Producing Senders from Coroutine-Native Code

Document Number:	P4093R1
Date:	2026-05-01
Intent:	Inform
Audience:	LEWG

Reply-to:	Vinnie Falco vinnie.falco@gmail.com Steve Gerbino steve@gerbino.co C++ Alliance Proposal Team

Table of Contents

Abstract

Coroutine-native awaitables can be wrapped as senders, but compound I/O results must be reduced to an error code before crossing the bridge.

An `IoAwaitable` (P4003R3 [1]) can be wrapped as a `std::execution` sender. Awaitables returning `void` or a single value map to `set_value`. Awaitables returning `error_code` map to `set_value()` on success and `set_error(ec)` on failure - no exceptions. Awaitables returning compound I/O results - any tuple-like whose first element is `error_code` with additional elements - are rejected at compile time. The coroutine body is the translation layer: it inspects the compound result, reduces it to an `error_code`, and returns that. The bridge routes the `error_code` through the three channels without exceptions.

Revision History

R1: May 2026 (pre-Brno mailing)

- Formatting corrections.

R0: April 2026 (post-Croydon mailing)

- Initial version.

1. Disclosure

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

This paper is part of the Network Endeavor (P4100R0), a project to bring coroutine-native I/O to C++.

The author developed and maintains Capy [2] and Corosio [3] and believes coroutine-native I/O is a practical foundation for networking in C++.

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

P4092R0 [4] showed the sender-to-awaitable direction. This paper shows the reverse. The bridge depends on Capy [2] and `beman::execution` [5], a community implementation of `std::execution` (P2300R10 [6]). The complete implementation is in Appendix A.

This paper asks for nothing.

2. The Bridge

`as_sender` wraps any `IoAwaitable` as a `std::execution` sender. The receiver's environment carries the I/O execution context:

```
capy::thread_pool pool;

auto sndr = capy::as_sender(capy::delay(500ms));

auto op = ex::connect(
    std::move(sndr),
    demo_receiver{
        {pool.get_executor(), std::stop_token{}},
        &done});

ex::start(op);
```

The receiver's environment answers a `get_io_executor` query with the pool's executor. The adapter extracts it, builds an `io_env`, and feeds it to `await_suspend(h, io_env const*)`. When the awaitable completes, the bridge calls `set_value` on the receiver.

```
main thread: 4256
  starting delay...
  set_value on thread 43448
  delay completed
```

The delay ran on a pool worker. Zero allocation beyond the coroutine frame.

---

3. The Three-Channel Problem

The bridge works for `delay`. What about `read_some`?

`read_some` returns `io_result<size_t>` - an `(error_code, size_t)` pair. The adapter must route this through three channels. The sender model provides three completion channels: `set_value` for success, `set_error` for failure, and `set_stopped` for cancellation. Algorithms like `when_all`, `upon_error`, and `retry` key on which channel fires. P4090R0 [7] documented the trade-off: route the whole pair through `set_value` and the composition algebra is bypassed; decompose it and the byte count is destroyed on error because `set_error` carries only the `error_code`. Neither option preserves both values and retains composition. The same finding now appears inside the bridge adapter itself.

There is a floor below which compound results should not cross into the sender channel model.

---

4. The Abstraction Floor

The solution is to not bridge compound results directly. The adapter inspects the return type structurally and rejects any tuple-like whose first element is `error_code` with additional elements:

```
template<class IoAw>
auto as_sender(IoAw&& aw)
{
    using R = decltype(
        std::declval<std::decay_t<IoAw>&>()
            .await_resume());
    static_assert(
        !detail::is_compound_ec_result_v<
            std::decay_t<R>>,
        "as_sender does not accept awaitables "
        "whose result destructures into "
        "(error_code, ...). Wrap the "
        "operation in a task<error_code> "
        "that inspects the compound result "
        "and returns the error code.");
    return awaitable_sender<std::decay_t<IoAw>>{
        std::forward<IoAw>(aw)};
}
```

The constraint is structural, not nominal. It does not name `io_result`. It asks: does the return type have a tuple protocol, is element 0 `error_code`, and are there additional elements? This catches `io_result<size_t>`, `std::tuple<error_code, size_t>`, `std::pair<error_code, size_t>`, or any user-defined type with the same shape. `std::expected<size_t, error_code>` is not caught - it lacks the tuple protocol. The constraint targets the most common I/O result shapes.

Awaitables returning a bare `error_code` - or a single-element tuple-like whose sole element is `error_code` - are binary outcomes. The bridge routes them: `set_value()` when zero, `set_error(ec)` otherwise. No exceptions.

---

5. Above and Below

The bridge inspects the `await_resume` return type structurally and selects the channel mapping at compile time:

`await_resume` type	Example API	`tuple_size`	Element 0	Bridge behavior
`void`	`delay(500ms)`	N/A	N/A	`set_value()`
`error_code`	`task<error_code>`	N/A	N/A	`set_value()` / `set_error(ec)`
`io_result<>`	`stream.connect(ep)`	1	`error_code`	`set_value()` / `set_error(ec)`
`int`, `string`, etc.	`task<int>`	N/A	N/A	`set_value(T)`
`io_result<size_t>`	`stream.read_some(buf)`	2	`error_code`	**rejected**
`tuple<error_code, size_t>`	-	2	`error_code`	**rejected**
`pair<error_code, size_t>`	-	2	`error_code`	**rejected**

The first four rows are above the abstraction floor. The channels work: `when_all` cancels siblings on I/O failure, `upon_error` is reachable, `retry` fires. No exceptions.

The last three rows are below the floor. Rejected at compile time.

---

6. The Translation Layer

To use I/O in a sender pipeline, wrap it in a `task<error_code>` that inspects the compound result and returns the error code:

```
capy::task<std::error_code>
read_all(auto& stream, auto buf)
{
    auto [ec, n] = co_await capy::read(
        stream, buf);
    if (ec)
        co_return ec;
    // use n...
    co_return {};
}

auto sndr = capy::as_sender(
        read_all(stream, buf))
    | ex::upon_error(
        [](std::error_code ec) {
            std::cout << "read failed: "
                      << ec.message() << "\n";
        });
```

The `task<error_code>` lives above the floor. The `co_await capy::read(stream, buf)` lives below it. The coroutine body is the translation layer: inspect the compound result, perform application logic, return the error code. The bridge routes it through the three channels. No exceptions.

Cost: one coroutine frame per I/O operation that crosses the sender boundary. `as_sender(stream.read_some(buf))` is a compile error, not a silent loss of error information.

---

7. P3552R3 Analysis

P3552R3 [8] defines `std::execution::task<T>`, a coroutine type that is also a sender. Its completion signature is `set_value_t(T)`. When `T` is `std::pair<error_code, size_t>`, the compound result lands on the value channel. This is "just use `set_value`" (P4090R0 [7] Section 5): `upon_error` is unreachable, `when_all` does not cancel siblings on I/O failure, `retry` does not fire. The programmer who writes `task<std::pair<error_code, size_t>>` has silently opted into "just use `set_value`":

```
std::execution::task<
    std::pair<std::error_code, std::size_t>>
read_some_task(auto& stream, auto buf)
{
    auto [ec, n] = co_await stream.read_some(
        buf);
    co_return std::pair{ec, n};
}

auto sndr = read_some_task(stream, buf)
    | ex::upon_error(
        [](std::error_code ec) {
            // unreachable
        });
```

`task` is general-purpose. A `static_assert` rejecting compound `error_code` results would be too broad. The constraint belongs at a bridge point with I/O intent, not on the general-purpose coroutine type. A programmer who uses `task<pair<error_code, size_t>>` directly gets the value-channel behavior documented in P4090R0 [7] Section 5 - both values preserved, composition algebra bypassed.

P3552R3 [8] converts unhandled `set_error` to an exception via `AS-EXCEPT-PTR`. The observation is architectural: `as_sender` enforces the abstraction floor at the IoAwaitable-to-sender boundary. `task` does not enforce it. The programmer chooses where the floor lives.

---

8. `split_ec`

A sender adapter can enforce the floor inside the pipeline:

```
do_read(sock, buf)          // sender completing with
                            //   set_value(error_code)
    | split_ec()            // set_value() or
                            //   set_error(ec)
    | ex::upon_error(
        [](std::error_code ec) {
            // reachable, no exceptions
        });
```

`split_ec` advertises both `set_value_t()` and `set_error_t(std::error_code)` and selects between them at runtime. The implementation is a receiver adapter - no type erasure, no variant sender, no allocation. The complete implementation is in Capy [2].

Three enforcement points, one abstraction floor. `as_sender` enforces it at the IoAwaitable-to-sender boundary. `split_ec` enforces it inside the pipeline. `task` does not enforce it. The programmer chooses where the floor lives.

---

9. Conclusion

The compound result stays below the floor. The binary outcome crosses above it. The channels work.

---

10. Acknowledgments

The authors thank Dietmar Kühl for `beman::execution` [5] and for the channel-routing enumeration in P2762R2 [9], Michał Dominiak, Eric Niebler, and Lewis Baker for `std::execution`, Chris Kohlhoff for identifying the partial-success problem in P2430R0 [10], Kirk Shoop for the completion-token heuristic analysis in P2471R1 [11], Fabio Fracassi for P3570R2 [12], Peter Dimov for the refined channel mapping, and Ville Voutilainen for reflector discussion on the abstraction floor.

References

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

[2] cppalliance/capy - Coroutine primitives library.

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

[4] P4092R0 - "Consuming Senders from Coroutine-Native Code" (Vinnie Falco, Steve Gerbino, 2026).

[5] bemanproject/execution - Community implementation of `std::execution`.

[6] P2300R10 - "std::execution" (Michał Dominiak et al., 2024).

[7] P4090R0 - "Sender I/O: A Constructed Comparison" (Vinnie Falco, Steve Gerbino, 2026).

[8] P3552R3 - "Add a Coroutine Task Type" (Dietmar Kühl, Maikel Nadolski, 2025).

[9] P2762R2 - "Sender/Receiver Interface For Networking" (Dietmar Kühl, 2023).

[10] P2430R0 - "Slides: Partial success scenarios with P2300" (Chris Kohlhoff, 2021).

[11] P2471R1 - "NetTS, ASIO and Sender Library Design Comparison" (Kirk Shoop, 2021).

[12] P3570R2 - "Optional variants in sender/receiver" (Fabio Fracassi, 2025).

Appendix A. Bridge Implementation

```
#include <boost/capy/concept/io_awaitable.hpp>
#include <boost/capy/ex/executor_ref.hpp>
#include <boost/capy/ex/io_env.hpp>
#include <boost/capy/io_result.hpp>

#include <beman/execution/execution.hpp>

#include <concepts>
#include <coroutine>
#include <exception>
#include <stop_token>
#include <tuple>
#include <type_traits>
#include <utility>

namespace boost::capy {

struct get_io_executor_t
{
    template<class Env>
    auto operator()(
        Env const& env) const noexcept
        -> decltype(env.query(
            std::declval<
                get_io_executor_t const&>()))
    {
        return env.query(*this);
    }
};

inline constexpr get_io_executor_t
    get_io_executor{};

struct io_sender_env
{
    executor_ref io_executor;
    std::stop_token stop_token;

    auto query(
        get_io_executor_t const&)
            const noexcept -> executor_ref
    {
        return io_executor;
```

```
    }

    auto query(
        beman::execution::get_stop_token_t
            const&) const noexcept
        -> std::stop_token
    {
        return stop_token;
    }
};

namespace detail {

template<class T, class = void>
struct has_tuple_protocol
    : std::false_type {};

template<class T>
struct has_tuple_protocol<T,
    std::void_t<
        typename std::tuple_size<T>::type>>
    : std::bool_constant<
        (std::tuple_size<T>::value > 0)> {};

template<class T,
    bool = has_tuple_protocol<T>::value>
struct is_ec_outcome
    : std::is_same<T, std::error_code> {};

template<class T>
struct is_ec_outcome<T, true>
    : std::bool_constant<
        std::tuple_size_v<T> == 1 &&
        std::is_same_v<
            std::tuple_element_t<0, T>,
            std::error_code>>
{};

template<class T>
constexpr bool is_ec_outcome_v =
    std::is_same_v<T, std::error_code> ||
    is_ec_outcome<T>::value;

template<class T,
    bool = has_tuple_protocol<T>::value>
```

```
struct is_compound_ec_result
    : std::false_type {};

template<class T>
struct is_compound_ec_result<T, true>
    : std::bool_constant<
        std::tuple_size_v<T> >= 2 &&
        std::is_same_v<
            std::tuple_element_t<0, T>,
            std::error_code>>
{};

template<class T>
constexpr bool is_compound_ec_result_v =
    is_compound_ec_result<T>::value;

template<class IoAw, class Receiver>
struct bridge_task
{
    struct promise_type;
    using handle_type =
        std::coroutine_handle<promise_type>;

    struct promise_type
    {
        io_env const* env_ = nullptr;

        bridge_task get_return_object() noexcept
        {
            return bridge_task{
                handle_type::from_promise(
                    *this)};
        }

        std::suspend_always
        initial_suspend() noexcept
        {
            return {};
        }

        std::suspend_always
        final_suspend() noexcept
        {
            return {};
        }
```

```
    void return_void() noexcept {}
    void unhandled_exception() noexcept {}

    template<class A>
    struct transform_awaiter
    {
        std::decay_t<A>& aw_;
        promise_type* p_;

        bool await_ready() noexcept
        {
            return aw_.await_ready();
        }

        decltype(auto) await_resume()
        {
            return aw_.await_resume();
        }

        auto await_suspend(
            std::coroutine_handle<> h)
                noexcept
        {
            return aw_.await_suspend(
                h, p_->env_);
        }
    };

    template<class A>
    auto await_transform(A&& a)
    {
        return transform_awaiter<A>{
            a, this};
    }
};

handle_type h_{};

~bridge_task()
{
    if(h_)
        h_.destroy();
}
```

```
    bridge_task() noexcept = default;

    bridge_task(bridge_task&& o) noexcept
        : h_(std::exchange(o.h_, {}))
    {
    }

    bridge_task& operator=(
        bridge_task&& o) noexcept
    {
        if(h_)
            h_.destroy();
        h_ = std::exchange(o.h_, {});
        return *this;
    }

    bridge_task(
        bridge_task const&) = delete;
    bridge_task& operator=(
        bridge_task const&) = delete;

private:
    explicit bridge_task(
        handle_type h) noexcept
        : h_(h)
    {
    }
};

} // namespace detail

template<class IoAw>
struct awaitable_sender
{
    using sender_concept =
        beman::execution::sender_t;

    using result_type = decltype(
        std::declval<std::decay_t<IoAw>&>()
            .await_resume());

    static auto make_sigs()
    {
        if constexpr (
            std::is_void_v<result_type>)
```

```
        return beman::execution::
            completion_signatures<
                beman::execution::
                    set_value_t(),
                beman::execution::
                    set_error_t(
                        std::exception_ptr),
                beman::execution::
                    set_stopped_t()>{};
    else if constexpr (
        detail::is_ec_outcome_v<
            result_type>)
        return beman::execution::
            completion_signatures<
                beman::execution::
                    set_value_t(),
                beman::execution::
                    set_error_t(
                        std::error_code),
                beman::execution::
                    set_error_t(
                        std::exception_ptr),
                beman::execution::
                    set_stopped_t()>{};
    else
        return beman::execution::
            completion_signatures<
                beman::execution::
                    set_value_t(result_type),
                beman::execution::
                    set_error_t(
                        std::exception_ptr),
                beman::execution::
                    set_stopped_t()>{};
}

using completion_signatures =
    decltype(make_sigs());

IoAw aw_;

template<class Receiver>
struct op_state
{
    using operation_state_concept =
```

```
        beman::execution::operation_state_t;

    IoAw aw_;
    Receiver rcvr_;
    io_env env_;
    detail::bridge_task<IoAw, Receiver>
        bridge_;

    op_state(IoAw aw, Receiver rcvr)
        : aw_(std::move(aw))
        , rcvr_(std::move(rcvr))
    {
    }

    op_state(op_state const&) = delete;
    op_state(op_state&&) = delete;
    op_state& operator=(
        op_state const&) = delete;
    op_state& operator=(
        op_state&&) = delete;

    void start() noexcept
    {
        auto renv =
            beman::execution::get_env(
                rcvr_);
        auto ex = get_io_executor(renv);

        std::stop_token st;
        if constexpr (requires {
            { renv.query(
                beman::execution::
                    get_stop_token_t{}) }
                -> std::convertible_to<
                    std::stop_token>; })
        {
            st = renv.query(
                beman::execution::
                    get_stop_token_t{});
        }

        env_ = io_env{ex, st, nullptr};

        bridge_ = [](
            IoAw aw,
```

```
    Receiver rcvr,
    std::stop_token const* st)
    -> detail::bridge_task<
        IoAw, Receiver>
{
    try
    {
        if constexpr (
            std::is_void_v<
                result_type>)
        {
            co_await std::move(aw);
            if (st->stop_requested())
                beman::execution::
                    set_stopped(
                        std::move(
                            rcvr));
            else
                beman::execution::
                    set_value(
                        std::move(
                            rcvr));
        }
        else if constexpr (
            detail::is_ec_outcome_v<
                result_type>)
        {
            auto result =
                co_await
                    std::move(aw);
            if (st->stop_requested())
            {
                beman::execution::
                    set_stopped(
                        std::move(
                            rcvr));
            }
            else
            {
                std::error_code ec;
                if constexpr (
                    std::is_same_v<
                        result_type,
                        std::error_code
                        >)
```

```
                    ec = result;
                else
                    ec = get<0>(
                        result);
                if (!ec)
                    beman::execution
                        ::set_value(
                        std::move(
                            rcvr));
                else
                    beman::execution
                        ::set_error(
                        std::move(
                            rcvr),
                        ec);
            }
        }
        else
        {
            auto result =
                co_await
                    std::move(aw);
            if (st->stop_requested())
                beman::execution::
                    set_stopped(
                        std::move(
                            rcvr));
            else
                beman::execution::
                    set_value(
                        std::move(
                            rcvr),
                        std::move(
                            result));
        }
    }
    catch(...)
    {
        beman::execution::
            set_error(
                std::move(rcvr),
                std::current_exception
                    ());
    }
}(std::move(aw_),
```

```
                std::move(rcvr_),
                &env_.stop_token);

            bridge_.h_.promise().env_ =
                &env_;
            bridge_.h_.resume();
        }
    };

    template<class Receiver>
    auto connect(Receiver rcvr) &&
        -> op_state<Receiver>
    {
        return op_state<Receiver>(
            std::move(aw_),
            std::move(rcvr));
    }

    template<class Receiver>
    auto connect(Receiver rcvr) const&
        -> op_state<Receiver>
    {
        return op_state<Receiver>(
            aw_, std::move(rcvr));
    }
};

template<class IoAw>
auto as_sender(IoAw&& aw)
{
    using R = decltype(
        std::declval<std::decay_t<IoAw>&>()
            .await_resume());
    static_assert(
        !detail::is_compound_ec_result_v<
            std::decay_t<R>>,
        "as_sender does not accept awaitables "
        "whose result destructures into "
        "(error_code, ...). Wrap the "
        "operation in a task<error_code> "
        "that inspects the compound result "
        "and returns the error code.");
    return awaitable_sender<
        std::decay_t<IoAw>>{
            std::forward<IoAw>(aw)};
```

```
}

namespace detail {

template<class Sender>
struct split_ec_sender
{
    using sender_concept =
        beman::execution::sender_t;

    using completion_signatures =
        beman::execution::completion_signatures<
            beman::execution::set_value_t(),
            beman::execution::set_error_t(
                std::error_code),
            beman::execution::set_error_t(
                std::exception_ptr),
            beman::execution::set_stopped_t()>;

    Sender sndr_;

    template<class Receiver>
    struct ec_receiver
    {
        using receiver_concept =
            beman::execution::receiver_t;

        Receiver rcvr_;

        auto get_env() const noexcept
        {
            return beman::execution::get_env(
                rcvr_);
        }

        void set_value(
            std::error_code ec) && noexcept
        {
            if (!ec)
                beman::execution::set_value(
                    std::move(rcvr_));
            else
                beman::execution::set_error(
                    std::move(rcvr_), ec);
        }
```

```
    void set_value() && noexcept
    {
        beman::execution::set_value(
            std::move(rcvr_));
    }

    template<class E>
    void set_error(E&& e) && noexcept
    {
        beman::execution::set_error(
            std::move(rcvr_),
            std::forward<E>(e));
    }

    void set_stopped() && noexcept
    {
        beman::execution::set_stopped(
            std::move(rcvr_));
    }
};

template<class Receiver>
struct op_state
{
    using operation_state_concept =
        beman::execution::
            operation_state_t;

    using inner_op_t = decltype(
        beman::execution::connect(
            std::declval<Sender>(),
            std::declval<
                ec_receiver<Receiver>>()));

    inner_op_t op_;

    op_state(Sender sndr, Receiver rcvr)
        : op_(beman::execution::connect(
            std::move(sndr),
            ec_receiver<Receiver>{
                std::move(rcvr)}))
    {
    }
```

```
        op_state(op_state const&) = delete;
        op_state(op_state&&) = delete;
        op_state& operator=(
            op_state const&) = delete;
        op_state& operator=(
            op_state&&) = delete;

        void start() noexcept
        {
            beman::execution::start(op_);
        }
    };

    template<class Receiver>
    auto connect(Receiver rcvr) &&
        -> op_state<Receiver>
    {
        return op_state<Receiver>(
            std::move(sndr_),
            std::move(rcvr));
    }

    template<class Receiver>
    auto connect(Receiver rcvr) const&
        -> op_state<Receiver>
    {
        return op_state<Receiver>(
            sndr_, std::move(rcvr));
    }
};

} // namespace detail

template<class Sender>
auto split_ec(Sender&& sndr)
{
    return detail::split_ec_sender<
        std::decay_t<Sender>>{
            std::forward<Sender>(sndr)};
}

} // namespace boost::capy
```