error, and stopped - arrive on the scheduler `sch`.

```
auto sndr = schedule_from(
    pool.get_scheduler(),
    async_read(socket, buf));
```

An implementer's tool. Most readers will not need this directly, but it is satisfying to know it exists - precise control over where all three completion channels arrive. Where `continues_on` transitions the value channel, `schedule_from` transitions all three.

The equivalent program:

```
auto data = read(socket);
```

---

7. Environment

A pipeline sometimes needs to ask its surroundings a question - what scheduler am I running on, what allocator should I use, has anyone asked me to stop? The environment carries those answers, and these algorithms read from it and write to it.

Theory: In Wadler's formulation [15], the Reader monad R(A) = E -> A threads an environment E through a computation without mentioning it in the function signature. `read_env` is `ask` - the projection that extracts a value from E. `write_env` is `local` - the combinator that applies a transformation to the environment before passing it to the inner computation. The sender environment - scheduler, allocator, stop token - is E. The receiver carries it. The receiver's `get_env` is the elimination form that opens E for inspection.

7.1 `read_env`

`read_env(query)` creates a sender that reads the value associated with `query` from the receiver's environment and completes with it on the value channel.

```
auto sndr = read_env(get_scheduler)
          | let_value([](auto sch) {
                return schedule(sch)
                     | then([] {
                           return load_config();
                       });
            });
```

The pipeline asks its context a question. What scheduler am I on? What allocator should I use? The environment carries these answers, and `read_env` retrieves them.

`read_env` is the introspection primitive. The sender does not carry the environment value itself - it reads it from the receiver at connection time, when the environment is known. This enables context-dependent behavior: the same pipeline can behave differently depending on which scheduler, allocator, or stop token the enclosing context provides. The composition reads naturally once the receiver's role as environment carrier is understood.

The equivalent program:

```
auto config = load_config();
```

7.2 `write_env`

`write_env(env, sndr)` wraps `sndr` so that downstream receivers see the additional or overridden entries in `env`.

```
auto sndr = write_env(
    make_env(get_allocator, pool_alloc),
    process_file(path));
```

The reader can shape the environment for downstream operations. Override the allocator, substitute a scheduler, inject custom context - the downstream pipeline sees the modified environment without any change to its code.

`write_env` overlays entries onto the receiver's environment. The wrapped sender and all of its children see the modified environment. This is how a pipeline provides a custom allocator, overrides a scheduler, or injects application-specific context without threading parameters through every function signature.

The equivalent program:

```
auto result = process_file(path);
```

7.3 `unstoppable`

`unstoppable(sndr)` wraps `sndr` so that it does not receive stop requests from the parent context.

```
auto sndr = unstoppable(
    commit_transaction(db, txn));
```

Some operations must not be interrupted. `unstoppable` shields a sender from cancellation - the operation runs to completion regardless of what the parent pipeline decides. Transaction commits, resource cleanup, finalization steps - these are the operations that earn this protection.

Theory: This is effect masking in the algebraic-effects framework [12]. The stopped channel is removed from the effect signature of the inner computation - the handler that would intercept the cancellation `shift` is replaced by the identity, absorbing the effect and producing no observable consequence. The completion signatures of the wrapped sender lose `set_stopped_t()`; the type system records the masking.

The equivalent program:

```
commit_transaction(db, txn);
```

---

8. Structured Concurrency

Run things at the same time, wait for all of them, and know that nothing outlives its scope. These algorithms give C++ something its statements alone cannot express - concurrent execution with structured lifetime guarantees.

Theory: In Milner's process algebra [18], parallel composition P | Q runs two processes independently and synchronizes them at a rendezvous. `when_all(S1, S2, ..., Sn)` is the n-ary tensor product in a symmetric monoidal category of senders, with a synchronization barrier at the join. The structured-lifetime guarantee - no child sender outlives the join point - is a region discipline imposed on the tensor: the scope of the product is delimited, and deallocation occurs at the closing delimiter. `when_all_with_variant` is the same tensor product composed with a coproduct injection on each factor, accommodating heterogeneous value completions.

8.1 when_all

`when_all(sndrs...)` starts all of the given senders concurrently and completes when every sender has completed. The value completions are concatenated.

```
auto sndr = when_all(
    fetch_user(user_id),
    fetch_orders(user_id),
    fetch_preferences(user_id))
  | then([](auto user, auto orders,
            auto prefs) {
        return build_profile(
            user, orders, prefs);
    });
```

The reader will recognize this immediately - run everything at once, wait for all of them. If any fails, cancel the rest. This is structured concurrency, and it is one of the genuine strengths of the sender model.

`when_all` is genuine structured concurrency. All child senders start together, run concurrently, and the join point is the `when_all` sender's completion. If any child completes with an error or is stopped, the remaining children are cancelled. The lifetime guarantee is structural - no child outlives the join point. Once the programmer has internalized the pattern, the intent is clear: three fetches, run concurrently, results combined.

The equivalent program:

```
auto user = fetch_user(user_id);
auto orders = fetch_orders(user_id);
auto prefs = fetch_preferences(user_id);
auto profile =
    build_profile(user, orders, prefs);
```

8.2 when_all_with_variant

`when_all_with_variant(sndrs...)` is semantically equivalent to `when_all(into_variant(sndrs)...)`. It allows child senders with different value completion signatures.

```
auto sndr = when_all_with_variant(
    fetch_json(url_a),
    fetch_binary(url_b))
  | then([](auto json_var, auto bin_var) {
        return combine(json_var, bin_var);
    });
```

The same structured concurrency, but the senders may produce different types. Each result is wrapped in a variant - a natural accommodation for heterogeneous work.

Each child of `when_all` must have exactly one value completion signature, though children may differ from each other in what that signature is - the results are concatenated in input order. `when_all_with_variant` relaxes the single-signature constraint: children with multiple possible value completion paths have each path wrapped in a `variant`, and the programmer destructures the variants at the join point.

The equivalent program:

```
auto json = fetch_json(url_a);
auto binary = fetch_binary(url_b);
auto result = combine(json, binary);
```

8.3 `split`

`split(sndr)` converts a single-shot sender into a multi-shot sender. The result is shared: multiple downstream pipelines can consume the same sender's completion.

```
auto shared = split(fetch_data(url));
auto sndr = when_all(
    shared | then([](auto data) {
        return analyze(data);
    }),
    shared | then([](auto data) {
        return archive(data);
    }));
```

A sender's result, shared among multiple consumers. The sender runs once; everyone gets a copy. The reader will find this a natural way to express data sharing in a pipeline.

`split` allocates shared state - the sender's result is stored once and distributed to all consumers. This is the sender equivalent of binding a value to a local variable and using it in multiple expressions. The allocation is the cost; the sharing is the benefit.

Theory: In Girard's linear logic [19], a linear resource is consumed exactly once - and senders are linear by default, consumed by a single `connect`. `split` applies the contraction structural rule, Delta : A -> A (x) A, duplicating the resource at the cost of an allocation. In linear logic contraction is not free; it must be explicitly introduced. The allocation is the computational witness of that introduction - reference-counted shared state that permits multiple consumers where the type system otherwise forbids it.

The equivalent program:

```
auto data = fetch_data(url);
auto analysis = analyze(data);
auto archived = archive(data);
```

---

9. Signal Adaptation

Sometimes the shape of a completion does not quite fit what the next stage expects. These algorithms reshape it - wrapping values, collapsing channels, making everything line up.

Theory: The signal adaptation algorithms are natural transformations on the completion-signature functor [12]. The sender's completion type is a coproduct - Value + Error + Stopped - and the adaptation algorithms are morphisms in the category of coproducts that reshape the signature set. `into_variant` applies the universal property of the coproduct, mapping an n-ary sum into a single tagged union. `stopped_as_optional` and `stopped_as_error` collapse one summand into another, reducing the coproduct's arity. These are precisely the effect-handler signature transformations in the algebraic-effects literature - an outer handler that intercepts one effect and re-emits it on a different channel.

9.1 `into_variant`

`into_variant` adapts a sender with multiple value completion signatures into one that completes with a single `variant` of `tuple`s.

```
auto sndr = query_database(stmt)
          | into_variant;
// if the query can complete with either
// set_value(row) or set_value(int),
// now completes with
// set_value(variant<tuple<row>,
//                   tuple<int>>)
```

When a sender might produce different types of values, `into_variant` wraps them all into a single variant type - a unification step that lets the rest of the pipeline proceed with one type.

`into_variant` collapses heterogeneous value completions into a single variant type. The result is a sender with exactly one value completion signature. This is used when a sender's multiple value paths must be unified - for example, before passing to `when_all`, which requires a single value completion signature from each child.

The equivalent program:

```
auto result = query_database(stmt);
```

9.2 `stopped_as_optional`

`stopped_as_optional` converts a sender's value completion into an `optional` and its stopped completion into an empty `optional`.

```
auto sndr = dequeue(work_queue)
          | stopped_as_optional;
// value(item)   -> value(optional(item))
// stopped()     -> value(optional())
```

Cancellation becomes an empty optional; values become engaged optionals. Both paths merge onto the value channel - the reader handles both cases with a single `if`.

`stopped_as_optional` is the bridge between the stopped channel and ordinary value processing. A queue that reports "closed" via `set_stopped` becomes a sender that completes with an empty `optional` - the programmer handles both cases in the value channel with a single `if`.

The equivalent program:

```
auto item = dequeue(work_queue);
// returns optional<item_type>
```

9.3 `stopped_as_error`

`stopped_as_error(e)` converts a sender's stopped completion into an error completion carrying `e`.

```
auto sndr = timed_operation(deadline)
          | stopped_as_error(
                make_error_code(
                    errc::timed_out));
```

Cancellation becomes an error. The two failure paths unify into one, and the downstream error handling takes over.

`stopped_as_error` reclassifies cancellation as an error. The stopped channel is eliminated from the completion signatures and replaced by an error. This is used when the downstream pipeline handles errors but not cancellation - the reclassification unifies the two failure paths.

The equivalent program:

```
auto result = timed_operation(deadline);
if (!result)
    throw system_error(
        make_error_code(errc::timed_out));
```

---

10. Data Parallelism

Do the same thing to every element. The `bulk` family is a parallel for-loop expressed as a sender - one function, applied uniformly across an index space.

Theory: Hillis and Steele [20] showed that the foundational primitive of data-parallel computation is the uniform application of a function across a set of indices - the data-parallel `map`. The `bulk` family is this primitive realized as a sender algorithm, with the execution policy selecting the evaluation strategy (sequential fold, parallel map, SIMD broadcast) and the scheduler selecting the hardware. The three variants - `bulk`, `bulk_chunked`, `bulk_unchunked` - partition the index space at different granularities, but the algebraic structure is the same indexed traversal.

10.1 `bulk`

`bulk(policy, shape, f)` invokes `f(i, args...)` for each index `i` in the range `[0, shape)`, where `policy` is an execution policy (`std::par`, `std::seq`), `shape` is the number of invocations, and `args` are the values produced by the predecessor sender.

```
auto sndr = just(records)
          | bulk(std::par, records.size(),
                [](size_t i, auto& recs) {
                    recs[i].score =
                        compute_score(recs[i]);
                });
```

A parallel for-loop. The function runs once per index, and the execution policy controls whether the invocations overlap.

`bulk` is the data-parallel primitive. The execution policy controls whether the invocations may run concurrently. The `parallel_scheduler` (P2079R10, "Parallel Scheduler") [21] provides a customized implementation that executes the index space across the system thread pool. The design is remarkably expressive - a single algorithm covers sequential iteration, parallel map, and GPU-dispatched computation depending on the scheduler and policy.

The equivalent program:

```
std::for_each(std::execution::par,
    records.begin(), records.end(),
    [](auto& r) {
        r.score = compute_score(r);
    });
```

10.2 `bulk_chunked`

`bulk_chunked(policy, shape, f)` partitions the index space into chunks and invokes `f` once per chunk with the chunk's index range, where `policy` is the execution policy, `shape` is the total index count, and `f` receives a range of indices plus the predecessor's values.

```
auto sndr = just(data)
          | bulk_chunked(std::par,
                data.size(),
                [](auto chunk, auto& d) {
                    for (auto i : chunk)
                        d[i] = transform(d[i]);
                });
```

The same parallel work, but the implementation groups indices into chunks. Each invocation processes a range - ideal for cache-friendly access patterns and vectorization.

`bulk_chunked` gives the implementation control over how the index space is divided. The `parallel_scheduler` uses this to distribute work across threads in chunks sized for the hardware. The callable receives a range of indices rather than a single index, enabling vectorization and cache-friendly access patterns within each chunk.

The equivalent program:

```
std::for_each(std::execution::par,
    data.begin(), data.end(),
    [](auto& x) { x = transform(x); });
```

10.3 `bulk_unchunked`

`bulk_unchunked(policy, shape, f)` invokes `f` once per index with no chunking guarantees, where `policy`, `shape`, and `f` have the same roles as in `bulk`.

```
auto sndr = just(pixels)
          | bulk_unchunked(std::par,
                pixels.size(),
                [](size_t i, auto& px) {
                    px[i] = apply_filter(px[i]);
                });
```

One invocation per index, no chunking. The simplest mental model for parallel iteration.

`bulk_unchunked` is the unchunked counterpart. The callable receives one index at a time. The implementation may still batch invocations internally, but the callable's interface is per-index. Each piece of the `bulk` family fits together with the precision one expects from a well-engineered system: `bulk` dispatches to `bulk_chunked` by default, which the scheduler can further specialize.

The equivalent program:

```
std::for_each(std::execution::par,
    pixels.begin(), pixels.end(),
    [](auto& p) { p = apply_filter(p); });
```

---

11. Async Scopes

When work is spawned dynamically, someone has to keep track of it. These algorithms tie each sender's lifetime to a scope, so that shutdown waits for everything to finish.

Theory: In Tofte and Talpin's framework [22], a region is a lexically delimited lifetime boundary that no allocation may outlive. A `counting_scope` is a region - the join operation is its closing delimiter. `associate` is allocation within the region: the sender's lifetime is bounded by the region's scope. `spawn_future` is the same region binding with a return channel - the continuation that delivers the result when the inner computation completes. The structured guarantee is enforced dynamically (the count reaches zero) rather than statically (a type-level region variable), but the invariant is identical: no child outlives its region.

11.1 `associate`

`associate(token, sndr)` ties a sender's lifetime to a scope. The scope will not complete its join until the associated sender has completed.

```
counting_scope scope;
auto token = scope.get_token();

for (auto& conn : accepted_connections) {
    auto sndr = associate(token,
        starts_on(pool.get_scheduler(),
                  handle_connection(conn)));
    // sndr is fire-and-forget within the scope
}
// scope.join() waits for all connections
```

A sender's lifetime is tied to a scope. The scope does not shut down until the sender finishes - no dangling work, no orphaned operations.

`associate` is the structured spawn. The sender runs independently - it is not piped into a continuation - but its lifetime is bounded by the scope. When the scope joins, all associated senders must have completed. This is how a server manages connection lifetimes: each connection is associated with the scope, and shutdown waits for all connections to drain.

The equivalent program:

```
for (auto& conn : accepted_connections)
    pool.post([&] {
        handle_connection(conn);
    });
pool.join();
```

11.2 spawn_future

`spawn_future(token, sndr)` spawns a sender into a scope and returns a new sender that completes with the spawned sender's result.

```
auto sndr = spawn_future(token,
    starts_on(pool.get_scheduler(),
              fetch_data(key)))
  | then([](auto data) {
        return process(data);
    });
```

Start work in the background and keep a handle to the result. The work runs now; the result arrives when the reader asks for it.

`spawn_future` is `associate` with a return channel. The spawned sender begins executing immediately, and the returned sender completes with the result when it is ready. This is the sender equivalent of `std::async` - fire off work, get a handle to the result, consume it later.

The equivalent program:

```
auto future = std::async(
    std::launch::async,
    [&] { return fetch_data(key); });
auto result = process(future.get());
```

---

12. The task Coroutine Type

P3552R3 [2] adds `execution::task<T, C>` to C++26 - a coroutine type that is also a sender. P3796R1 ("Coroutine Task Issues") [23] catalogs open design concerns. The `task` is the bridge between coroutine-style `co_await` and the sender pipeline model. The integration between the two worlds is seamless: a `task` can `co_await` any sender, and a `task` can be used as a sender in any pipeline.

12.1 task<T>

`task<T>` declares a coroutine that produces a value of type `T` on the value channel when it completes.

```
task<int> the_answer() {
    co_return 42;
}

int main() {
    auto [v] =
        sync_wait(the_answer()).value();
    // v == 42
}
```

A `task` is lazy - the coroutine body does not execute until the task is connected to a receiver and started. It is a sender: it can be piped, composed, passed to `sync_wait`, or used as a child of `when_all`. The completion signatures are `set_value_t(T)`, `set_error_t(exception_ptr)`, and `set_stopped_t()`.

12.2 co_await a Sender

Inside a `task`, any sender can be `co_await`ed. The sender is connected to an internal receiver, started, and the result is delivered as the value of the `co_await` expression.

```
task<int> add_one() {
    int v = co_await just(41);
    co_return v + 1;
}
```

When the sender completes with `set_value(args...)`, the `co_await` expression produces the values. A single value is returned directly. Multiple values are returned as a `tuple`. A sender with no value arguments produces `void`. If the sender completes with `set_error`, the error is thrown as an exception. If it completes with `set_stopped`, the coroutine is cancelled without resuming.

12.3 task_scheduler

`task_scheduler` is a type-erased scheduler used by `task` for scheduler affinity. When a `task` is connected to a receiver, the scheduler is obtained from the receiver's environment via `get_start_scheduler(get_env(rcvr))` and stored in the `task_scheduler`.

```
task<void> work() {
    co_await async_read(socket, buf);
    // resumes on the task's scheduler
    process(buf);
}

sync_wait(
    starts_on(pool.get_scheduler(),
              work()));
```

The `task_scheduler` uses small-object optimization to avoid allocation for common scheduler types. The type erasure is the cost of not knowing the scheduler type at coroutine definition time. The programmer has everything they need: the scheduler is obtained automatically, the affinity is maintained transparently, and the type erasure overhead is minimal.

12.4 `inline_scheduler`

`inline_scheduler` completes immediately on the calling thread. Using it as the task's scheduler type disables scheduler affinity.

```
struct no_affinity {
    using start_scheduler_type = inline_scheduler;
};

task<void, no_affinity> fast_path() {
    co_await async_op();
    // no rescheduling - resumes wherever
    // the sender completed
}
```

Disabling affinity removes the rescheduling overhead at the cost of the guarantees affinity provides. The programmer who understands the implications is free to make this choice.

12.5 Scheduler Affinity

The `task`'s promise type injects `affine` around every `co_await`ed sender via `await_transform`. The effect: after each `co_await`, execution resumes on the task's scheduler regardless of where the sender completed.

```
task<void> affine_demo() {
    // running on pool's scheduler
    co_await async_read(socket, buf);
    // still on pool's scheduler
    auto result = parse(buf);
    co_await async_write(socket, result);
    // still on pool's scheduler
}
```

Scheduler affinity means the programmer can reason about execution context the same way they reason about synchronous code - each line runs on the same context as the line before it. The `affine` insertion is invisible. The rescheduling happens only when necessary - if the sender already completes on the correct scheduler, `affine` skips the transition.

12.6 Allocator Support

The context parameter `C` declares the allocator type via `C::allocator_type`. Two allocator paths exist post-P3980R1 [24]: the frame allocator is specified at the call site via `allocator_arg` (which must be the first parameter), and the environment allocator is obtained from `get_allocator(get_env(rcvr))` at `connect` time.

```
struct alloc_ctx {
    using allocator_type =
        pmr::polymorphic_allocator<byte>;
};

task<int, alloc_ctx> allocated_work(
        allocator_arg_t,
        pmr::polymorphic_allocator<byte>,
        int input) {
    co_return input * 2;
}

pmr::monotonic_buffer_resource pool;
auto sndr = allocated_work(
    allocator_arg,
    pmr::polymorphic_allocator<byte>{&pool},
    21);
```

The frame allocator controls coroutine frame allocation at the call site. The environment allocator - obtained from the receiver at `connect` time - is available to child operations via `get_allocator` and propagates through the receiver chain. The design supports environments where heap allocation is prohibited.

12.7 `co_yield with_error(e)`

`co_yield with_error(e)` reports an error on the error channel without throwing an exception.

```
task<void> validate(request const& req) {
    if (!req.valid())
        co_yield with_error(
            make_error_code(
                errc::invalid_argument));
    co_return;
}
```

This is how a `task` delivers a typed error to the sender composition algebra without relying on exceptions. The coroutine is suspended and completes with `set_error(e)`. The downstream pipeline handles the error through `upon_error` or `let_error`. This is a feature of the design: errors can be reported precisely, with the type preserved, and the composition algebra participates.

12.8 Cancellation

`co_await just_stopped()` cancels the coroutine. The coroutine completes with `set_stopped()`.

```
task<void> check_cancel() {
    auto token = co_await
        read_env(get_stop_token);
    if (token.stop_requested())
        co_await just_stopped();
    co_return;
}
```

Any sender that completes with `set_stopped` cancels the `task`. The coroutine is never resumed - local variables are destroyed and the task completes on the stopped channel. The `task`'s stop token is linked to the parent context's stop token, so cancellation propagates structurally.

12.9 Error Channel Mapping

When a `task` `co_await`s a sender that completes with `set_error(ec)`, the error is delivered as a thrown exception.

```
task<void> error_mapping_demo() {
    try {
        co_await async_connect(
            socket, endpoint);
    } catch (system_error const& e) {
        // set_error(ec) became throw
        log("connect failed: {}",
            e.code().message());
    }
}
```

This is how the three-channel model composes with coroutines. The value channel becomes the `co_await` return value. The error channel becomes a thrown exception. The stopped channel becomes cancellation. The mapping is clean and the programmer can use familiar `try`/`catch` for error handling. The composition algebra built on `set_error` integrates with coroutines through the exception mechanism - the two error models meet at the `co_await` boundary.

12.10 Compound Results

When an I/O operation returns both a status code and a byte count, the compound result can stay on the value channel. Both values arrive at the `co_await` expression.

```
task<void> read_with_status() {
    auto [ec, n] = co_await
        async_read(socket, buf);
    if (!ec) {
        process(buf, n);
    } else if (
            ec == errc::connection_reset) {
        cleanup(n);
    }
}
```

Both values are visible. The programmer branches with `if`. The composition algebra - `retry`, `upon_error`, `when_all` - does not participate in this dispatch, because the result stayed on the value channel. This is the trade-off: data preservation or composition algebra participation, not both simultaneously. The programmer always has the option of writing sender code directly instead of using the coroutine, gaining access to the full composition algebra at the cost of the programming model documented in Sections 3 through 11.

Alternatively, the compound result can be bundled into the error type:

```
struct io_result {
    error_code ec;
    size_t bytes;
};

task<void> bundled_error() {
    auto [ec, n] = co_await
        async_read(socket, buf);
    if (ec)
        co_yield with_error(
            io_result{ec, n});
    process(buf, n);
}
```

The composition algebra now participates - `retry` sees the error, `upon_error` can handle it. Every `upon_error` handler downstream must accept `io_result` alongside any other error types in the pipeline. The data survives the channel crossing because it is inside the error object. Both approaches are legitimate design choices. The programmer evaluates the trade-off for the domain at hand.

12.11 Pipelines Inside Coroutines

A sender pipeline can be `co_await`ed as a single expression inside a `task`.

```
task<string> fetch_and_transform(url u) {
    auto result = co_await (
        async_fetch(u)
      | then([](auto resp) {
            return resp.body();
        })
      | then([](auto body) {
            return compress(body);
        }));
    co_return result;
}
```

The pipeline is composed using the pipe operator, then `co_await`ed as a unit. The `task` provides the execution context; the pipeline provides the composition. The integration between coroutines and senders is seamless - each model contributes its strength.

12.12 Tasks as Senders

A `task` is a sender. It can be used anywhere a sender is expected - piped into `then`, passed to `when_all`, composed with any algorithm from Sections 3 through 11.

```
auto sndr = fetch_and_transform(my_url)
          | then([](string compressed) {
                return store(compressed);
            });
auto [stored] =
    sync_wait(std::move(sndr)).value();
```

The `task` enters the sender world as a first-class participant. The coroutine's completion signatures become the sender's completion signatures. The scheduler affinity, allocator support, and error handling all compose through the sender protocol. The programmer has everything they need.

---

13. Composition

The final section demonstrates how senders and coroutines compose together in a realistic program. The example uses four layers, each building on the previous, combining the algorithms from this tutorial into a single system. The resulting program is readable - each layer is a natural application of the patterns introduced earlier.

13.1 Sensor Fusion

A `task` that `co_await`s a structured-concurrency pipeline to read three sensors in parallel and fuse the results.

```
task<sensor_data> read_sensors(
        sensor_array const& sensors) {
    auto [lidar, radar, camera] = co_await
        when_all(
            starts_on(
                sensors.lidar_ctx(),
                async_read(
                    sensors.lidar()))
            | then([](raw_data raw) {
                  return decode_lidar(raw);
              }),
            starts_on(
                sensors.radar_ctx(),
                async_read(
                    sensors.radar()))
            | then([](raw_data raw) {
                  return decode_radar(raw);
              }),
            starts_on(
                sensors.camera_ctx(),
                async_read(
                    sensors.camera()))
            | then([](raw_data raw) {
                  return decode_camera(raw);
              }));
    co_return fuse(lidar, radar, camera);
}
```

Three sensors, three execution contexts, one `when_all`. The `task` provides scheduler affinity; the pipeline provides concurrency. The fused result is a single `sensor_data` value.

13.2 Collision Detection

The `task` from 13.1 is used as a sender inside a pipeline that evaluates whether immediate braking is required.

```
auto collision_pipeline(
        sensor_array const& sensors,
        vehicle_state const& state) {
    return read_sensors(sensors)
         | then([&](sensor_data data) {
               return detect_obstacles(
                   data, state);
           })
         | let_value(
               [](obstacle_set obstacles) {
               if (obstacles.imminent())
                   return just(brake_cmd{
                       obstacles.severity()});
               return just(
                   brake_cmd::none());
           });
}
```

The `task` is a sender. The pipeline consumes it with `then` and `let_value`. The branching inside `let_value` returns different `brake_cmd` values - both branches return the same sender type, so no type erasure is needed.

13.3 Actuator Command

A `task` that `co_await`s the collision pipeline and sends brake commands on a real-time scheduler.

```
task<actuator_result> actuate_brakes(
        sensor_array const& sensors,
        vehicle_state const& state,
        brake_controller& brakes) {
    auto cmd = co_await
        collision_pipeline(sensors, state);
    if (cmd.severity() > 0) {
        co_await starts_on(
            brakes.realtime_ctx(),
            async_write(
                brakes.channel(), cmd));
        co_return
            actuator_result::applied(cmd);
    }
    co_return actuator_result::none();
}
```

The pipeline from 13.2 is `co_await`ed inside a `task`. The brake command is issued on a real-time scheduler - a context transition from the sensor-processing context to the actuator context.

13.4 Failover

The `task` from 13.3 is used as a sender inside a pipeline that handles sensor failure with emergency braking.

```
auto safety_controller(
        sensor_array const& sensors,
        vehicle_state const& state,
        brake_controller& brakes,
        counting_scope& scope) {
    return associate(
        scope.get_token(),
        actuate_brakes(
            sensors, state, brakes)
      | let_error(
            [&](error_code ec) {
            return starts_on(
                brakes.realtime_ctx(),
                async_write(
                    brakes.channel(),
                    brake_cmd::emergency()))
              | then([ec](auto) {
                    log("sensor failure: "
                        "{}, emergency "
                        "brake applied",
                        ec);
                    return actuator_result
                        ::emergency();
                });
        })
      | upon_stopped([&] {
            log("controller stopped");
            return
                actuator_result::none();
        }));
}
```

Four layers. A sender pipeline containing a `task` containing a sender pipeline containing a `task` containing a sender pipeline. Each layer is individually reasonable - a natural application of the algorithms this tutorial has introduced. The structure is intuitive. The composition follows the patterns established in the preceding sections. The tutorial has, in a sense, been unnecessary - the Sub-Language is its own best teacher.

13.5 The Equivalent Program

```
task<actuator_result> safety_controller(
        sensor_array const& sensors,
        vehicle_state const& state,
        brake_controller& brakes) {
    try {
        auto data = co_await
            read_all_sensors(sensors);
        auto obs =
            detect_obstacles(data, state);
        if (obs.imminent()) {
            auto cmd =
                brake_cmd{obs.severity()};
            co_await brakes.apply(cmd);
            co_return
                actuator_result::applied(
                    cmd);
        }
        co_return actuator_result::none();
    } catch (...) {
        co_await
            brakes.emergency_stop();
        co_return
            actuator_result::emergency();
    }
}
```

---

14. Real World Examples

The following examples are drawn from the stdexec [25] reference implementation (whose algorithm customization model is addressed by P3826R5 ("Fix Sender Algorithm Customization") [26]) and the sender-examples [27] repository. They demonstrate patterns that combine the algorithms from this tutorial into working code at a scale the reader may encounter in practice.

14.1 The Backtracker

A recursive tree search with continuation-passing failure recovery, drawn from backtrack.cpp [28] in the sender-examples repository. The `search_tree` function traverses a binary tree looking for a node that satisfies a predicate. When a subtree yields no match, the failure continuation - itself a sender - takes over.

```
using any_node_sender = any_sender_of<
    set_value_t(tree::NodePtr),
    set_stopped_t(),
    set_error_t(exception_ptr)>;

auto search_tree(
        auto test,
        tree::NodePtr tree,
        scheduler auto sch,
        any_node_sender&& fail)
    -> any_node_sender {
    if (tree == nullptr)
        return std::move(fail);
    if (test(tree))
        return just(tree);
    return on(sch, just())
      | let_value(
            [=, fail = std::move(fail)]()
                mutable {
            return search_tree(
                test,
                tree->left(),
                sch,
                on(sch, just())
                  | let_value(
                        [=, fail =
                            std::move(fail)]()
                            mutable {
                        return search_tree(
                            test,
                            tree->right(),
                            sch,
                            std::move(fail));
                    }));
        });
}
```

The function is recursive Kleisli composition with type-erased return. 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. The `any_sender_of<>` type alias uses a type-erased sender facility [29] from stdexec [25] - it is not part of C++26. Without type erasure, the recursive return type would be infinite.

The equivalent program:

```
auto search_tree(auto test,
        tree::NodePtr node)
    -> tree::NodePtr {
    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());
}
```

14.2 The `retry` Algorithm

A complete sender algorithm implementation from retry.hpp [30] in the stdexec [25] examples. Where the preceding sections show how to *use* sender algorithms, this example shows how to *implement* one. The `retry` algorithm re-executes a sender whenever it completes with an error. The code uses stdexec extension types (`exec::receiver_adaptor`) and macros (`STDEXEC_TRY`, `STDEXEC_CATCH_ALL`) that are not part of the C++26 standard library. Some names appear unqualified because the original source uses `using namespace` directives omitted here for brevity.

```
template <class Fun>
    requires std::is_nothrow_move_constructible_v<
        Fun>
struct _conv {
    Fun f_;
    explicit _conv(Fun f) noexcept
        : f_(static_cast<Fun&&>(f)) {}
    operator std::invoke_result_t<Fun>() && {
        return static_cast<Fun&&>(f_)();
    }
};

template <class S, class R>
struct _op;

template <class S, class R>
struct _retry_receiver
    : exec::receiver_adaptor<
          _retry_receiver<S, R>> {
    _op<S, R>* o_;

    auto base() && noexcept -> R&& {
        return static_cast<R&&>(o_->r_);
    }
    auto base() const& noexcept
        -> R const& {
        return o_->r_;
    }
    explicit _retry_receiver(
        _op<S, R>* o) : o_(o) {}

    template <class Error>
    void set_error(Error&&) && noexcept {
        o_->_retry();
    }
};

template <class S, class R>
struct _op {
    S s_;
    R r_;
    std::optional<connect_result_t<
        S&, _retry_receiver<S, R>>> o_;
```

```
    _op(S s, R r)
        : s_(static_cast<S&&>(s))
        , r_(static_cast<R&&>(r))
        , o_{_connect()} {}
    _op(_op&&) = delete;

    auto _connect() noexcept {
        return _conv{[this] {
            return stdexec::connect(
                s_,
                _retry_receiver<S, R>{
                    this});
        }};
    }

    void _retry() noexcept {
        STDEXEC_TRY {
            o_.emplace(_connect());
            stdexec::start(*o_);
        }
        STDEXEC_CATCH_ALL {
            stdexec::set_error(
                static_cast<R&&>(r_),
                std::current_exception());
        }
    }

    void start() & noexcept {
        stdexec::start(*o_);
    }
};

template <class S>
struct _retry_sender {
    using sender_concept =
        stdexec::sender_t;
    S s_;

    explicit _retry_sender(S s)
        : s_(static_cast<S&&>(s)) {}

    template <class... Args>
    using _error =
        completion_signatures<>;
    template <class... Args>
```

```
    using _value =
        completion_signatures<
            set_value_t(Args...)>;

    template <class... Env>
    static consteval auto
    get_completion_signatures()
        -> transform_completion_signatures<
            completion_signatures_of_t<
                S, Env...>,
            completion_signatures<
                set_error_t(
                    exception_ptr)>,
            _value, _error> {
        return {};
    }

    template <class R>
    auto connect(R r) &&
        -> _op<S, R> {
        return {
            static_cast<S&&>(s_),
            static_cast<R&&>(r)};
    }

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

template <class S>
auto retry(S s) -> sender auto {
    return _retry_sender{
        static_cast<S&&>(s)};
}
```

The implementation demonstrates 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 equivalent program:

```
template <class T, class F>
auto retry(F make_sender) -> task<T> {
    for (;;) {
        try {
            co_return
                co_await make_sender();
        } catch (...) {}
    }
}
```

---

15. Conclusion

This tutorial has progressed from the simplest sender algorithm - `just(42)` - to a four-layer composition of sender pipelines and coroutines, and then to real-world algorithm implementations drawn from production repositories. Some readers may find the later examples challenging. There is no rush. The examples reward careful study and repeated reading. The Sender Sub-Language is C++26's asynchronous programming model - the standard's answer to structured concurrency and heterogeneous execution. The programmer who masters it has mastered the model the standard provides. With patience and practice, every pattern in this tutorial becomes familiar.

There is something to be said for one size fits all. Every C++ developer learns the same abstractions. Every codebase uses the same patterns. Every team shares the same vocabulary for asynchronous programming. The consistency is real, and the predictability is a gift. For the programmer whose domain would benefit from a different model?

The ecosystem provides. [31]

Acknowledgments

The authors thank the P2300R0 authors - Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, and Bryce Adelstein Lelbach - for building `std::execution` and for the theoretical depth that informs this tutorial. The authors also thank Dietmar Kühl and Maikel Nadolski for the `task` coroutine type, Steve Downey for the sender-examples that informed our understanding, and Peter Dimov for editorial guidance.

References

[1] N5014. Thomas Köppe (ed.). "Working Draft, Standard for Programming Language C++." 2025.

[2] P3552R3. Dietmar Kühl, Maikel Nadolski. "Add a Coroutine Task Type." 2025.

[3] Capy. Vinnie Falco. Coroutine I/O primitives.

[4] Corosio. Vinnie Falco. Coroutine-native networking library.

[5] Guy Steele and Gerald Sussman. *The Lambda Papers*. MIT AI Memo series, 1975-1980.

[6] Eugenio Moggi. "Notions of Computation and Monads". *Information and Computation*, 1991.

[7] Gordon Plotkin. "Call-by-name, call-by-value and the lambda-calculus". *Theoretical Computer Science*, 1(2):125-159, 1975.

[8] SML/NJ. Standard ML of New Jersey.

[9] Chicken Scheme. Scheme implementation using CPS compilation.

[10] GHC. Glasgow Haskell Compiler.

[11] Bartosz Milewski. *Category Theory for Programmers*. 2019.

[12] Olivier Danvy and Andrzej Filinski. "Abstracting Control". *LFP*, 1990.

[13] Timothy Griffin. "A Formulae-as-Types Notion of Control". *POPL*, 1990.

[14] Simon Marlow (ed.). *Haskell 2010 Language Report*. 2010.

[15] Philip Wadler. "The Essence of Functional Programming". *POPL*, 1992.

[16] P3941R4 - "Scheduler Affinity" (Dietmar Kühl, 2026).

[17] P4151R1 - "Rename affine_on" (Robert Leahy, 2026).

[18] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989. ISBN 0-13-114984-9.

[19] Jean-Yves Girard. "Linear Logic". *Theoretical Computer Science*, 50(1):1-102, 1987.

[20] W. Daniel Hillis and Guy L. Steele Jr. "Data Parallel Algorithms". *Communications of the ACM*, 29(12):1170-1183, 1986.

[21] P2079R10. Lucian Radu Teodorescu, Ruslan Arutyunyan, Lee Howes, Michael Voss. "Parallel Scheduler." 2025.

[22] Mads Tofte and Jean-Pierre Talpin. "Implementation of the Typed Call-by-Value λ-calculus using a Stack of Regions". *POPL*, 1994.

[23] P3796R1. Dietmar Kühl. "Coroutine Task Issues." 2025.

[24] P3980R1 - "Task's Allocator Use" (Dietmar Kühl, 2026).

[25] stdexec. NVIDIA's reference implementation of `std::execution`.

[26] P3826R5. Eric Niebler. "Fix Sender Algorithm Customization." 2026.

[27] sender-examples. Steve Downey. Example code for C++Now talk.

[28] backtrack.cpp. Backtracking search example in sender-examples.

[29] any_sender_of.hpp. Type-erased sender facility in stdexec (not part of C++26).

[30] retry.hpp. Retry algorithm example in stdexec.

[31] Corosio. Coroutine-native networking for C++.